**C# Framework**

[C# 2.0 6](#_Toc374451877)

[1. Generics 6](#_Toc374451878)

[2. Nullable types 6](#_Toc374451879)

[3. Anonymous methods 7](#_Toc374451880)

[4. Method Group Conversions 7](#_Toc374451881)

[5. Partial Types/Partial Classes 8](#_Toc374451882)

[C# 3.0 9](#_Toc374451883)

[1. Auto-Implemented Properties 9](#_Toc374451884)

[2. LINQ 10](#_Toc374451885)

[a) Query Expression Syntax 10](#_Toc374451886)

[b) Implicitly Typed Variables/Type Inference 10](#_Toc374451887)

[c) Reason of introduction of var variable in C# 11](#_Toc374451888)

[d) Lambda expressions 12](#_Toc374451889)

[e) Extension Methods 12](#_Toc374451890)

[Pointers/Rules while using Extension Methods 12](#_Toc374451891)

[Rules for Creating Extension Methods 13](#_Toc374451892)

[Why do you need Extension Methods 13](#_Toc374451893)

[f) Anonymous types: 14](#_Toc374451894)

[Scope of Anonymous Types 14](#_Toc374451895)

[No Type Duplication 15](#_Toc374451896)

[Usefulness 15](#_Toc374451897)

[Equalness 15](#_Toc374451898)

[g) Objects/Collections Initializers 15](#_Toc374451899)

[C# 4.0 16](#_Toc374451900)

[1. Dynamic Keyword 16](#_Toc374451901)

[Why use dynamic keyword 17](#_Toc374451902)

[var versus dynamic 17](#_Toc374451903)

[Reflection versus dynamic keyword 17](#_Toc374451904)

[2. Named and Optional Paramters 18](#_Toc374451905)

[3. Covariance and Contravariance 19](#_Toc374451906)

[4. Garbage Collection Enhancements 22](#_Toc374451907)

[5. DLR (Dynamic Language Runtime) 23](#_Toc374451908)

[C# 5.0 24](#_Toc374451909)

[Object Oriented Programming Basics 25](#_Toc374451910)

[1. Encapsulation 25](#_Toc374451911)

[2. Inheritance 25](#_Toc374451912)

[3. Polyorphism 25](#_Toc374451913)

[Example of Compile Time Polymorphism 26](#_Toc374451914)

[Example of Run Time Polymorphism 26](#_Toc374451915)

[Virtual Methods 26](#_Toc374451916)

[Important Points with regards to Example Practiced 28](#_Toc374451917)

[Interface 30](#_Toc374451918)

[Multiple Inheritance using Interface 30](#_Toc374451919)

[Blocking Inheritance 31](#_Toc374451920)

[What is Explicit Interface Implementation 32](#_Toc374451921)

[Abstract Classes 33](#_Toc374451922)

[Limitations of Abstract Class and Abstract method 33](#_Toc374451923)

[When should a class have to be absolutely declared as abstract 33](#_Toc374451924)

[Practical Usage of abstract Class 33](#_Toc374451925)

[Difference between Interface/Abstract Classes/Virtual Methods 34](#_Toc374451926)

[Access Modifiers 35](#_Toc374451927)

[Memory Management 37](#_Toc374451928)

[Value Types versus Reference Types 37](#_Toc374451929)

[Differences between Structs and Classes 40](#_Toc374451930)

[What is Managed Heap 41](#_Toc374451931)

[Delegates 42](#_Toc374451932)

[Action<T> / Function<T> / Predicate Delegates 44](#_Toc374451933)

[Anonymous Methods 48](#_Toc374451934)

[Lambda Expressions 49](#_Toc374451935)

[Events 53](#_Toc374451936)

[Definition 53](#_Toc374451937)

[Event Syntax Components 53](#_Toc374451938)

[EventArgs and CustomEventArgs 56](#_Toc374451939)

[Events returning void type 56](#_Toc374451940)

[Advantages of Events over Delegates 56](#_Toc374451941)

[Misc Event Concepts 57](#_Toc374451942)

[Memory Leak through Events 57](#_Toc374451943)

[Garbage Collection 61](#_Toc374451944)

[Garbage Collection Phases 62](#_Toc374451945)

[Generations 62](#_Toc374451946)

[Performance 63](#_Toc374451947)

[Concept of Finalization 64](#_Toc374451948)

[Large Object Heap 69](#_Toc374451949)

[Important Concepts 72](#_Toc374451950)

[Object 72](#_Toc374451951)

[Equals 72](#_Toc374451952)

[Finalize 72](#_Toc374451953)

[GetHashCode 73](#_Toc374451954)

[GetType 73](#_Toc374451955)

[ToString 73](#_Toc374451956)

[ReferenceEquals 74](#_Toc374451957)

[System.ValueType 75](#_Toc374451958)

[Composition versus Inheritance 75](#_Toc374451959)

[Difference between decimal, double and float 76](#_Toc374451960)

[Weak References 77](#_Toc374451961)

[Disadvantage: 78](#_Toc374451962)

[Closures 79](#_Toc374451963)

[1. Concept of Closure: 79](#_Toc374451964)

[2. Side Effects of Closure 79](#_Toc374451965)

[Difference between Equals() versus == 82](#_Toc374451966)

[For value types 82](#_Toc374451967)

[For Reference types 82](#_Toc374451968)

[IEquatable<T> 85](#_Toc374451969)

[IComparable versus IComparer 86](#_Toc374451970)

[Specific string Implementation 86](#_Toc374451971)

[IEnumerable versus IEnumerator 87](#_Toc374451972)

[IEnumerable versus IQueryable 88](#_Toc374451973)

[IEnumerator, ICollection, IDictionary and IList 89](#_Toc374451974)

[Reflection 90](#_Toc374451975)

[What is reflection ? 90](#_Toc374451976)

[How to Implement Reflection 90](#_Toc374451977)

[Benefits of Reflection 90](#_Toc374451978)

[Practical Use-Cases of Reflection 90](#_Toc374451979)

[Type Class 90](#_Toc374451980)

[Difference between GetType() and typeof 91](#_Toc374451981)

[Difference between Reflection and dynamic keyword 92](#_Toc374451982)

[Some background on dynamic keyword: 92](#_Toc374451983)

[Static Variables/Methods/Classes 93](#_Toc374451984)

[Static Methods 93](#_Toc374451985)

[Static Classes 94](#_Toc374451986)

[Difference between const, readonly and static 95](#_Toc374451987)

[Indexer 96](#_Toc374451988)

[Difference between Control.BeginInvoke and Delegate.BeginInvoke 96](#_Toc374451989)

[Boxing and UnBoxing 97](#_Toc374451990)

[Fields 98](#_Toc374451991)

[Properties 98](#_Toc374451992)

[Parse & TryParse 99](#_Toc374451993)

[Pass By Value and Pass By Reference 100](#_Toc374451994)

[Ref and Out keywords 101](#_Toc374451995)

[i++ and ++i 102](#_Toc374451996)

[Characteristics of Recursion 103](#_Toc374451997)

[Use-Cases of Recursion 103](#_Toc374451998)

[Application Domains 105](#_Toc374451999)

[Enumeration 107](#_Toc374452000)

[Yield Keyword 107](#_Toc374452001)

[Deep Copy versus Shallow Copy 107](#_Toc374452002)

[Control 107](#_Toc374452003)

[Initialization of variable 107](#_Toc374452004)

[Constructors 108](#_Toc374452005)

[Net Architecture 109](#_Toc374452006)

[Exception Handling 110](#_Toc374452007)

[Regular Expressions 111](#_Toc374452008)

[Regex vs. String Methods 111](#_Toc374452009)

[Static vs. Instance Methods 111](#_Toc374452010)

[Method for Regex Algorithm 112](#_Toc374452011)

[Interview Questions – 113](#_Toc374452012)

[Related Questions – 113](#_Toc374452013)

[C# Questions 115](#_Toc374452014)

[Framework Concepts 115](#_Toc374452015)

[.Net Basics 116](#_Toc374452016)

[Object Oriented Concepts 118](#_Toc374452017)

[Delegates 120](#_Toc374452018)

[Garbage Collection 121](#_Toc374452019)

[Exception Handling 126](#_Toc374452020)

[Misc 127](#_Toc374452021)

[Performance Profiling Links 127](#_Toc374452022)

|  |
| --- |
| **C# 2.0** (released with .NET 2.0 and VS2005) |
| **Features:**   1. Generics 2. Nullable Types 3. Anonymous Methods 4. Method Group Conversions 5. Partial Types/Partial Classes |
| **C# 3.0** (released with .NET 3.5 and VS2008) |
| **Features:**   1. Auto-Implemented Properties 2. LINQ 3. Query Expression 4. var type or type Inference 5. Lambda Expressions (replaces Anonymous Methods) 6. Extension methods 7. Anonymous types 8. Object Initializers |
| **C# 4.0** **(**released with .NET 4 and VS2010**)** – Aug 2010 |
| **Features:**   1. Late Binding (Dynamic) 2. Optional and Named Parameters 3. Co-variance and Contra-variance 4. Garbage Collection Enhancements 5. Dynamic Language Runtime (DLR) |
| **C# 5.0** **(**released with .NET 4.5**) –** Aug 2012 |
| **Features:**   1. Async programming 2. Caller info attributes |

C# 2.0

1. Generics

* Generics refers to a technique of writing code for a class without specifying the data type that the class works with. Essentially they allow you to define type-safe data structures, without committing to actual data types.
* You specify the data type when you declare an instance of a generic class. This allows generic classes to be specialized for many different data types without having to rewrite those classes.
* Generics are the most powerful feature of C# 2.0

Advantages

* Re-usability - This results in a significant performance boost and higher quality code, because you get to reuse data processing algorithms without duplicating type-specific code.
* Type Safety Exists - As a result of generics being used the compiler can perform compile-time checks on code for type safety, i.e. are you trying to put an int into that list of strings? Using an ArrayList would cause that to be a less transparent runtime error.
* Performance - Faster than using objects as it either avoids boxing/unboxing (where .net has to convert value types to reference types or vice-versa) or casting from objects to the required reference type.

<http://beyondrelational.com/modules/2/blogs/61/posts/11217/all-about-generics-in-c.aspx>

2. Nullable types

The null value is useful for initializing reference types, but null is itself a reference, and we cannot assign it to a value type. The following statement is therefore illegal in C#:-

int i = null; // illegal

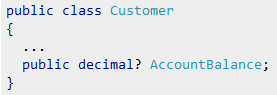
However, C# defines a modifier (Introduced in .Net 2.0 ) that we can use to declare that a variable is a nullable value type. A nullable value type behaves in a similar manner to the original value type, but we can assign the null value to it. We use the question mark (?) to indicate that a value type is nullable, like this:-

int? i = null; // legal   
OR   
Nullable<int>i = null

When would you use Nullable Value types:

Example: A real world scenario when your Database table contains nullable columns in it and you need to create objects which map to the columns in the database. To achieve such requirement nullable types are really very useful to create exactly same mapping and allow user to easily work in this scenario.

* If the columns are reference type that is String such as (email address and customer address), there is not a problem as you can defined it as null in C#
* But if the columns are value type that is double such as (customer account balance), you cannot map it to C# without using nullable types.



* Everytime in code we have to do a null check before performing some operation over value types. Having them declared as nullable value types is easier to implement.

3. Anonymous methods

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.

Example 1:

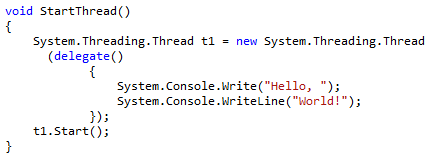
  
Example 2:



Advantages

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.



Further Comments

* 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.

4. Method Group Conversions

This simplifies the syntax used to assign a method to a delegate. Method group conversion allows you to assign the name of a method to a delegate, without the use new or explicitly invoking the delegate's constructor.

delegate string StrMod(string str);

In the past, to assign a method called removeSpaces( ) to that delegate, you would use a statement like:

strOp = new StrMod(removeSpaces);

With the addition of method group conversion, this statement can now be written more compactly:

strOp = removeSpaces;

5. Partial Types/Partial Classes

Partial types allow a single type to be built from multiple source files. This is particularly useful with autogenerated code, where the tool (e.g. a GUI designer) can "own" one file, and the developer can work in a different one. The partial keyword is used to indicate that the type may span multiple files. Partial types apply to interfaces, classes and structs - but not enums or delegates.

The concept of partial classes was introduced with .NET 2.0. The essential idea is that a single class can be defined across multiple files. Each file contains a portion of the class. The files can even be of different file types; such as .cs and.xaml. The compiler is responsible for dynamically combining the code into a single class as it is compiled. You can identify partial classes by the partial keyword in C#, and Partial in VB. Partial classes are also helpful in scenarios where some of the source code is generated by a tool but needs to be extended manually by a developer. The generated code and handwritten code can be kept in separate files.

C# 3.0

Properties (Background)

Properties are an extension of fields but they do not designate storage locations. Properties provide a flexible mechanism to read, write, or compute the values of private fields through accessors that read, write, or compute their values.

Backing Field Support for Properties (Background)

Typically a property on a class is declared in two steps:

1. Define a private backing field. For example: *private string \_name;*
2. Define public properties that encapsulate the backing field. For the above backing field the property would be declared like:

*private string \_name;*

*public string Name   
{   
 get   
 {   
 return \_name;   
 }   
 set   
 {  
 \_ name = value;   
 }   
 }*

Why do we need to encapsulate the field when we are doing just nothing in the property getter and setter?

Before accessing the field some business validation may be required. But if the fields are made public in the first release, the client code will start consuming the fields directly. In future when fields are encapsulated by property methods, to enforce business rules, it will break the code contract with the consumers. To avoid this problem we expose public properties backed by private fields."

1. Auto-Implemented Properties

* Auto-implemented properties make property-declaration more concise.
* Simple way to write properties which just get and set their values directly from/to a backing variable
* The compiler will create a private, anonymous backing field that is not accessible except through the property getter and setter.

|  |  |
| --- | --- |
| C# 3.0 (Auto-Implemented Properties) | Regular way with Backing variables |
|  |  |

2. LINQ

* LINQ is a Microsoft .NET Framework component that adds native data querying capabilities to .NET languages
* LINQ extends the language by the addition of query expressions, which are akin to SQL statements, and can be used to conveniently extract and process data from arrays, enumerable classes, XML documents, relational databases, and third-party data sources.
* LINQ defines a set of method names called as standard query operators

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | Select | 5 | Join/GroupJoin | 9 | Reverse |
| 2 | Where | 6 | Take/TakeWhile | 10 | GroupBy |
| 3 | SelectMany | 7 | Skip/SkipWhile | 11 | Distinct |
| 4 | Sum/Min/Max/Average | 8 | OrderBy/ThenBy | 12 | First/FistOrDefault |

* LINQ Providers
  + LINQ to Objects
  + LINQ to SQL
  + LINQ to SQL
  + LINQ to DataSets

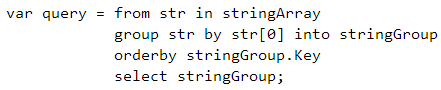
What is Query Expression?

A query expression is a query expressed in query syntax. A query expression is a first-class language construct. It is just like any other expression and can be used in any context in which a C# expression is valid. A query expression consists of a set of clauses written in a declarative syntax similar to SQL or XQuery.

A query expression must begin with a **from** clause and must end with a **select** or **group** clause. Between the first from clause and the last select or group clause, it can contain one or more of these optional clauses: where, orderby, join, let and even additional from clauses. You can also use the into keyword to enable the result of a join or group clause to serve as the source for additional query clauses in the same query expression.

a) Query Expression Syntax

* Query expressions use declarative syntax similar to SQL/XQuery to query over IEnumerable collections.
* At compile time query syntax is converted to method calls to a LINQ provider's implementation of the standard query operator extension methods.
* Applications control the standard query operators that are in scope by specifying the appropriate namespace with a using directive. The following query expression takes an array of strings, groups them according to the first character in the string, and orders the groups.



b) Implicitly Typed Variables/Type Inference

|  |  |
| --- | --- |
| Explicit Declaration of Data Type | Implicit Declaration of Data Type |
| int 1 = 0;  You are directly defining the data type | var i = 0;  By looking at the right hand side – during compilation - the left hand side data type is created  Data Type is defined **statically** (strongly typed) and not dynamically during runtime |

Type inference makes use of the var keyword.

Now you can infer the type assignment by simply using the **var** keyword as shown in the example below:

var firstname = “Scott”;  
var age = 28;  
var startdate = DateTime.Today;

This concept is important because any variables declared as var are equally strongly typed as their explicitly declared counterparts. After the variable has been declared and the type inferred, the variable’s type cannot be changed.

There are a few rules that need to be followed:

* The variable must be initialized. Otherwise the compiler doesn’t have anything to infer the type from.
* The initialize cannot be null. You can't use implicit typing (without casting) if the initializer is null literal

c) Reason of introduction of var variable in C#

This was done primarily with the introduction of LINQ and anonymous types so the results can be stored in a variable which are declared with “var”.

|  |  |
| --- | --- |
| Example 1 |  |

|  |  |
| --- | --- |
| Example 2: | |
| Option 1 – take the output in object but you don’t get any intelligence on the properties |  |
| Option 2 – you can create a class with the 2 properties |  |
| Option 3 – Use of var – don’t need to create a class, put it in IEnumerable etc |  |

d) Lambda expressions

Lambda expressions (expressions with => in them) are similar to anonymous methods in C# 2. They have been introduced since 3.0 framework. They can be used whenever you have a delegate as a parameter type.

*See example C:\\_Net Technical Material\Visual Studio 2010 Projects\01-C\_Sharp\_Concepts\aMine\Delegates\_Examples\*06-Lamda\_Expression\_Example  
  
For example, List<T> contains a ConvertAll method taking a delegate. To convert a List<int> into a List<double>, where each value in the resulting list is half the corresponding value in the original list, we could write:



Lambda expressions completely replace anonymous methods in C# 3. There are no good reasons ever to use an anonymous method any more.

*See detailed explanation within Delegates section below*

e) Extension Methods

* Extension method is a new feature in C# 3.0.
* They allow existing classes to be extended without relying on inheritance or having to change the class's source code.
* Extension methods are defined as static methods but are called by using instance method syntax.

This means that if you want to add some methods into the existing String class you can do it quite easily. The benefit on the client side is that it feels the method on the existing class. For example – String.Spellchecker (which can be implemented as extension method)

Pointers/Rules while using Extension Methods

* Extension methods cannot be used to override existing methods
* An extension method with the same name and signature as an instance method will not be called
* The concept of extension methods cannot be applied to fields, properties or events
* Use extension methods sparingly....overuse can be a bad thing!

Rules for Creating Extension Methods

1. Define a static class to contain the extension method. Side Note: Name of the static class doesn’t matter
2. Implement extension method as a static method with at least the same visibility as containing class.
3. The first parameter of the method specifies the type that the method operates on; it must be preceded with the “this” modifier.
4. In the calling code, add a using directive to specify the namespace that contains the extension method class.
5. Call the methods as if they were instance methods on the type.

Example: Here's an example of creating an extension method in C# that adds a RemoveNonNumeric() method to the String class. Notice that the class is defined as static as well as the extension method itself. The "this" keyword in the parameter signature tells the compiler to add the extension method to the String class since "string" follows the keyword.

|  |  |
| --- | --- |
| Extension Method Definition | Extension Method Usage |
|  |  |

Another Example

|  |  |
| --- | --- |
| Original Method | Extension Method Definition |
|  |  |

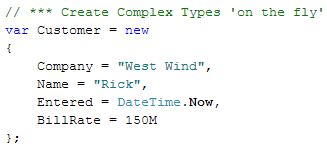
|  |  |
| --- | --- |
| Extension Method Intellisense | Extension Method Usage |
|  |  |

Why do you need Extension Methods

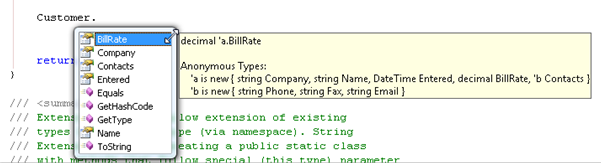
Because many of the existing MS methods are marked as sealed which means derived classes of them cannot be created. And hence from you calling code perspective – if you need to provide a few more methods which are not originally defined on the base class – extension methods are the standard methodology.

f) Anonymous types:

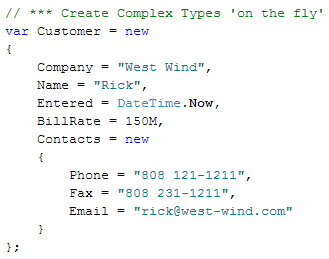
Anonymous Types are essentially compiler generated types that you don't explicitly declare with an official type declaration. Rather you define the type inline as part of the code where you need to use the new type.



This code creates a new instance of an anonymous type - in this case a simple object instance that holds customer data. Once you have declared the type like the above you can then use the type as if there was a full type declaration, so you can access Customer.Company, Customer.BillRate and so forth.



Notice that the C# compiler fixes up any members that are created and assigns an appropriate type to each. So as you can see in the screenshot BillRate is typed as decimal based on the inferred type which in this case is a literal decimal value. So Company is a string and Entered a DateTime member by the same logic. Type inference is not limited to literal values - so any typed value expression can be used as an input to force the type to be created. As long as the compiler can tell what the base value is the member can be created as a specific type. You can even take this one step further by nesting anonymous types like this:



Notice the Contacts member which is another anonymous type nested inside the first type.

Scope of Anonymous Types

Anonymous types are scoped only to the currently executing method. Because the generated type is anonymous there's no public name for the type that you can access. In other words you can't instantiate the type directly on your own and you can't reference the type outside of the method that created it.

No Type Duplication

|  |  |
| --- | --- |
| If you create two anonymous types that have the exact same type signature in the same assembly, the C# compiler is smart enough to consolidate the anonymous class definition and reuse that type. So if you have code like the following:  Only one class is created for the type. |  |

Usefulness

First there's use of this functionality as part of LINQ. If you are querying data and you want to filter your result field list by using projection into a custom type, Anonymous Types are used. So if you run a LINQ to SQL query for example:

|  |  |
| --- | --- |
| You are in fact creating an IQueryable<AnonymousType> which can then be turned into something like a List<AnonymousType>. The fact that an anonymous type can be created on the fly and that it can be done in a single line of code makes it possible for LINQ to utilize this functionality to create custom shaped result types. |  |

Equalness

|  |  |
| --- | --- |
| The Equals and GetHashcode methods on anonymous types are defined in terms of the Equals and GetHashcode of the properties, so that two instances of the same anonymous type are equal if and only if all their properties are equal. |  |

g) Objects/Collections Initializers

Object Initializer Example

|  |  |
| --- | --- |
|  |  |

Collection Initializer Example



C# 4.0

1. Dynamic Keyword

Visual C# 2010 introduces a new type, dynamic. The type is a static type, but an object of type dynamic bypasses static type checking. In most cases, it functions like it has type object. At compile time, an element that is typed as dynamic is assumed to support any operation.

Though born as a statically typed language, C# becomes dynamically typed in any context in which you use the dynamic keyword, such as this:

dynamic number = 10;  
Console.WriteLine(number);

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
| Static Typing or Early Binding | |
|  | It is obvious that this won’t compile. Simply because, the compile/design time checking ensures type safety - and as we don’t have a method named SomeStupidCall in our Human class, we can’t compile the same. |

|  |  |
| --- | --- |
| Dynamic Typing or Late Binding | |
|  | And now, run the application. The application will break for sure. But hey, that is your fault, not the compiler’s. |
| Another Example  If I use dynamic keyword – during compile time – it will not show error but during runtime – it will complain – hence it will give strongly type during runtime. | |
|  |  |

Why use dynamic keyword

1. You may not know the type of the object you’re dealing with. You may have clues but not the certainty to statically type a given variable—which is just what happens in many common situations, such as when you work with COM objects, or when you use reflection to grab instances. In this context, the dynamic keyword makes some situations less painful to deal with. Code written with dynamic is easier to read and write, making for an application that’s easier to understand and maintain.

|  |  |
| --- | --- |
| When you are consuming MS Office components. |  |

1. Your object may have an inherently changing nature. You may be working with objects created in dynamic programming environments such as IronPython and IronRuby. But you can also use this functionality with HTML DOM objects (subject to expando properties) and the Microsoft .NET Framework 4 objects specifically created to have dynamic natures.

var versus dynamic

The keywords var and dynamic are only apparently similar. In the end, dynamic and var have quite opposite meanings.

*Difference 1*: Var is about reinforcing and improving static typing. It aims to ensure that the type of a variable is inferred by the compiler looking at the exact type being returned by the initializer. With var, your code is as statically typed as it would have been had you opted for the classic approach of using explicit types in a variable declaration.

The keyword dynamic is about avoiding static typing altogether. When used in a variable declaration, dynamic instructs the compiler to stop working out the type of the variable at all. The type has to be intended as the type it happens to have at run time.

*Difference 2*: var can only appear within a local variable declaration. You can’t use var to define a property on a class, nor can you use it to specify the return value or a parameter of a function.

Reflection versus dynamic keyword

|  |  |
| --- | --- |
|  | Dynamic keyword uses reflection internally |

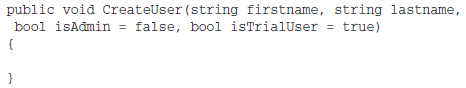
2. Named and Optional Paramters

Optional Parameters

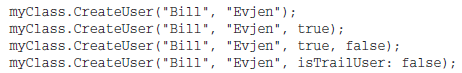
* They allow you to provide default values for some of the parameters of your methods and hence allow for type overloading by the consumer, even if there is only a single method in place to deal with all the variants.
* Example - Without Optional parameters:



With Optional Parameters:



* Parameters firstname and lastname don’t have default value set
* isAdmin andisTrialUser have a default value set
* As a consumer – you are now able to do something like this –



Named Parameters

* The last example above makes use of named parameters.
* It makes the code easier to write and understand because now you are able to use the parameter name in the code prior to the value being provided
* Example: File.Copy() method in the System.IO namespace

Typically we would write as this:



With named parameters, it is easy to understand what is being passed :



1. Improved COM Interoperability
2. Co- and Contra-Variance
3. Anonymous Methods & Anonymous Delegates (Inline delegates)

* 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

Links - <http://msdn.microsoft.com/en-us/library/0yw3tz5k(VS.80).aspx>

3. Covariance and Contravariance

What is a type: Fundamentally, a type in C# is a mathematical entity that obeys certain algebraic rules, just as natural numbers, complex numbers, quaternions, matrices, sets, sequences, and so on, are mathematical entities that obey certain algebraic rules.

The C# language has rules for associating types with compile-time expressions and storage locations, and rules for ensuring that the expression type is compatible with the storage type.

Type Safety: In computer science, type safety is the extent to which a programming language discourages or prevents type errors. A type error is erroneous or undesirable program behaviour caused by a discrepancy between differing data types.

* Other C# types support have always automatically supported covariance and contravariance.
* C# 4.0 will simply ensure that generic delegates and interfaces follow suit.

Within the type system of a programming language, covariance and contravariance refers to the ordering of types from narrower to wider and their interchangeability or equivalence in certain situations (such as parameters, generics, and return types).

* Covariant: converting from wider (double) to narrower (float).
* Contravariant: converting from narrower (float) to wider (double).
* Invariant: Not able to convert.

For example, a type that may assume the values {a,b,c,d} is wider than one that may only assume the values {a,b}. Hence, a type conversion from {a,b,c,d}->{a,b}, such as in the case of passing a double to a function expecting a float, is a covariant conversion.

Similarly, a type conversion from {a,b}->{a,b,c,d}, such as in the case of calling a function returning a float in the place of one returning a double, is a contravariant conversion of the function (the function type is its result type).

Detailed Explanation

In C#, roughly we can say that covariance means we can substitute derived type in place of base type. Contravariance means we can substitute base class in place of derived class (you are thinking it's not possible, right? We'll see how it's possible).

One of the new features is covariance and contravariance on type parameters that is now supported by generic delegates and generic interfaces.

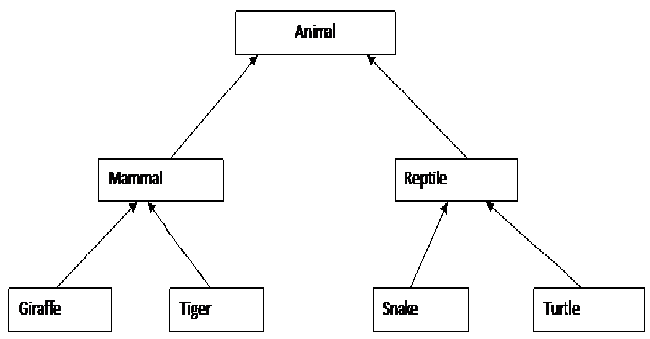
Restrictions

1. Covariance and contravariance issues are available only to delegates and interfaces
2. Generic type used for co- or contravariance should be a reference type.

Why this covariance only for reference types but not for value type? It's because for reference types, the array originally keeps only pointers to the original object and base pointer can refer to derived types. In case of value type, the original value is stored in array itself so the size may vary depending on the type. So covariance is not supported for array of values. For example, the following statement will not compile:

long[] arr = new int[100];

1. Type that is to be used for covariance can be used only as type for return values in corresponding interface of delegate. And type that is to be used for contravariance can be used only as type for input parameters in corresponding interface of delegate.



1. From C# 1.0, arrays where the element type is reference type are covariant. For example, the following statement in C# is OK.

Animal[] animals=new Mammal[10];

In the above code, mammal can be stored in animals array as mammal is derived from Animal. But remember, this is only true for reference types. So covariance is not supported for array of values.

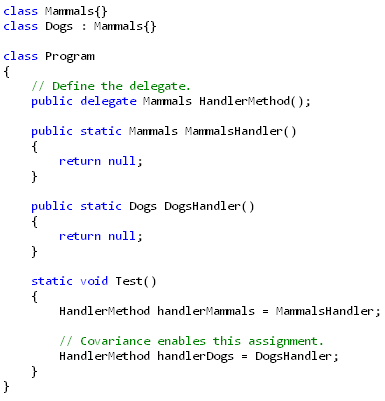
long[] arr = new int[100];

1. Covariance from method to delegates were included in C# 2.0. In the following code snippets (which is valid in C# 2.0 and later), you'll find that return type supports covariant. The original delegate has return type of Animal. But the method we have assigned (here, CopyMammal) to a variable (here, cfunc) has return type Mammal. So we can see that covariance is supported in return types.

**Some Background for Delegates:**

Steps for using Delegates

1. Defining the delegate
2. Creating methods which will be assigned to delegate object
3. Creating the delegate object and assigning methods to those delegate objects
4. Calling the methods via delegate objects



1. Contravariance is supported in parameters. Let's take a look at the following code snippets for understanding how contravariance works in parameters types:

//delegate which take one mammal argument and return nothing

delegate void CopyState(Mammal a);

void copyMammalState(Mammal mammal)

{

}

void copyAnimalState(Animal mammal)

{

}

void CopyGiraffeSate(Giraffe giraffe)

{

}

Now the following code will compile as contravariance is supported here. This is contravariance since we are using Animal parameter of CopyAnimalState in place of Mammal defined in CopyState delegate.

CopyState cs1 = copyAnimalState;

But the following code will not be supported as covariance is not supported in parameters.

CopyState cs2 = CopyGiraffeSate;

Videos –

C# 4.0 - <http://channel9.msdn.com/posts/matthijs/C-40-and-beyond-by-Anders-Hejlsberg/>  
C# 4.0 Code Samples - <http://code.msdn.microsoft.com/cs2010samples>

Links to pursue –   
<http://bogdanbrinzarea.wordpress.com/2009/04/24/learning-net-40-new-features/> **(Not helpful)**  
<http://msdn.microsoft.com/en-us/magazine/ff796223.aspx> **(Not helpful)**  
<http://blogs.msdn.com/b/csharpfaq/archive/2010/02/16/covariance-and-contravariance-faq.aspx> (Need to review)

4. Garbage Collection Enhancements

Background on Existing GC Process:

* Memory management in the CLR is broken into three generations made up of groups of memory segments.
* The first two are Gen 0 and Gen 1, called **ephemeral** generations since they are generally very short-lived.
* When an object is first created, it lives in Gen 0 until garbage collection runs.
* If by the time garbage collection runs the object is dead, that is, already deallocated and finalized, the memory that was allocated to that object is freed for use.
* If the object is still in use, it gets promoted to Gen 1.
* When garbage collection runs on a Gen 1 segment there are objects still in use, those objects are moved to Gen 2.
* Gen 2 segments are where the pain lives for garbage collection. Objects in Gen 2 tend to be long-lived and in applications where garbage collection becomes a problem—inevitably it is a large Gen 2 store that's the problem.
* Normally, garbage collection is a blocking process. This isn't a big deal for Gen 0 and Gen 1 segments because the garbage collection runs so quickly. But because Gen 2 can get so large, garbage collection takes longer, causing pauses in the UI of an application or forcing a web server to delay processing of web pages.

Uptill 4.0 Framework

Before CLR 4.0, garbage collection on Gen 2 worked in concurrent mode. This meant that memory could still be allocated in Gen 0 while a Gen 2 garbage collection was running. So in effect, execution of your application is not interrupted by a Gen 2 garbage collection, at least for a little while. Where things break down is when it's time for a Gen 0 or Gen 1 garbage collection and the Gen 2 garbage collection is still running—then everything stops. The ephemeral generations cannot run garbage collection while the Gen 2 garbage collection is running and the allocation will be delayed, adding latency to your application.

|  |  |  |
| --- | --- | --- |
|  | Concurrent Execution | Issue Scenario |
|  |  |

New in 4.0 Framework

In CLR 4.0, background garbage collection has been introduced, replacing concurrent garbage collection. Background garbage collection allows the ephemeral generations to run simultaneously with the Gen 2 garbage collection. Actually, it's not truly simultaneous; what has been added is that while a Gen 2 garbage collection is running, it checks to see whether the Gen 0 or Gen 1 collection needs to run and pauses its processing to let the much faster ephemeral generation garbage collection finish. This should keep your application working as long as your threads aren't dependent on memory objects in the Gen 2 segment.

Caveat

One caveat is this new background garbage collection is available only in the workstation mode of garbage collection. Server mode does not have this capability. Workstation and server mode garbage collection have nothing to do with the OS per se; they are modes that depend on multiple core machines. Server mode garbage collection will use many threads across many cores to do garbage collection very rapidly, although in the concurrent and blocking way.

The workstation mode of garbage collection is the default mode of the CLR unless you're running in the CLR in ASP.NET or within SQL Server—then you default to server mode. You can also set your application to run in server mode using the <gcServer> tag in the application configuration file. Also, a single CPU machine always runs the CLR in workstation mode.

5. DLR (Dynamic Language Runtime)

The dynamic language runtime (DLR) is a runtime environment that adds a set of services for dynamic languages to the common language runtime (CLR). The DLR makes it easier to develop dynamic languages to run on the .NET Framework and to add dynamic features to statically typed languages.

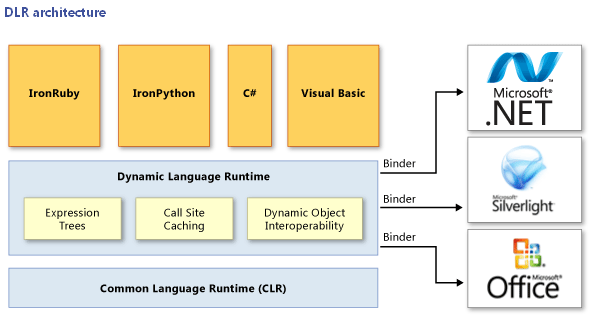
Dynamic languages can identify the type of an object at run time, whereas in statically typed languages such as C# and Visual Basic (when you use Option Explicit On) you must specify object types at design time. Examples of dynamic languages are Lisp, Smalltalk, JavaScript, PHP, Ruby, Python, ColdFusion, Lua, Cobra, and Groovy.

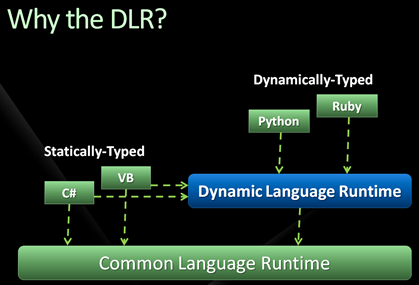
What are Dynamic Languages used for - Dynamic languages make excellent scripting languages. Customers can easily extend applications created by using dynamic languages with new commands and functionality. Dynamic languages are also frequently used for creating Web sites and test harnesses, maintaining server farms, developing various utilities, and performing data transformations.

Purpose of DLR: The purpose of the DLR is to enable a system of dynamic languages to run on the .NET Framework and give them .NET interoperability.

DLR Advantages:

1. Simplifies Porting Dynamic Languages to the .NET Framework
2. Enables Dynamic Features in Statically Typed Languages
3. Provides Future Benefits of the DLR and .NET Framework
4. Enables Sharing of Libraries and Objects





C# 5.0

<http://blogs.msdn.com/b/csharpfaq/archive/2012/02/29/visual-studio-11-beta-is-here.aspx>

Object Oriented Programming Basics

Three core pillars:

1. Encapsulation
2. Inheritance
3. Polymorphism

1. Encapsulation

It provides a capability to declare an accessibility level on members. This ensures that no outsiders can touch state in a manner that’s inconsistent with the consistency intended for (instances of) the type. This can be done by incorporating them in a class.

Even if the outside world is well intentioned with exposed writable state, other dangers are lurking. For example, private state may not be secure to be exposed. Also, exposing implementation details of a type hinders the type’s owner in evolving the way the type works. This is the main reason to never expose fields directly but to instead provide get/set accessors through properties.

2. Inheritance

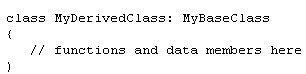
Inheritance provides 3 important benefits:

* Reuse
* Extend
* Modification

Two distinct types of inheritance — implementation inheritance and interface inheritance:

1. Implementation inheritance: Means that a type derives from a base type, taking all the base type's member fields and functions. The derived type adopts the base type's implementation of each function, unless it is indicated in the definition of the derived type that a function implementation is to be overridden.  
   Advantage: Most useful when you need to add functionality to an existing type, or when a number of related types share a significant amount of common functionality.

Syntax:



1. Interface inheritance: Means that a type inherits only the signatures of the functions and does not inherit any implementations.  
   Advantage: Most useful when you want to specify that a type makes certain features available

3. Polyorphism

Definition: The final pillar of object-oriented programming is known as polymorphism, from the Greek word that stands for **"Having many forms"** and **"Having multiple forms"**

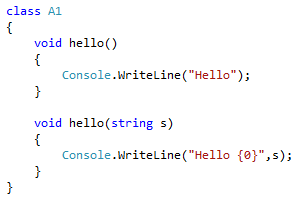
Polymorphism means same operation may behave differently on different classes.

Example of Compile Time Polymorphism: Method Overloading  
Example of Run Time Polymorphism: Method Overriding

Example of Compile Time Polymorphism : Method Overloading

- Method with same name but with different arguments is called method overloading.

- Method Overloading forms compile-time polymorphism.



Example of Run Time Polymorphism : Method overriding forms Run-time polymorphism.

C# gives us polymorphism through inheritance. Inheritance-based polymorphism allows us to define methods in a base class and override them with derived class implementations.

The virtual and override keywords are at the heart of the principle of polymorphism. virtual and override to be able to specialize a class.

Polymorphism Key points:

* Establishing type hierarchies by deriving classes using the : syntax
* Declaring virtual members using the virtual keyword
* Overriding base class members using the override keyword
* Blocking member overrides using the sealed keyword
* Dispatching to a base class member using the base keyword
* Hiding base class members using the new keyword
* Abstracting base classes using the abstract keyword

Virtual Methods

|  |  |
| --- | --- |
| By declaring a base class function as virtual, you allow the function to be overridden in any derived classes. | It is also permitted to declare a property as virtual. |
|  |  |

* You can override a virtual function in a derived class, and when the method is called, the appropriate method for the type of object is invoked
* In C#, functions are not virtual by default for performance reasons but (aside from constructors) can be explicitly declared as virtual.
* Neither member fields nor static functions can be declared as virtual. The concept simply wouldn't make sense for any class member other than an instance function member.

Hiding Methods

Use the new modifier to explicitly hide a member inherited from a base class. To hide an inherited member, declare it in the derived class using the same name, and modify it with the new modifier

|  |  |
| --- | --- |
|  |  |

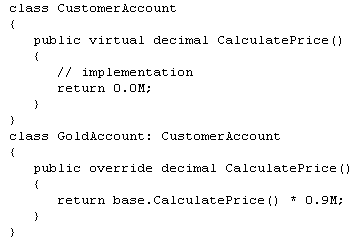
If a method with the same signature is declared in both base and derived classes, but the methods are not declared as virtual and override, respectively, then the derived class version is hidden. You will also get a compiler warning when this is done.

|  |  |
| --- | --- |
|  | **Compiler Error**    **Output**     * In most cases, you would want to override methods rather than hide them. By hiding them you risk calling the wrong method for a given class instance. * C# syntax is designed to ensure that the developer is warned at compile time about this potential problem, thus making it safer to hide methods if that is your intention. |

Read this article for sure - <http://stackoverflow.com/questions/8229171/benefit-of-using-new-keyword-in-derived-class-member-having-same-name-with-base>

Calling Base Version of the Functions

* C# has a special syntax for calling base versions of a method from a derived class: base.<MethodName>(). For example, if you want a method in a derived class to return 90 percent of the value returned by the base class method, you can use the following syntax:



* Note that you can use the base.<MethodName>() syntax to call any method in the base class — you don't have to call it from inside an override of the same method.

Important Points with regards to Example Practiced

1. 01 – Polymorphism

|  |  |
| --- | --- |
| * See how both base class and derived class constructors are called whenever derived class instance is involved. |  |
| * Also the base class constructer is always called first |
| * In 02 - Polymorphism\_With\_Interface: it’s an example of Polymorphism with Interface hence no constructer can be there in interface |
| * In 03 – Polymorphism: Only the default constructer of the base class is always called |  |

-------------------------------

1. Is polymorphism possible with Interfaces?  
   Answer: **Yes**. See 02 - Polymorphism\_With\_Interface   
   As you see from the example – when polymorphism is used with interfaces – there is no virtual/override keywords used.

-------------------------------

1. 04 – Polymorphism - When we mark a method as virtual in the base class, it means that the derived class which inherits this method can override the method and provide its own implementation as shown in our example. The derived class can use the inherited virtual method without overriding.

|  |  |
| --- | --- |
| **Base class** | **Derived Class** |
|  |  |
| virtual | new override |

-------------------------------

1. 06 – Polymorphism/07 – Polymorphism : Important -  
   When doing polymorphism – meaning:



AND

If a method with the same signature is declared in both base and derived classes, but the methods are not declared as virtual and override, respectively, the base class method is (always) called. You will also get a compiler warning when this is done (unless new keyword is used in the Derived class)

|  |
| --- |
|  |
|  |
|  |
| Output |

1. 08 – Polymorphism -

Rule of Thumb: In polymorphism – whenever override is present, it’s always calls method of instance



|  |  |  |
| --- | --- | --- |
| **Stack** |  | **Heap** |
| Base Class b (In my code Animal Class – a2) |  | Derived Class (In my code Dog Class) |
| (Reference) |  | (Instance) |

|  |  |
| --- | --- |
| **Base Class** | **Derived Class** |
|  |  |
|  |  |

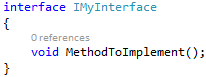
Whether it’s a virtual or not in the base class – if the new word is used in the derived class – the base class method is (always) called.

|  |  |
| --- | --- |
| **Base Class** | **Derived Class** |
|  |  |
| a2.Talk();  a2.Dance(); |  |

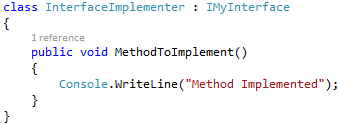
Interface

* Interfaces provide a contract between two parties: an implementer and a consumer.
* An interface looks like a class, but has no implementation. The only thing it contains are declarations of events, indexers, methods and/or properties.
* The reason interfaces only provide declarations is because they are inherited by classes and structs, which must provide an implementation for each interface member declared.
* So, what are interfaces good for if they don't implement functionality? They're great for putting together plug-n-play like architectures where components can be interchanged at will.
* Cant use a protection keyword in interface (such as public) because there is no implementation and hence the default is automatically public (vs a class where it is private)
* You will receive a compiler error if you don't implement all interface members.

Definition of interface



Class Implementing the Interface



Example 2

For instance, if class foo implements the IDisposable interface, it is making a statement that it guarantees it has the Dispose() method, which is the only member of the IDisposable interface. Any code that wishes to use class foo may check to see if class foo implements IDisposable. When the answer is true, then the code knows that it can call foo.Dispose().

Multiple Inheritance using Interface

Interfaces may also inherit other interfaces.

Some languages such as C++ support what is known as multiple inheritance, in which a class derives from more than one other class.

You can only inherit from a single class. It is however possible to implement multiple interfaces.

In essence

* C# does not support multiple implementation inheritance.
* It does allow types to be derived from multiple interfaces — multiple interface inheritance. This means that a C# class can be derived from one other class, and any number of interfaces.

|  |  |
| --- | --- |
| Example 1 | Example 2 |
|  |  |

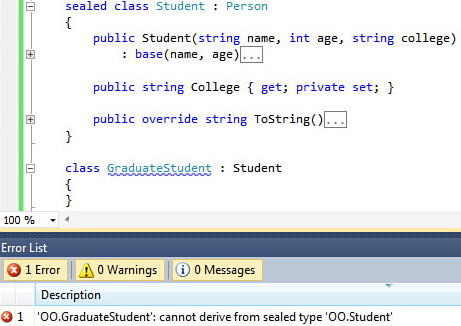
The code contains two interfaces: IMyInterface and the interface it inherits, IParentInterface. When one interface inherits another, any implementing class or struct must implement every interface member in the entire inheritance chain. Therefore, the InterfaceImplementer class must implement the MethodToImplement() method specified in the IMyInterface interface and the ParentInterfaceMethod() method specified in the IParentInterface interface.

Blocking Inheritance

C# allows classes and methods to be declared as sealed. In the case of a class, this means that you can't inherit from that class. In the case of a method, this means that you can't override that method.

The most likely situation in which you'll mark a class or method as sealed is if the class or method is internal to the operation of the library, class, or other classes that you are writing, so that you ensure that any attempt to override some of its functionality will lead to instability in the code. You might also mark a class or method as sealed for commercial reasons, in order to prevent a third party from extending your classes in a manner that is contrary to the licensing agreements.

The .NET base class library frequently uses sealed classes to make these classes inaccessible to third-party developers who might want to derive their own classes from them. For example, string is a sealed class.



What is Explicit Interface Implementation

When you have a class which implements 2 interfaces which have the exact same method, you need to do explicit interface implementation so as to make it clear the method being called (belongs to which interface). See example below:

|  |  |
| --- | --- |
| 2 steps necessary to achieve this:   1. When you implement the interface methods in the class, you have to explicitly reference the interface (See red dotted arrow) 2. When calling the method, you have to cast them with the interface too (see blue arrow)   Once you do explicit interface, the only way to invoke them is through type-casting. So you cannot do P.InterfaceMethod()   1. Access Modifiers are not allowed on explicitly implemented interface members |  |

If you want to have a default implementation and a explicit implementation, that is possible too. See code sample below.

|  |  |
| --- | --- |
| Couple of points here:   * Now since InterfaceMethod is not decorated by I1.Interfacemethod - it becomes default. * Now you can call P.InterfaceMethod() and it will call the default implementation * The method can now have access modifier (public) since Explicit interface implementation is not used for default method |  |

What is the difference between Inherits and Implements

Inheritance implies extending existing behavior. An interface has no behavior, you have to implement all the behavior of the methods/properties defined in the interface. Hence you "implement" an interface and cannot really inherit anything from it. If you inherit from something else you inherit all it's concrete behavior.

Inheritance is transitive. If ClassC is derived from ClassB, and ClassB is derived from ClassA, ClassC inherits the members declared in ClassB and ClassA.

Abstract Classes

* Abstract classes are one of the essential behaviors provided by .NET. Commonly, you would like to make classes that only represent base classes, and don’t want anyone to create objects of these class types.
* Abstract methods have no implementations. The implementation logic is provided rather by classes that derive from them. We use an abstract class to create a base template for derived classes.
* An abstract class means that, no object of this class can be instantiated, but can make derivations of this.
* An abstract class can contain either abstract methods or non abstract methods. Abstract members do not have any implementation in the abstract class, but the same has to be provided in its derived class.

Limitations of Abstract Class and Abstract method

* Declaration of abstract methods are only allowed in abstract classes.
* The access modifier of the abstract method should be same in both the abstract class and its derived class. If you declare an abstract method as protected, it should be protected in its derived class. Otherwise, the compiler will raise an error.
* An abstract method cannot have the modifier virtual. Because an abstract method is implicitly virtual.
* An abstract member cannot be static.
* An abstract method cannot be private.
* Abstract class cannot be a sealed class.

When should a class have to be absolutely declared as abstract

1. When the class itself is inherited from an abstract class, but not all base abstract methods have been overridden.
2. When at least one of the methods in the class is abstract.

Practical Usage of abstract Class

* Abstract classes, and interfaces (to a lesser degree), are both what we consider a contract.
* Its easier to modify the contract when they are designed as abstract classes versus interface

Difference between Interface/Abstract Classes/Virtual Methods

One would use an interface when one wants to enforce a contract, the methods of which must be implemented by all derived classes.

One would use virtual when one wants to define default implementation details while allowing for the option of derived classes overriding and defining their own implementations.

Last, one would use abstract when one wants to provide default implementation details while also forcing derived classes to provide their own implementation of those methods defined as abstract. A rule of thumb is that whenever class implementations are incomplete, mark them as abstract.

|  |  |
| --- | --- |
| **Abstract Classes** | **Virtual** |
| An abstract function can have no functionality. You're basically saying, any child class MUST give their own version of this method | A virtual function, is basically saying look, here's the functionality that may or may not be good enough for the child class. So if it is good enough, use this method, if not, then override me, and provide your own functionality. |

|  |  |
| --- | --- |
| **Abstract Classes** | **Interface** |
| Abstract classes are used when we want to share common functionality in parent child relationship. | Interfaces are used to define contract, enforce standardization, decoupling and dynamic polymorphism. |
| Abstract classes are inherited. | Interfaces are implemented. |
| Derived classes exhaust their single base class inheritance option. | Classes can implement multiple interfaces without using up their base class option. But, there are no default implementations. |
| New non-abstract members may be added that derived classes will inherit without breaking version compatibility. | Extending an interface with new members breaks version compatibility. |
| Optionally, provide default (virtual) member implementation. | All members are virtual and cannot provide implementations. |
| Cannot be instantiated except as part of subclasses. Only derived classes can call an abstract class constructor. | Cannot be instantiated |
| Can include data fields. | Cannot include data fields. However, abstract properties may be declared. |

Rule of Thumb for interfaces: When implementing interfaces, consider these rules of thumb:

* When functionality dictated by an interface needs to be implemented over and over again, use an abstract base class.
* Do not implement interfaces on abstract base classes, but rather in the derived classes.
* When creating an inheritance hierarchy, don’t be afraid of implementing the identical interface at different places in the inheritance hierarchy. So long as you instantiate the appropriate object in the inheritance hierarchy, the appropriate methods will be called.
* Use the virtual, abstract, and override keywords to define default implementations that some derived class can or should override.

Access Modifiers

**Public**: This class is accessible to all other classes. If a class is declared without explicitly specifying an access modifier for it, then it's public by default.

**private**: Accessible only by the class in which it is declared.

**sealed:** Prevents a class from every being derived. If another class tries to use this class as its base class either directly or indirectly then the C# compiler will raise an error.

**protected**: Accessible only by the class in which it is declared, as well as any derived classes (derivation can be in the same assembly or different doesn’t matter – It will be accessible as long it is derived)

**internal:** Accessible only from within the same assembly (in C#, an assembly is a package of inter-related data that contains both code and meta data).

**protected internal:**

* Access is limited to types derived from the containing class **or** the current assembly.
* protected internal is the only access modifiers combination allowed for a member or a type.

**---------------------------**

**abstract:** Similar to the concept of a pure virtual function in C++, an abstract class can't actually be instantiated. It contains a signature, but can only be used when it is the base class of a derived class.

**new:** Using the new keyword as an access modifier for a nested class allows us to hide an inherited method of a parent class by providing the compiler with a new version of that class

**Static:** The static modifier indicates that the method does not operate on a specific instance of your class and therefore is called without first instantiating the class.

|  |  |
| --- | --- |
| Protected Example | |
|  | Protected access modifier lets you do the above and no error is thrown |

**Will “Internal Protected” broaden or narrow the scope compared to “Protected”?**

* Broaden the scope – and that is due to the internal part of the access modifier. So within the same assembly somebody else can access it (without inheriting)

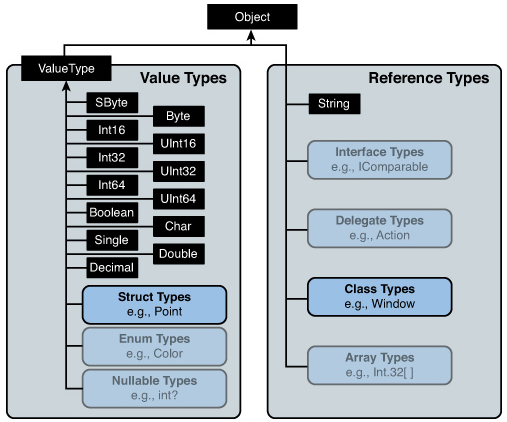
|  |  |
| --- | --- |
| C:\Users\Pratik\AppData\Local\Temp\SNAGHTML18ae533.PNG |  |

So, if we will try to compile AssemblyB we’ll get an error in line 23 that says : AssemblyA.A.SomeProtectedInternalMethod()’ is inaccessible due to its protection level.

Memory Management

Value Types versus Reference Types

|  |  |  |
| --- | --- | --- |
| **Sr.** | **Value Types** | **Reference Types** |
| 1 | Value types derive from System.ValueType | Reference types derive from System.Object |
| 2 | Value Types are stored on the stack | Instances of reference types are always stored in area of memory which is known as managed heap. |
| 3 | Predefined Value Type available: int, float, double, decimal, bool, char, structs | Predefined Reference Type available: object, string |
| 4 | When a value-type instance is created, a single space in memory is allocated to store the value. | With reference types, however, an object is created in memory, and then handled through a separate reference—rather like a pointer. |
| 5 | Highly efficient since they are on stack which uses a simple first-in last-out memory structure | Advantage: Heap is a random jumble of objects which allows objects to be allocated or deallocated in a random order. It does require though an overhead of a memory manager and a Garbage Collector to keep things in order |
| 6 | A value type may not contain NULL values (except nullable Value Types) | A reference type can contain NULL values |
| Important Note: Although if value types are declared as fields within reference types, they will be stored   inline on the heap | | |



|  |  |
| --- | --- |
| Predefined Value Types | Predefined Reference Types |
| Integer Types: sbyte, short, int, long, byte, ushort, uint, ulong  Floating Point Types: float, double  Decimal Type: decimal  Boolean Type: bool  Character Type: char | Object:   * In C#, the object type is the ultimate parent type from which all other intrinsic and user-defined types are derived * The object type implements a number of basic, general-purpose methods, which include Equals(), GetHashCode(), GetType(), and ToString().   String:   * Despite this style of assignment, string is a reference type. Behind the scenes, a string object is allocated on the heap, not the stack, and when you assign one string variable to another string, you get two references to the same string in memory. |

Practical Explanation

If you had a class wherein the following happened:

Emp a = new employee(); // Employee had an id associated with it int Empid and a method say GetEmp Details.

Public void GetEmp()  
{  
 Forach (int a in Empid)  
 {……}  
}

So what is stored where:

* Object a is stored on heap since its instantiating a class (reference type)
* Empid since its related to employee is stored on online heap (hence reference type and not value type)
* Int a is stored in stack since its not associated to the reference type Employee (so even though its within the class – its still stored on stack)

Detailed Explanation

Somewhere inside a processor's 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. To understand how the stack works, you 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. Look at this code:  
{

int a;

// do something

{

int b;

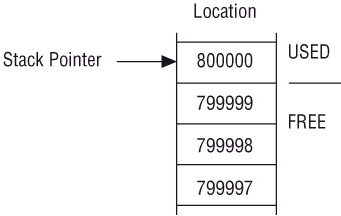
// do something else

}

}

First, a gets declared. Then, inside the inner code block, b gets declared. Then the inner code block terminates and b goes out of scope, then a goes out of scope. So, the lifetime of b is entirely contained within the lifetime of a.

The idea that you always deallocate variables in the reverse order to how you allocate them is crucial to the way the stack works.



If the compiler hits a line like int i, j, then the order of variables coming into scope looks indeterminate. Both variables are declared at the same time and go out of scope at the same time. In this situation, it does not matter in what order the two variables are removed from memory. The compiler internally always ensures that the one that was put in memory first is removed last, thus preserving the rule about no crossover of variable lifetimes.

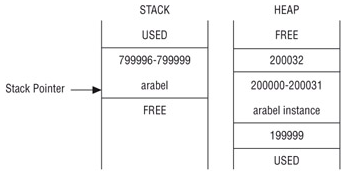
Reference Data Types

Although 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 is where the managed heap comes in.

void DoWork()

{   
 Customer c;  
 c = new Customer();  
 Customer otherCustomer2 = new EnhancedCustomer();  
}

* First, you declare a Customer reference called c. The space for this will be allocated on the stack, but remember that this is only a reference, not an actual Customer object.
* The next line, c= new Customer(); 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 c to the address of the memory it has allocated to the new Customer object.



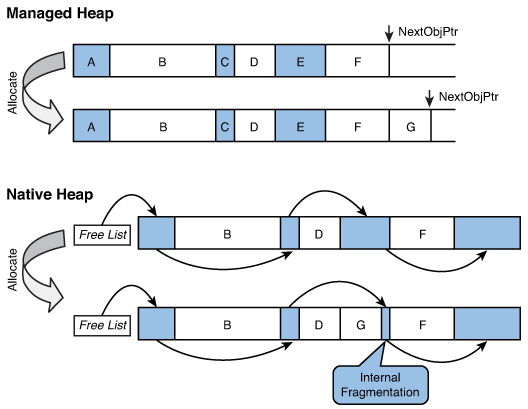
It is clear from the example that the process of setting up a reference variable is more complex than that for setting up a value variable, and there is a performance overhead. Despite this overhead, you now have a mechanism for allocating variables that is not constrained by the limitations of the stack. By assigning the value of one reference variable to another of the same type, you have two variables that reference the same object in memory. When a reference variable goes out of scope, it is removed from the stack as described in the previous section, but the data for a referenced object is still sitting on the heap. The data will remain on the heap until either the program terminates or the **garbage collector** removes it, which will happen only when it is no longer referenced by any variables.

That is the power of reference data types, and you will see this feature used extensively in C# code. It means that you have a high degree of control over the lifetime of your data, because it is guaranteed to exist in the heap as long as you are maintaining some reference to it.

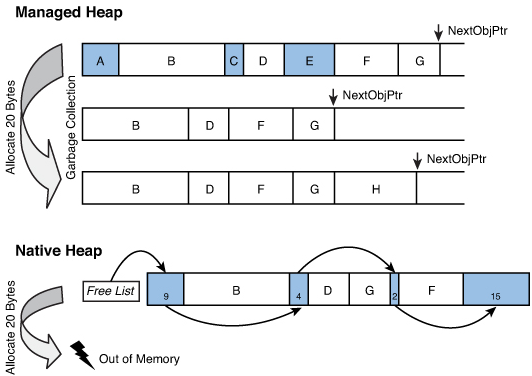
Differences between Structs and Classes

|  |  |  |
| --- | --- | --- |
| **Sr.** | **Struct** | **Class** |
| 1 | When a struct is instantiated – it gets created on a stack since its value type | When a class is instantiated – it gets created on heap since its reference type |
| 2 | Does not support inheritance (inheritance polymorphism).  Since does not support inheritance – access modifier cannot be protected or protected internal | Supports Inheritance.  All access modifiers are allowed |
| 3 | All Fields of a struct must be fully initialized inside the constructor      *myString must be fully assigned before it leaves the constructor* | No mandatory to initialize all fields inside the constructor of a call      *No matter whether the Field myString is initialized or not* |
| 4 | Structs cannot have explicit parameter-less constructors | Classes can have explicit parameter-less constructors |
| 5 | Structs can be instantiated without using the new operator | Classes must be instantiated with the new operator  MyClass myc  Myc.myVar = 100 //NullReferenceException  Correct Option 🡪 |
| 6 | Struct is not permitted to declare a destructor | Class is permitted to declare a destructor |
| 7 | Since struct is value type- they cannot be assigned to NULL | Since classes are reference type – they can be assigned to NULL |

What is Managed Heap

****

* Garbage Collector takes care of freeing up unused memory hence providing a way for automated memory management
* Easy to allocate memory on the managed heap – conceptually similar to stack based collection. See where’s there is enough space left, allocate and move the high-water mark.
* Managed heap doesn’t suffer from internal fragmentation since its managed by Garbage collection
* In Managed Heap, memory is reclaimed automatically when heap runs out of space by Garbage Collector



For bullet 2 and 3

Delegates

Delegate is a type which safely encapsulates a method, similar to function pointer in C and C++. Unlike C function pointers, delegates are object-oriented, type-safe and secure. Essentially a delegate contains the address of a method.

Using a delegate allows the programmer to encapsulate a reference to a method inside a delegate object. The delegate object can then be passed to code which can call the referenced method, without having to know at compile time which method will be invoked.

.NET delegate is **type-safe** because when you define the delegate, you have to give full details of the signature and the return type of the method it is going to represent. What this means is that a function pointer in C is nothing but a pointer to a memory location. You have no idea what that pointer is really pointing to. Things like parameters and return types are not known.

Abstraction in C# can be achieved by the following:

Core 3 mechanisms:

* Interface: Kind of contract between a class and its consumers. Pure form of abstraction which can help loosen up the coupling between subsystems
* Inheritance: More tightly coupled than interface. It allows to share contracts and implementation between classes and can help code reuse when used judiciously.
* Delegates and Events: Miniature form of interface – where only a single method is abstracted instead of set of methods and event is simply a clean way to use a delegate as a member of a class. Allows consumer of your class to register for notifications and also send those notifications when something happens.

When you want to use a class in C#, you do so in two stages.

* First, you need to define the class — that is, you need to tell the compiler what fields and methods make up the class.
* Then (unless you are using only static methods), you instantiate an object of that class.

With delegates it is the same process.

* You have to start by defining the delegates you want to use. Defining delegates means telling the compiler what kind of method a delegate of that type will represent.
* Then, you have to create one or more instances of that delegate. Behind the scenes, the compiler creates a class that represents the delegate. After you have defined a delegate, you can create an instance of it so that you can use it to store details of a particular method.

**Basic Delegate**The signature of a single cast delegate is shown below:

<public> delegate return-type identifier ([parameters]);

where:

return-type: The result type, which matches the return type of the function.   
identifier: The delegate name.   
parameters: The Parameters, that the function takes.

An interesting and useful property of a delegate is that it does not know or care about the class of the object that it references. Any object will do; all that matters is that the method's argument types and return type match the delegate's. This makes delegates perfectly suited for "anonymous" invocation.

|  |
| --- |
| public delegate void SimpleDelegate ()  This declaration defines a delegate named SimpleDelegate, which will encapsulate any method that takes  no parameters and returns no value. |
| public delegate int ButtonClickHandler (object obj1, object obj2)  This declaration defines a delegate named ButtonClickHandler, which will encapsulate any method that takes  two objects as parameters and returns an int |

**A delegate will allow us to specify what the function we'll be calling looks like without having to specify which function to call.** The declaration for a delegate looks just like the declaration for a function, except that in this case, we're declaring the signature of functions that this delegate can reference.

There are **four** steps in defining and using delegates:

|  |  |
| --- | --- |
| **Delegate (accepting string) returning void** | **Delegate (accepting string) returning string** |
| **1) Declaration/Define**    **2) Define the method which will be encapsulated**    **3) Instantiation**    **4) Invocation** | **1) Declaration/Define**    **2) Define the method which will be encapsulated**    **3) Instantiation**    **4) Invocation** |
|  | **Step 3 & 4 – Combined is generally the way to write** |

There is an unfortunate problem with terminology here. With classes there are two distinct terms — class , which indicates the broader definition, and object , which means an instance of the class. Unfortunately, with delegates there is only the one term; delegate can refer to both the class and the object. When you create an instance of a delegate, what you have created is also referred to as a delegate. You need to be aware of the context to know which meaning we are using when we talk about delegates.

Action<T> / Function<T> / Predicate Delegates

They provide a mechanism to declare generic delegates.

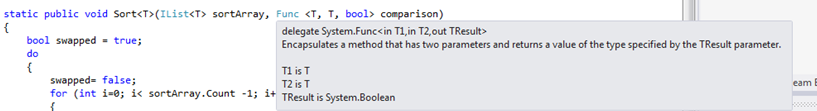
**Action<T>:**

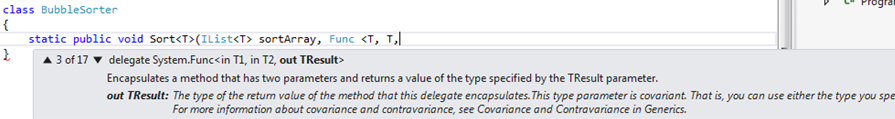
* Encapsulates a method that has no parameters and does not return a value. Essentially - The generic Action<T> delegate is meant to reference a method with void return.
* You can use this delegate to pass a method as a parameter without explicitly declaring a custom delegate.
* You can pass upto 16 different parameters
* Signature: 
* When you use the Action delegate, you do not have to explicitly define a delegate that encapsulates a parameterless procedure.

|  |  |
| --- | --- |
| Using Standard Delegate Syntax | Using Action Delegate |
|  |  |

**Func<T>:**

* Encapsulates a method that has no parameters and returns a value of the type specified by the TResult parameter.
* Using the generic Func<TResult> delegate essentially allows you to invoke methods with a return type.
* You can pass upto 16 different parameters
* Signature: 
* out TResult
* The type of the return value of the method that this delegate encapsulates.
* This type parameter is covariant. That is, you can use either the type you specified or any type that is more derived.





|  |  |
| --- | --- |
| Using Standard Delegate Syntax | Using Func Delegate |
|  |  |

**Predicate<T> Delegate:**

Represents the method that defines a set of criteria and determines whether the specified object meets those criteria. Predicate is very much different from the other two. It doesn’t have multiple overloads like the others and it can only return true/false corresponding to each item in the collection.

So, a predicate delegate is a delegate which points to a boolean function that returns true or false and takes a generic type as an argument. A predicate delegate thus is a delegate which is capable of taking any custom type as an argument. This makes it quite useful because what we get as a result is a generic delegate. The bool function that it points to has logic to evaluate any condition and return true/false accordingly.

The most common use of a predicate delegate is for searching items in a collection

**Absolutely read the following articles for better understanding of Action and Func Delegates**

Article 1 – <http://blogs.msdn.com/b/brunoterkaly/archive/2012/03/02/c-delegates-actions-lambdas-keeping-it-super-simple.aspx>

Article 2 - <http://www.blackwasp.co.uk/FuncAction.aspx>

Differences Between Action, Func & Predicate Delegates

One important thing that distinguishes each one of them with the others is the returned value. They return either a collection , scalar value(single item) or nothing. Following listing describes what could be the possible outcomes of these delegates.

Action<> : It doesn’t return anything.

Func<> : It has the capability to return value. Returned value can be of any type, irrespective of the type of input source. This makes it the most used generic delegate. When working with Func<> delegate you can get the following possible outcomes :

* Same type and same length as of the input source e.g. OrderBy
* Same length but different type e.g. Select (fordCars.Select(item => item.Length))
* Single value same type e.g. First
* Single value different type e.g. Average
* Same type but shortened in length e.g. TakeWhile , SkipWhile

Predicate<> : Predicate subjects each item in collection to a test and returns true or false.

**MultiCast Delegate**

It is a delegate which holds the reference of more than one method.

Best Practice: Multicast delegates must contain only methods that return void otherwise we would get the result of the last method that is invoked by the delegate.

But if you want to do multicast delegates and still have return value – then you need to use GenInvocationList() to get handle on result of each result (or for that matter if one of the methods throws an error – see below)

See example - G:\04 - Net Technical Material\!Visual Studio 2010 Projects\01-C\_Sharp\_Concepts\aMine\02 - Delegates\_Examples\10-Multicast\_Delegates\_GetInv\_Example

**Exception in Multicast Delegates**

When an exception occurs during the invocation of one of the methods a delegate refers to, the delegate does not continue invoking methods in its invocation list. The result might also differ because the order of calling the methods is not defined.

In such a scenario, this can be avoided by iterating the list on our own. The Delegate class defines the method GetInvocationList() (invoke the methods directly, rather than letting the delegate invoke them) that returns an array of Delegate objects. We can now use this delegate to invoke the methods associated with them directly, catch exceptions and continue with the next iteration.

GetInvocationList() returns an array of System.Delegate types, each representing a particular method that may be invoked.

**Write notes on**Comparison Delegate  
delegate inherited from multicast delegate  
delegate calling 2 methods with return type - in multicasting it will always give the second value only unless you use getinvocationlist

Anonymous Methods

Up to this point, a method must already exist for the delegate to work (that is, the delegate is defined with the same signature as the method(s) it will be used with). However, there is another way to use delegates — with anonymous methods.

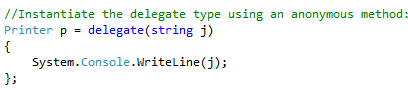
An anonymous method is a block of code that is used as the parameter for the delegate.

The syntax for defining a delegate with an anonymous method doesn’t change. It’s when the delegate is instantiated that things change.

Step 1 (Declaring the Delegate) (Doesn't change) – if using traditional delegate syntax



Step 2 (Defining the method which will be encapsulation and instantiation will be done in a single step)



If you are using Action or Func delegates - all the 3 steps - defining a delegate, defining the method and instantiation are all done in one step – 1,2 and 3

|  |  |
| --- | --- |
| Action Delegate and Anonymous method | Func delegate and Anonymous method |
|  |  |
| Example 2 - Explanation - The delegate Func<string, string> takes a single string parameter and returns a string. anonDel is a variable of this delegate type. Instead of assigning the name of a method to this variable, a simple block of code is used, prefixed by the delegate keyword, followed by a string parameter. |  |

The benefit of using anonymous methods is to reduce the amount of code you have to write. You don’t have to define a method just to use it with a delegate. This becomes very evident when defining the delegate for an event. This can help reduce the complexity of code, especially where there are several events defined. With anonymous methods, the code does not perform faster. The compiler still defines a method; the method just has an automatically assigned name that you don’t need to know.

Beginning with C# 3.0, you can use Lambda expressions instead of writing anonymous methods.

Lambda Expressions

Since C# 3.0, you can use a new syntax for assigning code implementation to delegates: Lambda expressions. Lambda expressions can be used whenever you have a delegate parameter type.

The left side of the Lambda operator => lists the parameters needed. The right side following the Lambda operator defines the implementation of the method that is assigned to the variable lambda.

Here’s a simple C# example of a lambda expressions: Y => y \* 2

This reads as “y goes to y times 2” the body of a lambda expression can consist of any number of statements, but typically you want to keep it to 2 or 3, mainly for readability.

|  |  |
| --- | --- |
| Action Delegate and Lamda Expression | Func delegate and Lamda Expression |
|  |  |
| Example 2 - Explanation – Converted the above anonymous example to using lamda expression |  |

Parameters

With Lambda expressions there are several ways to define parameters.

Example 1 - If there’s only one parameter, just the name of the parameter is enough. The following Lambda expression uses the parameter named s. Because the delegate type defines a string parameter, s is of type string.



Example 2 -



--------------------------------------

If a delegate uses more than one parameter, you can combine the parameter names inside brackets. Here the parameters x and y are of type double as defined by the Func<double, double, double> delegate:



--------------------------------------

For convenience, you can add the parameter types to the variable names inside the brackets:



--------------------------------------

**Variables outside of Lambda Expression**

With Lambda expressions you can access variables outside the block of the Lambda expression. This is a great feature but can also be very dangerous if not used correctly.

In the example here, a Lambda expression of type Func < int, int > requires one int parameter and returns an int . The parameter for the Lambda expression is defined with the variable x . The implementation also accesses the variable someVal that is outside the Lambda expression. As long as you do not think that the Lambda expression creates a new method that is used later when f is invoked, this might not look confusing at all. Looking at this code block the returned value calling f should be the value from x plus 5, but this might not be the case:

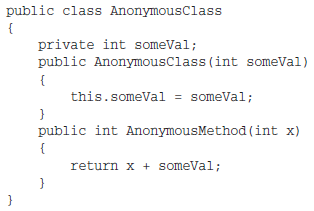


Assuming the variable someVal is later changed, and then the Lambda expression invoked, the new value of someVal is used. The result here invoking f(3) is 10 :



In particular, when the Lambda expression is invoked by a separate thread you might not know when the invocation happened and thus what value the outside variable currently has.

Now you might wonder how it is possible at all to access variables outside of the Lambda expression from within the Lambda expression. To understand this, look at what the compiler does when you defi ne a Lambda expression. With the Lambda expression x = > x + someVal , the compiler creates an anonymous class that has a constructor to pass the outer variable.



Best Article on Delgates/Anonymous Methods/Lamda evolution - <http://www.codeproject.com/Articles/47887/C-Delegates-Anonymous-Methods-and-Lambda-Expressio>

**Events**

Talking Points

1. What are events

* Why do they exist
* Syntax (Thread safety perspective)
* Declaring an event (Utilizing the standard EventHandler versus MyCustomEventHandler)
* Invoking an event
* Hooking up an event
* EventArgs and Custom EventArgs
* Why are events always of void return delegate type

2. Difference between Delegates and Events

3. Memory Leak through Events

4. Publisher/Subscriber Code WalkThrough (Multiple examples)

5. Publisher/Subscriber through WCF Code WalkThrough (Amit Example)

6. Weak Events

--------------------------------------

Events

Definition

An event in C# is a way for a class to provide notifications to clients of that class when some interesting thing happens to an object. The most familiar use for events is in graphical user interfaces; typically, the classes that represent controls in the interface have events that are notified when the user does something to the control (for example, click a button).

Events are declared using delegates.

|  |  |
| --- | --- |
| Delegates really have two sides to them. One side defines one or more target methods referred to by the (multicast) delegate; the other defines a way to invoke those methods through the delegate. To keep things clear, let’s define a little bit of vocabulary here. We’ll say a target method gets called, whereas a delegate is getting invoked.  When we look at the picture from left to right, delegates look like very interactive beasts, where the invocation site is in the driver seat: It actively invokes the delegate, triggering the target methods to be called. | Now let’s flip the figure around, keeping shapes and arrows in place but performing a few rename operations. For the verbs, to invoke becomes to raise and to call becomes to handle. For the nouns, replace target method with event handler and invocation site with event source.  Again, scanning the picture from left to right, things start to look quite different. Instead of an outgoing Invoke arrow that interacts with the delegate, we now see a bunch of incoming Handle arrows that allow us to react to an incoming event. If we follow those arrows against the flow through the delegate, we end up at the event source on the right, which raised an event. |
|  |  |

Event Syntax Components

1) Event Declaration Approaches  
The event keyword is used to formally declare an event. There are two valid event declaration syntax alternatives. Regardless of the syntax you write, the C# compiler will translate both property declarations into the following three components in the output assembly.

* Privately scoped event handler (or a functionally equivalent data structure). The delegate is privately scoped in order to prevent external code from invoking the event, and thereby preserving encapsulation.
* publicly scoped Add method; used to add subscribers to the private event handler.
* publicly scoped Remove method used to remove subscribers from the private event handler.

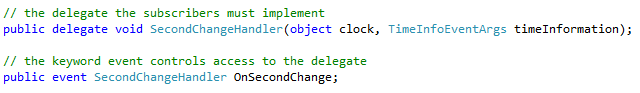
|  |  |
| --- | --- |
| Declaration | Threading Considerations |
| **1. Field Like Syntax** | |
| Using .Net provided Event Handler    OR  Custom Event Handler | The field-like syntax is automatically thread safe: |
| **2. Property like Syntax** | |
| The property-like syntax appears very similar to a typical property declaration, but with explicit add and remove blocks in place of "getter" and "setter" blocks. Instead of retrieving or setting the value of a private member variable, they add and remove incoming delegate instances to/from the underlying event handler or other data structure that servers a similar role.  Also once we implement add and remove accessors, the compiler no longer generates a field for the multicast delegate, leaving all plumbing to the developer. | The property-like syntax will be as thread safe as you make it. The above is a thread safe version |

Under the covers - Using EventHandlers versus CustomEventHandlers

Read this article from MSDN- <http://msdn.microsoft.com/en-us/library/system.eventhandler.aspx>

Event Class Definition





If I use standard .Net provided EventHandler - the above would convert to:



Read this article for sure - <http://stackoverflow.com/questions/803242/understanding-events-and-event-handlers-in-c-sharp/803320#803320>

2) Invoking a Event  
Once a class has declared an event, it can treat that event just like a field of the indicated delegate type. The field will either be null, if no client has hooked up a delegate to the event, or else it refers to a delegate that should be called when the event is invoked. Thus, invoking an event is generally done by first checking for null and then calling the event.

**Invoking an event can only be done from within the class that declared the event.**

|  |  |
| --- | --- |
| Invoking a Delegate | Associated Code |
|  |  |

3) Hooking up an event

From outside the class that declared it, an event looks like a field, but access to that field is very restricted. The only things that can be done are:

* Compose a new delegate onto that field.
* Remove a delegate from a (possibly composite) field.

This is done with the += and -= operators.

Leveraging standard EventHandlers:

a) If the wiring is done on the main function (and not on the subscriber side)



b) If the wiring is done on the subscriber side (and not on the main function)



When using Custom EventHandlers (such as MyEventHandler), then:

|  |
| --- |
| **Example 1** |
|  |
| **Example 2** |
|  |

EventArgs and CustomEventArgs

EventArgs - Represents the base class for classes that contain event data, and provides a value to use for events that do not include event data.

**EventHandler<TEventArgs> Delegate**

Represents the method that will handle an event when the event provides data.

-------------------------------------------------

Events returning void type

Why is it that the return type of events always void?



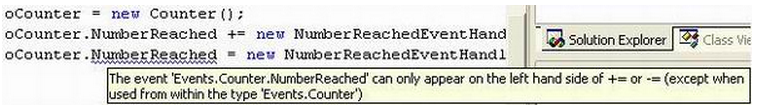
Typically you would not expect to return a value from an event handler as a function return value because an event can have multiple subscribers and each would be returning a return value independently of the other other handlers and would require special event firing code to decide what to do with all the return values.

Typically, if you need to communicate back from an event handler the EventArgs structure would contain members that the handler can update and each handler will get an opportunity to look at the values and update accordingly, and the code firing the event only needs to react to the final value in the structure.

Advantages of Events over Delegates

1. Encapsulation - You declare the delegate as private and then have the event property as public - so it provides encapsulation.

In conclusion: an event declaration adds a layer of protection on the delegate instance. This protection prevents clients of the delegate from resetting the delegate and its invocation list, and only allows adding or removing targets from the invocation list



More Explanation: In essence, declaring the event keyword prevents any of the delegate’s users from setting it to null. Why is this important? Imagine that as a client I would add to the delegates invocation list a callback to one of my class’ functions. So would other clients. All is well and good. Now imagine that someone, instead of using the “+=”, is simply setting the delegate to a new callback by using “=”. This basically just throws the old delegate and its invocation list down the drain and creates a whole new delegate with a single item in its invocation list. All the other clients will not receive their callbacks when the time comes. It is this kind of situation that having the event keyword is aiming to solve. If I keep the event keyword in the Counter class and try to compile the following code in my application, it will cause a compilation error. By removing the event keyword, however, this error will not happen.

1. Only the class that created the event has full control to fire it, the consumer class can only subscribe - The publisher has absolute control to fire the event unlike delegates where anybody can fire.

Misc Event Concepts

Can Interfaces contain Delegates: No

Can Interfaces have Events - <http://msdn.microsoft.com/en-us/library/ak9w5846(v=vs.80).aspx>

Approach when Events Handlers throw exceptions - <http://stackoverflow.com/questions/3114543/should-event-handlers-in-c-sharp-ever-raise-exceptions>

Memory Leak through Events

Definition

The term refer to some kind of resource that is no longer used, but still takes up memory. If you have many of those your application takes a lot of memory and eventually you run out of it.

How can it Happen

* Event in Pub Sub
* Also not removing event listeners. Any event listener that is created with an anonymous method or lambda expression that references an outside object will keep those objects alive. Remember to remove event listeners when they are no longer used.
* Keeping database connections or result sets open when they are not used. Remember to call Dispose() on all IDisposable objects. Use the using statement.
* Call to C functions using p/Invoke which allocate memory which you then never release.

Details



* Whenever a subscriber subscribes to an event that is published by the publisher, the publisher will hold a reference to the subscriber.
* The publisher will hold this reference as long as the event is not unsubscribed. So as long as the publisher is alive, the garbage collector will not collect the subscriber-objects. This leads to a memory leak (or sometimes worse, if the subscriber holds other resources such as files or threads).
* Typically this occurs when the publisher is a static object (which lives as long as the process lives).
* If you unsubscribe from the event (using -=) with an equal handler, then yes, that will remove the handler and the possible leak.

Code to Highlight Memory Leaks

Link 1 - <http://crazorsharp.blogspot.com/2009/03/net-memory-leaks-it-is-possible.html>  
Link 2- <http://stackoverflow.com/questions/12133551/c-sharp-events-memory-leak>

See example E:\04 - Net Technical Material\!Visual Studio 2010 Projects\01-C\_Sharp\_Concepts\aMine\02 - Delegates\_Examples\

24 - MemoryLeak\_Events\_Issue

25 - MemoryLeak\_Events\_Solution

How to detect Memory leaks

Use profiler tools such as the

* ANTS Memory Profiler or
* dotTrace
* Task Manager and the System Monitor let you gather evidence that helps in the diagnosis of memory leaks. WinDbg, in conjunction with the SOS extension, allows you to pinpoint where managed memory is allocated in an application.
* You could also take a look at the CLR Profiler from Microsoft which is free but is more difficult to use.

Code Walkthrough

15-Events\_RandNum\_Ex1 :

* This uses the standard EventHandler<TEventArgs> functionality



* Also the wiring between publisher and subscriber is done in the Main function



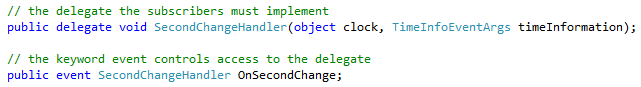
16-Events\_Ex\_2:

* Exactly same as above - only difference is you are passing a name to the consumer from the main program (just a variation - nothing significant)

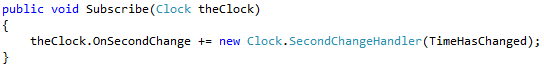


17-Events\_Ex\_3\_Clock:

* This example shows how the CustomEventHandler using delegates is utilized



* Also highlights its using <TEventArgs>
* Wiring of subscriber done within itself (not main)

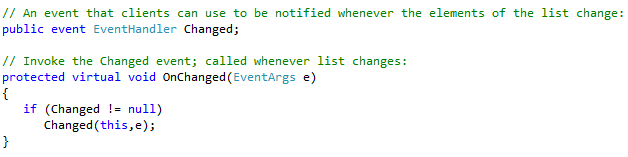


18-Events\_Ex\_4\_List

* Exact same as above (different example)

19-Events\_Ex\_5\_List

* Uses EventHandler and <EventArgs> (not T). Also non-generic version (not imp now)



------------------------------------

**Garbage Collection Talking Points**

|  |
| --- |
| If all code is Managed Code – Garbage collector is called internally and no special code is necessary  Garbage Collection (Non-Deterministic) Handled by CLR (Finalization) **through Object.Finalize** and fully optimized  - Active Roots to build Live Objects  - Generation Concepts |

|  |  |
| --- | --- |
| **Non-Deterministic Cleanup for Non-Managed Resources** | **Deterministic Cleanup** for Non-Managed Resources  (Same approach also be done for Managed resources – but **only** if you want deterministic cleanup) |
| **It is only required in the specific case where a class directly owns an unmanaged resource handle.** | **Prefer as long as your class is not the direct owner of an unmanaged resource.** |
| Un-Managed Code  Garbage Collection Non-Deterministic – (Finalization) **through Destructor** | Un-Managed Code  Garbage Collection In order to be deterministic – the following 2 options are available  Using Statement  Implementing  IDisposable Interface |

|  |  |
| --- | --- |
| **What is a Destructor  (concept of Finalization/also called Finalizers)** | **What is IDisposable Interface**   * Calling Dispose Method (no arguments) |
| The above translates into below Finalize Method internally |
| When the C# compiler compiles a destructor, it implicitly translates the destructor code to the equivalent of a Finalize() method, which ensures that the Finalize() method of the parent class is executed |

|  |
| --- |
| **Dispose Method (overloaded)** |

Garbage Collection

* The steps required to access a resource are as follows:

1. Allocate memory for the type that represents the resource.

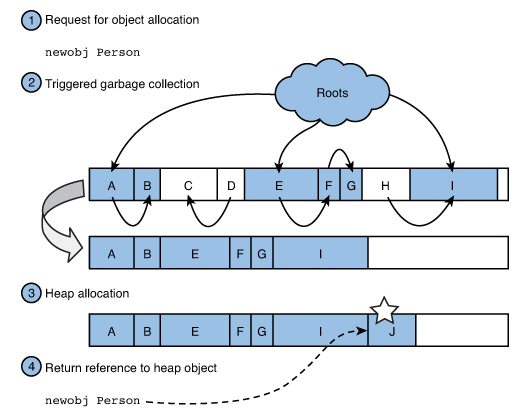
2. Initialize the memory to set the initial state of the resource and to make the resource usable.

3. Use the resource by accessing the instance members of the type (repeat as necessary).

4. Tear down the state of the resource to clean up.

5. Free the memory.

* The garbage collector is .NET's answer to memory management and in particular to the question of what to do about reclaiming memory that running applications ask for.
* Garbage collection is **not deterministic**. In other words, you cannot guarantee when the garbage collector will be called; it will be called when the CLR decides that it is needed, though it is also possible to override this process and call up the garbage collector in your code.



* If Garbage Collector was not available – then uptill now there have been two techniques have been used on the Windows platform for de-allocating memory that processes have dynamically requested from the system:

- Make the application code do it all manually

- Make objects maintain reference counts

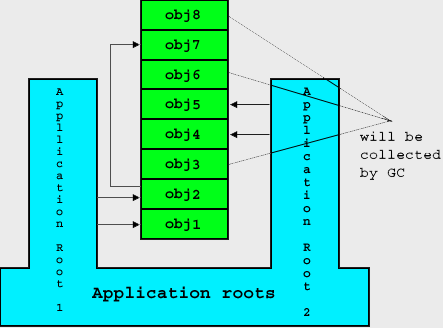
* How does the garbage collector know if the application is using an object or not?

**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 Reachable 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.



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.

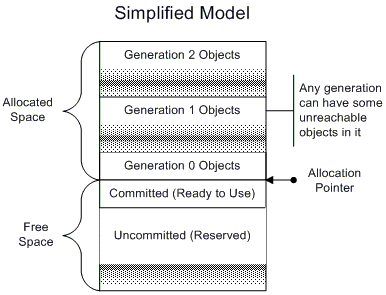
Once all the roots have been checked, the garbage collector’s graph contains the set of all objects that are somehow reachable from the application’s roots; any objects that are not in the graph are not accessible by the application, and are therefore considered garbage.

Garbage Collection Phases

Garbage collection in .NET is done using tracing collection and specifically the CLR implements the Mark/Compact collector.

Three Phases of Garbage collection:

1. Mark: Live objects are traced recursively by following references between objects starting from known live locations: Things on the heap, in static locations, and so on are all within reach of executing code and should not be deallocated.
2. After marking all the live objects, the collector sweeps the heap to free all dead objects. During this phase, objects may be detected that need finalization, complicating this a bit. But in essence, unmarked objects are dead and are subject to cleanup.
3. Compact: Compacting moves objects together so that they make up contiguous space. During this process, references need to be patched up. If an object moves in memory, all objects referring to it need to be tweaked so that they refer to the right location in memory.



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.

One feature of the garbage collector that exists purely to improve performance is called generations. When initialized, the managed heap contains no objects. A generational GC takes into account two facts:

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

The rules for this simplified model are as follows:

* All garbage-collectable objects are allocated from one contiguous range of address space.
* The heap is divided into generations so that it is possible to eliminate most of the garbage by looking at only a small fraction of the heap.
* Objects within a generation are all roughly the same age.
* 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.
* Higher-numbered generations indicate areas of the heap with older objects—those objects are much more likely to be stable.
* The order of objects in memory remains the order in which they were created, for good locality.

Additional Comments

* The oldest objects are at the lowest addresses, while new objects are created at increasing addresses. (Addresses are increasing going down in Figure 1 above.)
* The allocation pointer for new objects marks the boundary between the used (allocated) and unused (free) areas of memory.
* Periodically the heap is compacted by removing dead objects and sliding the live objects up toward the low-address end of the heap. This expands the unused area at the bottom of the diagram in which new objects are created.
* There are never any gaps between objects in the heap.
* Only some of the free space is committed. When necessary, more memory is acquired from the operating system in the reserved address range.

Concept of Live and Dead Objects

|  |  |
| --- | --- |
| **Full Collections** - Once the unreachable objects have been identified we want to reclaim that space for later use; the goal of the collector at this point is to slide the live objects up and eliminate the wasted space. With execution stopped, it's safe for the collector to move all those objects, and to fix all the pointers so that everything is properly linked in its new location. The surviving objects are promoted to the next generation number (which is to say the boundaries for the generations are updated) and execution can resume. |  |
| **Partial Collections** - Unfortunately, the full garbage collection is simply too expensive to do every time, so having generations in the collection helps us out. |

Performance

Now that we have a basic model for how things are working, let's consider some things that could go wrong that would make it slow. That will give us a good idea what sorts of things we should try to avoid to get the best performance out of the collector.

Too Many Allocations -

This is really the most basic thing that can go wrong. Allocating new memory with the garbage collector is really quite fast. So you want to make sure when you're creating new objects that it's really necessary and appropriate to do so, even though creating just one is fast. This may sound like obvious advice, but actually it's remarkably easy to forget that one little line of code you write could trigger a lot of allocations.

*For example, suppose you're writing a comparison function of some kind, and suppose that your objects have a keywords field and that you want your comparison to be case insensitive on the keywords in the order given. Now in this case you can't just compare the entire keywords string, because the first keyword might be very short. It would be tempting to use String.Split to break the keyword string into pieces and then compare each piece in order using the normal case-insensitive compare. Sounds great right?*

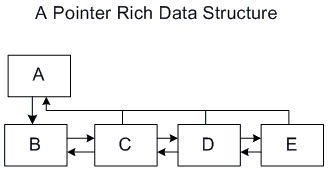
*Well, as it turns out doing it like that isn't such a good idea. You see, String.Split is going to create an array of strings, which means one new string object for every keyword originally in your keywords string plus one more object for the array. Yikes! If we're doing this in the context of a sort, that's a lot of comparisons and your two-line comparison function is now creating a very large number of temporary objects. Suddenly the garbage collector is going to be working very hard on your behalf, and even with the cleverest collection scheme there is just a lot of trash to clean up. Better to write a comparison function that doesn't require the allocations at all.*

Too-Large Allocations -   
When working with a traditional allocator, such as malloc(), programmers often write code that makes as few calls to malloc() as possible because they know the cost of allocation is comparatively high.

In the managed world this practice is much less compelling for several reasons:

* First, the cost of doing an allocation is extremely low—there's no searching for free blocks as with traditional allocators.
* Second, if you do choose to pre-allocate you will of course be making more allocations than are required for your immediate needs, which could in turn force additional garbage collections that might otherwise have been unnecessary.
* Finally, the garbage collector will be unable to reclaim space for objects that you are manually recycling, because from the global perspective all of those objects, including the ones that are not currently in use, are still live.

Too Many Pointers –   
If you create a data structure that is a large mesh of pointers you'll have two problems. First, there will be a lot of object writes (see Figure 3 below) and, secondly, when it comes time to collect that data structure, you will make the garbage collector follow all those pointers and if necessary change them all as things move around



Too Many Almost-Long-Life Objects -

Finally, perhaps the biggest pitfall of the generational garbage collector is the creation of many objects, which are neither exactly temporary nor are they exactly long-lived. These objects can cause a lot of trouble, because they will not be cleaned up by a gen0 collection (the cheapest), as they will still be necessary, and they might even survive a gen1 collection because they are still in use, but they soon die after that.

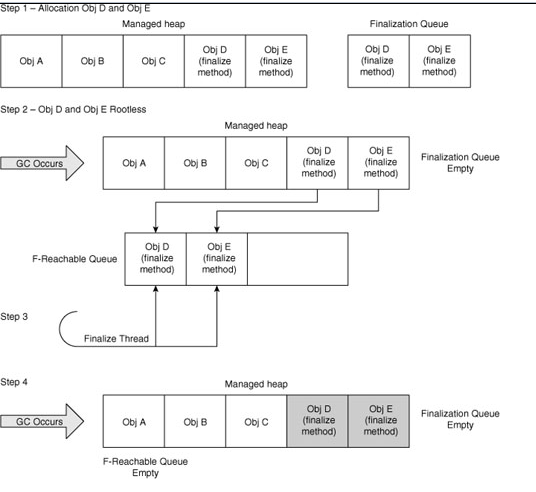
Concept of Finalization

You have seen that constructors allow you to specify actions that must take place whenever an instance of a class is created. Conversely, destructors are called before an object is destroyed by the garbage collector. Given this behavior, a destructor would initially seem like a great place to put code to free unmanaged resources and perform a general cleanup. Unfortunately, things are not so straightforward.

*Although we talk about destructors in C#, in the underlying .NET architecture these are known as finalizers. When you define a destructor in C#, what is emitted into the assembly by the compiler is actually a Finalize() method.*

*Unfortunate Syntax  
The use of the tilde syntax to indicate a finalizer method is a bit unfortunate because the same syntax is used in the world of C++ to indicate a destructor, which is a form of deterministic cleanup. In the world of the CLR, though, the garbage collector causes nondeterministic cleanup, and finalizers are part of that picture. In fact, under the hood, the finalizer is implemented as a Finalize method; you can verify this by looking at the generated IL code. Languages such as Visual Basic require the user to write a Finalize method instead, thus avoiding syntactical confusion.*

Finalizable objects go through an additional step when the garbage collector detects they’re no longer reachable and subject to deallocation. Instead of deallocating the object during the collection cycle, the object is put in a queue (sometimes referred to as the freachable queue, for finalization-reachable queue). Before deallocation happens, the finalizer methods for the objects in that queue are getting called. Finally, the object reaches a state where it can get deallocated just like any regular object. As a rule of thumb, don’t implement a finalizer method unless you absolutely need it. Violating this rule will put additional stress on the garbage collector, which is strongly discouraged.



Destructor

|  |  |
| --- | --- |
| It looks like a method, with the same name as the containing class, but prefixed with a tilde (~). It has no return type, and takes no parameters and no access modifiers. | When the C# compiler compiles a destructor, it implicitly translates the destructor code to the equivalent of a Finalize() method, which ensures that the Finalize() method of the parent class is executed |
|  |  |

Disadvantage of Explicitly calling Destructors

* They are non-deterministic. When a C++ object is destroyed, its destructor runs immediately. However, because of the way the garbage collector works when using C#, there is no way to know when an object's destructor will actually execute. Hence, you cannot place any code in the destructor that relies on being run at a certain time, and you should not rely on the destructor being called for different class instances in any particular order.
* The implementation of a destructor delays the final removal of an object from memory.

Objects that do not have a destructor are removed from memory in one pass of the garbage collector, but objects that have destructors require two passes to be destroyed. The first pass calls the destructor without removing the object, and the second pass actually deletes the object. In addition, the runtime uses a single thread to execute the Finalize() methods of all objects. If you use destructors frequently, and use them to execute lengthy cleanup tasks, the impact on performance can be noticeable.

*Note: Empty destructors should not be used. When a class contains a destructor, an entry is created in the Finalize queue. When the destructor is called, the garbage collector is invoked to process the queue. If the destructor is empty, this just causes a needless loss of performance*.

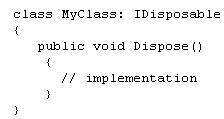
The presence of the garbage collector means that you will usually not worry about objects that you no longer need; you will simply allow all references to those objects to go out of scope and allow the garbage collector to free memory as required. However, the garbage collector does not know how to free unmanaged resources (such as file handles, network connections, and database connections). When managed classes encapsulate direct or indirect references to unmanaged resources, you need to make special provisions to ensure that the unmanaged resources are released when an instance of the class is garbage collected.

Explicit Release of Resources (Deterministic Resource Cleanup)

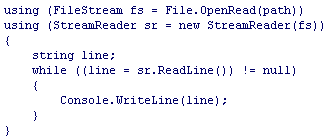
Dealing with various kinds of resources is an everyday task for most developers. This ranges from local files to remote databases or web services. While the .NET runtime has automatic memory management using the garbage collector, other external resources often don’t. In such cases, cooperation from your code might be needed to properly close such resources.

* Implementing a Dispose Method:

In C#, the recommended alternative to using a destructor is using the System.IDisposable interface. The IDisposable interface defines a pattern (with language-level support) that provides a deterministic mechanism for freeing unmanaged resources and avoids the garbage collector-related problems inherent with destructors. The IDisposable interface declares a single method named Dispose(), which takes no parameters and returns void.



* + The IDisposable interface is not something the runtime knows about intrinsically. In other words, this doesn’t change the way the garbage collector will ultimately reclaim the object’s memory allocated on the heap. What it does, on the other hand, is provide a means for code to trigger cleanup of resources the object holds on to (for example, file handles and database connections).
  + There is no performance benefit in implementing the Dispose method on types that use only managed resources (such as arrays) because they are automatically reclaimed by the garbage collector.
* A Dispose method should call the SuppressFinalize method for the object it is disposing. The System.GC class is a .NET class that represents the garbage collect, and the SuppressFinalize() method tells the garbage collector that a class no longer needs to have its destructor called. Because your implementation of Dispose() has already done all the cleanup required, there's nothing left for the destructor to do. Calling SuppressFinalize() means that the garbage collector will treat that object as if it doesn't have a destructor at all. As you know Finalize method is costly for performance and hence SuppressFinalize prevents its Finalize method from being called.
* Using Statement:
* This helps in using a certain resource and want to leave the guaranteed cleanup.

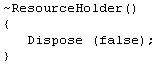


* The using statement consists of two different parts. Between the parentheses is the resource-acquisition expression; it indicates the resource that will be used inside the block underneath and needs to be cleaned up no matter how the block is left. The block itself is simply the code that executes with the acquired resource in scope.

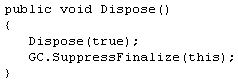
The big question now is how to relate the IDisposable pattern and the use of finalizers with one another. If a user calls the Dispose method explicitly, there’s no need for the finalizer to run anymore because you’ve already got a chance to clean up resources onto which the object holds. On the other hand, if Dispose is never called but the object got unreachable, the finalizer is required to run and do the same cleanup as the Dispose method would have done if it were called properly by the user during the object’s lifetime. So we have the same cleanup logic in both places. Those observations give rise to the following pattern:

Diagrammatic Representation

**Destructor being called**



**Calling Dispose Method through implementation of IDisposable Interface**

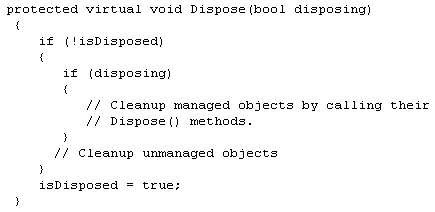


/\* called directly by the user \*/

/\* not called directly by the user \*/ but happens as part of garbage collection if the destructor is written

//Tell GC not to finalize this object

Calling Dispose Method through implementation of IDisposable Interface



You can see from this code that there is a second protected overload of Dispose(), which takes one bool parameter — and this is the method that does all the cleaning up. Dispose (bool) is called by both the destructor and by IDisposable.Dispose(). The point of this approach is to ensure that all cleanup code is in one place.

* If a consumer calls IDisposable.Dispose(), that consumer is indicating that all managed and unmanaged resources associated with that object should be cleaned up.
* If a destructor has been invoked, all resources still need to be cleaned up. However, in this case, you know that the destructor must have been called by the garbage collector and you should not attempt to access other managed objects because you can no longer be certain of their state. In this situation, the best you can do is clean up the known unmanaged resources and hope that any referenced managed objects also have destructors that will perform their own cleaning up.

The parameter passed to Dispose (bool) indicates whether Dispose (bool) has been invoked by the destructor or by IDisposable.Dispose() — Dispose (bool) should not be invoked from anywhere else in your code.

**Additional Concepts**

* GC.Collect Concept
* The System.GC class is a .NET class that represents the garbage collector, and the Collect() method forces a garbage collection to occur irrespective of whether it is needed.
* The GC class is intended for rare situations in which you know that it's a good time to call the garbage collector; for example, if you have just de-referenced a large number of objects in your code.
* Why is calling GC.Collect bad generally
* In general, calling GC.Collect will always lead to a performance loss, never a gain. This will introduce an immediate delay in your code, while the GC runs. If you wait for the GC to collect on its own schedule (which will happen later), it will collect all of your memory then, plus more, so it will technically be more efficient.
* I personally believe direct calls to GC.Collect are typically signs of a design flaw, or a programmer trying to optimize something they don't fully understand. There are almost no places where this is really necessary.
* In which circumstances would you call GC.Collect

Consider calling GC.Collect() if some non-recurring event has just happened and this event is highly likely to have caused a lot of old objects to die.

1. If you have an application that has a lot of objects that are going into Generation 2 and then dying suddenly
2. A classic example of this is if you're writing a client application and you display a very large and complicated form that has a lot of data associated with it. Your user has just interacted with this form potentially creating some large objects... things like XML documents, or a large DataSet or two. When the form closes these objects are dead and so GC.Collect() will reclaim the memory associated with them.

Another reason to call it would be to force the delay to happen when you want it to happen, so there is no chance that there will be a delay from GC during a critical operation. I have some code that is time sensitive, and have wanted to make sure the GC would not collect during this (short) operation, since it can cause some hangs. Calling GC.Collect before the time critical code helps reduce the likelihood of a collection occuring during the operation, which could cause a hang that was bad.

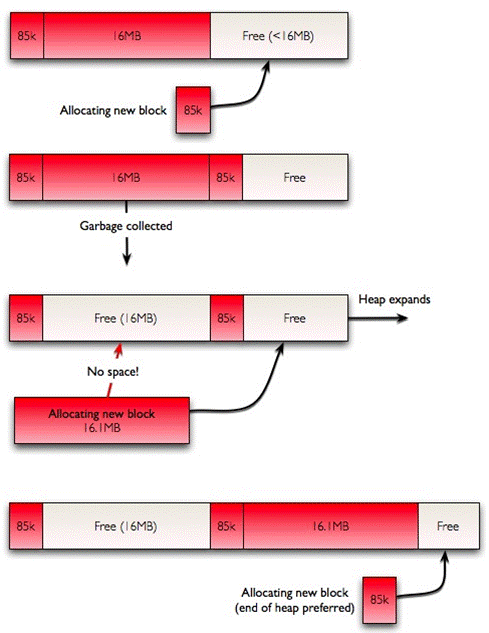
Large Object Heap

NET manages memory in four different regions, known as heaps. You can think of each of these as continuous regions of memory, though in practice .NET can create several fragmented regions for each heap. Three of the heaps, called the generation 0, 1 and 2 heaps are reserved for small objects: In current versions of .NET ‘small’ means objects that are 85,000 bytes or less. Any object is assigned to one of these generations according to the number of garbage collections it has survived, the veterans ending in generation 2. .NET knows that younger objects are more likely to be short lived and can reduce the performance cost of garbage collections by initially only looking at the recently allocated objects on generation 0. Perhaps more importantly, it can also move the survivors of a collection around so that there are no gaps, ensuring that the free space available for new objects is always together in one large lump. This helps with performance - .NET never needs to search for a hole big enough for a new object, unlike unmanaged code: if there’s enough memory available it’s always in the same place. When it needs to compact a series of objects, .NET actually copies all of them to a new block of memory rather than moving them in place; this improves performance by simplifying how objects are allocated. In these small heaps this means that the free space is always at the end, so there is never any need to scan elsewhere for a ‘hole’ big enough to store a new object.

The fourth heap is known as the Large Object Heap, or LOH. ‘Big’ objects go here - as the size at which an object may end up on this heap is 85,000 bytes, this usually means arrays with more than about 20,000 entries. It’s treated separately by the garbage collector, which will generally try to reclaim space from the other heaps before trying to tackle the giant objects that lurk here. Large objects pose a special problem for the runtime: they can’t be reliably moved by copying as they would require twice as much memory for garbage collection. Additionally, moving multi-megabyte objects around would cause the garbage collector to take an unreasonably long time to complete.

.NET solves these problems by simply never moving large objects around. After large objects are removed by the garbage collector, they leave behind holes in the large object heap, thereby causing the free space to become fragmented. When there’s no space at the end of the large object heap for an allocation, .NET searches these holes for a space, and expands the heap if none of the holes are large enough. This can become a problem. As a program allocates and releases blocks from the large object heap, the free blocks between the longer-lived blocks can become smaller. Over time, even a program that does not leak memory, and which never requires more than a fixed amount of memory to perform an operation, can fail with an OutOfMemoryException at the point that the largest free block shrinks to a point where it is too small for the program to use. In the worst cases, the amount of usable memory can shrink at a rate that seems unbelievable.

The design of the Large Object Heap means that its’ worse case will occur when the short-lived arrays are large and can vary in size a bit, and if the longer-lived arrays are small. The variation in size makes it likely that, when a new short-lived array is allocated, it won’t fit into the space left by one of its predecessors; .NET will then have to extend the LOH to ever-greater extents. This might not seem so bad: surely the smaller blocks can be put in the large gaps that are left behind?



What does this look like in a practical program? It seems fairly esoteric: allocate something big, then something small, keep the small thing and throw away the big thing. There are a few scenarios where this pattern can appear, and it can drastically reduce the amount of memory available to a program. The most trivial scenario could just be this:

List < int > list;

...

list.Add(x);

A List object’s underlying implementation is an array and when it needs to grow, .NET allocates a new, larger array and throws the old one away. All that’s needed to start to cause problems with fragmentation is to put some other object on the LOH before the array needs to grow. The main effect here is that you might be surprised how small the size is that your ‘large’ list can grow to before .NET tells you that it has run out of memory - long before the program is using up its 2Gb theoretical maximum.

It is difficult to diagnose this problem with the tools that are currently available. The best approach available at the moment is to combine a memory profiler with the performance counters. You can look at the ‘Large Object Heap Size’ performance counter to discover how many bytes are allocated to the large object heap. This counter includes free space, so to discover whether or not the problem is that memory has run out or because the memory that is available has become too fragmented to be used you will need to use a memory profiler. A good profiler will be able to tell you the total size of all of the live objects on the large object heap. If there is a big discrepancy between the two numbers, then it is possible that your program is suffering from heap fragmentation. Unfortunately, this isn’t guaranteed: there’s no problem if there’s a large amount of free space at the end of the heap - .NET just won’t clean it up until it has to for performance reasons.

Working around the problem

The best way to prevent fragmentation from occurring is to ensure that no objects above 85k in size are used for permanent storage. This ensures that the free space in the large object heap will always end up in one big chunk as short-lived objects are removed from play.

If large data structures need to live for a long time, and especially if they need to grow in size over time, the best approach is simply to consider using or writing a different data structure to store them. Arrays can contain up to around 10,000 elements before they are put on the large object heap and can cause problems, so a very effective way to store 100,000 entries might be to store 10 arrays each containing 10,000 elements: none will end up on the large object heap so no fragmentation will occur. This could be written as an IList subclass, which would make it easy to drop in transparently to replace existing code.

Depending on the design of the code, a simpler approach might simply be to make the ‘temporary’ but large structures more permanent. If a program needs to deal with a set of large files, it will make more efficient use of memory if it allocates enough space to deal with the largest one first, and then re-uses that for every file instead of allocating just enough for each and then throwing the memory away later. Many of the classes in System.Collections have a ‘Capacity’ property to facilitate this design pattern, and thread local static variables can be used (with care) to help share a large data structure in multithreaded applications.

It is unfortunate that the CLR garbage collector deals with large objects in this manner. The effect is similar to a memory leak, but one where we get no clue as to the cause by measuring the size of the objects in memory: a memory profiler will tell you that there’s no problem, while task manager tells you that the program’s memory usage is growing ever larger – which could be due to this issue or which could be due to the behavior of unmanaged code. What is needed instead is a way of finding out where objects are in memory and how they are affecting the free memory available to .NET. This can be done with some detective work: code can be inspected for areas where large amounts of memory are likely to be allocated, and a memory profiler (ANTS memory profiler) can be used to help discover these areas. Once these are known, a debugger can be used to step through the code and some pen and paper can give an idea of what the large object heap looks like, which can be used to decide on the best way forward.

Important Concepts

Object

A distinct, named set of attributes that represents something concrete, such as a user, a printer, or an application. The attributes hold data describing the thing that is identified by the directory object. Attributes of a user might include the user’s given name, surname, and e-mail address.

|  |  |
| --- | --- |
| Name | Description |
| Equals | Determines whether the specified Object is equal to the current Object. |
| Finalize | Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection. |
| GetHashCode | Serves as a hash function for a particular type. |
| GetType | Gets the Type of the current instance. |
| ToString | Returns a string that represents the current object. |
| ReferenceEquals | Determines whether the specified Object instances are the same instance. |

Equals

* Signature: 

Finalize

* Signature - 
* Allows an obj to try to free resources and perform other cleanup operations before its reclaimed by GC
* This is only necessary when you create objects that encapsulate unmanaged resources, you must explicitly release the unmanaged resources when you are finished using them in your application
* Finalize is automatically called only once on a given instance, unless the object is re-registered using a mechanism such as GC.ReRegisterForFinalize and GC.SuppressFinalize
* Implementing Finalize methods or destructors can have a negative impact on performance and you should avoid using them unnecessarily.

Implementation

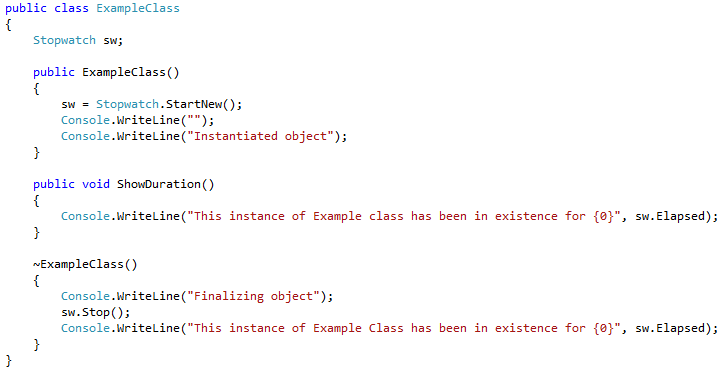
* To implement the Finalize method in C#, you must use destructor syntax.
* If an Object holds references to any resources, Finalize must be overridden by a derived class in order to free these resources before the Object is discarded during garbage collection.

Every implementation of Finalize in a derived type must call its base type's implementation of Finalize. This is the only case in which application code is allowed to call Finalize.

* This method is automatically called after an object becomes inaccessible, unless the object has been exempted from finalization by a call to GC.SuppressFinalize.
* A Finalize method should not throw exceptions, because they cannot be handled by the application and can cause the application to terminate.

Additional Details

* The garbage collector keeps track of objects that have Finalize methods, using an internal structure called the finalization queue. Each time your application creates an object that has a Finalize method, the garbage collector places an entry in the finalization queue that points to that object. The finalization queue contains entries for all the objects in the managed heap that need to have their finalization code called before the garbage collector can reclaim their memory.
* Implementing Finalize methods or destructors can have a negative impact on performance and you should avoid using them unnecessarily. Reclaiming the memory used by objects with Finalize methods requires at least two garbage collections.



GetHashCode

* Signature 
* Defined in the System.Object class
* The Object class's default implementation of GetHashCode() returns a unique integer that is guaranteed not to change during the lifetime of the object.

The GetHashCode method can be overridden by a derived type. If GetHashCode is not overridden, hash codes for reference types are computed by calling the Object.GetHashCode method of the base class, which computes a hash code based on an object's reference

* Because every type is derived, either directly or indirectly, from Object, all objects have access to this method.
* This method can be overridden to provide a hash function more suitable to a specific class – eg: The Point class in the System.Drawing namespace, for example, overrides GetHashCode(), returning the XOR of its x and y member variables
* A hash code is a numeric value that is used to insert and identify an object in a hash-based collection such as the Dictionary<TKey, TValue> class, the Hashtable class, or a type derived from the DictionaryBase class. The GetHashCode method provides this hash code for algorithms that need quick checks of object equality.
* You should never persist or use a hash code outside the application domain in which it was created, because the same object may hash across application domains, processes, and platforms.

**Characteristics of Hash Code Implementation:**

* if two things are equal (Equals(...) == true) then they must return the same value for GetHashCode()
* if the GetHashCode() is equal, it is not necessary for them to be the same; this is a collision, and Equals will be called to see if it is a real equality or not.
* There are a bunch of collision issues which come into play when computing the HashCode (if done by you).
* A hash function is used to quickly generate a number (hash code) that corresponds to the value of an object. Hash functions are usually specific to each type and, for uniqueness, must use at least one of the instance fields as input. Hash codes should not be computed by using the values of static fields.

GetType

* Signature: 
* Internally uses Reflection.
* GetType() return always the type of the current object, not the pointed object.

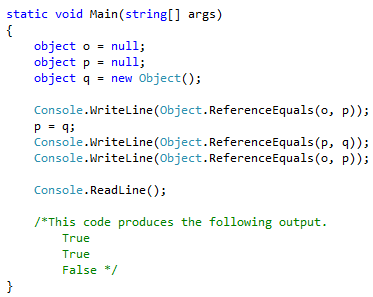
ToString

* Signature: 
* It converts an object to its string representation so that it is suitable for display.

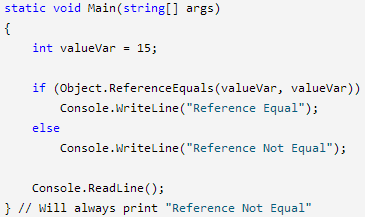
|  |  |
| --- | --- |
| Tostring default usage | ToString over-ridden |
|  |  |

ReferenceEquals

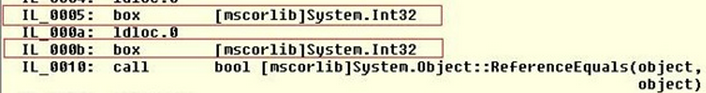
* Signature: 
* Determines whether the specified Object instances are the same instance.
* Unlike the Equals method and the equality operator, the ReferenceEquals method **cannot be overridden**. Because of this, if you want to test two object references for equality and are unsure about the implementation of the Equals method, you can call the ReferenceEquals method.



* However, note that if objA and objB are value types, they are boxed before they are passed to the ReferenceEquals method.



If you examine Object.ReferenceEquals it is designed to take two Objects as the input parameters. So when you pass Value Types to it .Net goes ahead and "Boxes" the parameters. Here is the corresponding MSIL that gets generated for the above code. Corresponding MSIL below:



System.ValueType

System.Object

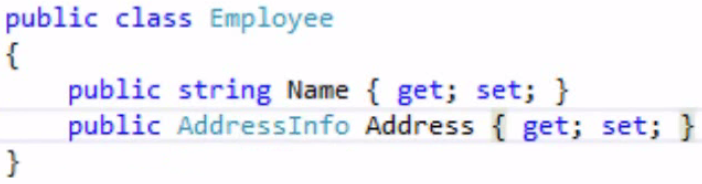
System.ValueType

* As you know, there are total two types in CLR: one is a value type; the other is a reference type.
* All value types inherit from System.ValueType.
* ValueType and all types in turn inherit from System.Object.

Composition versus Inheritance

Composition: Has-A relationship

Employee Object owns the lifetime of the AddressInfo Object. This is an example of composition



Inheritance: Is-A relationship

|  |  |
| --- | --- |
|  | Every Manager is-a employee and hence shows inheritance |

Difference between decimal, double and float

Precision is the main difference.

Float - 7 digits (32 bit) - single precision  
Double-15-16 digits (64 bit) - double precision  
Decimal -28-29 significant digits (128 bit) - 128-bit floating point data type

float and double are floating binary point types. In other words, they represent a number like this:

decimal is a floating decimal point type. In other words, they represent a number like this:  
 

Decimals have much higher precision and are usually used within financial applications that require a high degree of accuracy. Decimals are much slower (up to 20X times in some tests) than a double/float.

Suffix table when declaring a type



Additional Information

* During division – you need to cast one of the variables as output type (for example decimal) if you want the expected output to be decimal (Division of 2 ints will give an answer as int)
* Modulus: The % operator computes the remainder after dividing its first operand by its second. All numeric types have predefined remainder operators.

Weak References

Definition:

* A weak reference is a special type of reference where the garbage collector can still collect the underlying object, even though you still technically have a reference to it. Hence a weak reference object will keep a reference to an object that went out of scope or was set to null, until it is actually deleted by the garbage collector. But until that happens, we can get the object back!
* When you talk about a reference to an object in .NET - you are talking about a "strong" reference. As long as that reference exists, the garbage collector won't collect the object behind that reference.
* The key here is to remember that the garbage collector is not running all the time. As far as we, the programmers of an application, know it is completely random and could kick in at any time. This means that an object only referenced through a weak reference could sit around for long time, or for virtually no time at all (and really, it is even more complicated than trying to figure out the next time the garbage collector will run - because the garbage collector for C# is generational).
* And, as soon as you copy the reference out of the weak reference variable into a regular reference, the underlying object will no longer be collected (assuming that it hadn't already been collected), because now you have a strong reference to it.

When do we use it:

* Mildly used Cache Scenario - Say I had a large chunk of external data that would be handy to keep in memory, but really isn't used very often (or maybe it is used a bunch in bursts).
* A weak reference is a reference that does not keep the referenced object alive. This is especially handy for circular references (where objects keep themselves alive and need extra work by the garbage collector to detect they’re actually both dead instead, for example a an item in a list that also references the list it’s in).



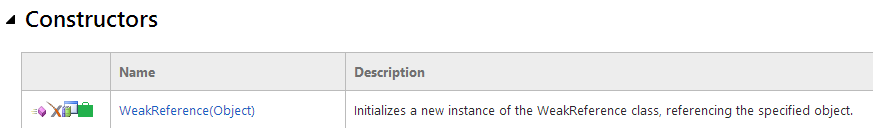
How does it work

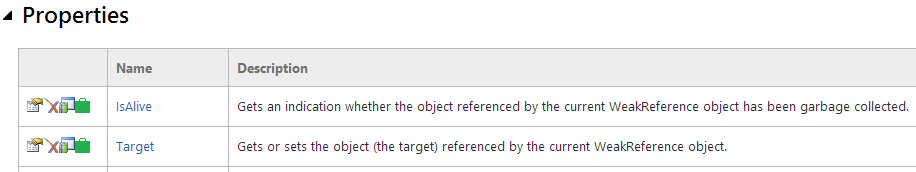
Step 1: To create a weak reference to an object, we simply call the constructor with the object:   
   
Step 2: To get the underlying object from a weak reference, we use its .Target property:   
   
If the object still exists, it is returned. If it was claimed by the garbage collector, a null reference is returned by the 'as' operator.

Properties:

**Target**: The Target property returns an object that contains reference to the instance you stored in the WeakReference. You should cast the object using the as-operator or explicit cast before using the instance.

**IsAlive**: One important property on the WeakReference type is the IsAlive property. Boolean property that indicates whether the object pointed to by the WeakReference has been collected or not.





**Best Practice**

Always remember to pull the reference out into a strong (or regular) reference before you do any manipulation or checking - otherwise, stuff could change out right from under you.

|  |  |
| --- | --- |
| **Bad Practice** | **Good Practice** |
|  |  |

Disadvantage:

* Weak references aren't a guaranteed profit for application performance. In most cases, they will make an algorithm more performant than when using very large local variables. But it's not guaranteed, and in some cases it could produce noticeable overhead.

So the best advice with weak references is: profile or benchmark it, to make sure that you choose the best solution for your specific situation.

**More Article/Code Example**: <http://msdn.microsoft.com/en-us/library/system.weakreference.aspx>

<http://csharp.2000things.com/tag/weakreference/>

**Further Concepts Reading**

Weak Events - <http://msdn.microsoft.com/en-us/library/aa970850.aspx>  
Weak Events/ Event Aggregator - <http://mark-dot-net.blogspot.com/2012/10/understanding-and-avoiding-memory-leaks.html>

Closures

1. Concept of Closure:

Definition

* Closures are usually associated with functional programming languages, where they link a function to its referencing environment, permitting access to variables outside of the function's scope.
* A closure is a first-class function with free variables that are bound in the lexical environment
* A closure is a block of code which can be executed at a later time, but which maintains the environment in which it was first created - i.e. it can still use the local variables etc of the method which created it, even after that method has finished executing.
* The general feature of closures is implemented in C# by anonymous methods and lambda expressions.

More Granular Details:

* When you create the function, the variables that it will use that are outside of its visible scope are, in concept, copied and stored with the closure's code. They can then be used whenever the delegate is called. This leads to great flexibility when using such delegates but also introduces the possibility of unexpected bugs.   
  *When the function is created, the external variables that it uses are "closed over", meaning that they are bound to the closure function in a way that makes them accessible.*

Inner Framework Details

* When you build your project, the compiler generates a new, hidden class that encapsulates the non-local variables and the code you include in the anonymous method or lambda expression. The code is included in a method and the non-local variables are represented as fields. This new class's method is called when the delegate is executed.

2. Side Effects of Closure

Example / Side-Effect 1 (Very basic scenario)

|  |  |
| --- | --- |
| Using Anonymous method | Using Lambda Expression |
|  |  |

Example / Side-Effect 2 (Class level declaration and pre initialization before Action is called)

|  |  |
| --- | --- |
| Consider the following code. Here the closure is held in a class-level Action variable. The Main method calls the SetUpClosure method to initialise the closure before executing it. The SetUpClosure method is important. You can see that the integer variable is created and initialized and then used within the closure. On completion of the SetUpClosure method, this integer variable goes out of scope. However, we are still invoking the delegate after this happens. Now the result is not so obvious. Will it compile and run correctly? Will an exception occur when trying to access the out-of-scope variable?  This is the closure in action. The "nonLocal" variable was captured, or "closed over", by the delegate code, causing it to remain in scope beyond the normal limits. In fact, it will remain available until the no further references to the delegate remain. You can see that we get the same result as in the original example. |  |

Example / Side-Effect 3 (Closures Capture Variables, Not Values)

|  |  |
| --- | --- |
| Some programming languages capture the values of variables used in closures when those closures are defined. C# captures the variables themselves. This is an very important distinction.  To illustrate, consider the following code. Here we are creating a closure that outputs our familiar mathematical equation. When the delegate is declared, the value of the integer variable is 1. However, after the closure is declared, but before it is executed, the variable's value changes to 10. | When called |

Example / Side-Effect 4 (Changes to closure variable transmitted in other direction too)

|  |  |
| --- | --- |
| Changes to a non-local closure variable are transmitted in the other direction too. In the following code the delegate changes the value before its called. |  |

Example / Side-Effect 5

|  |  |
| --- | --- |
| Here we can see that the action returned by CreateAction still has access to the counter variable, and can indeed increment it, even though CreateAction itself has finished. |  |

Example /Side-Effect 6 (Loop behavior – even gets more interesting with closures)

Ex-1

|  |  |
| --- | --- |
| This time we are using closures in a common scenario: multi-threaded or parallel programming. The code below shows a for loop that creates and starts five new threads. Each pauses briefly before showing the value from the loop's control variable. If the value of the control variable were captured, we'd see the numbers from one to five written to the console, though perhaps not in the correct order. However, as it is the variable that is bound to the closure, and the loop completes before the threads output their messages, we actually see the final value of 6 for each thread. |  |
| Luckily, this type of problem is easily fixed when you understand that it is variables and not values that are captured. All we need to do is to introduce a new variable instance for each iteration of the loop. This can be declared in the loop's body and given the value from the control variable. Under normal circumstances the temporary variable would go out of scope when the loop terminates but the closure will bind to it and keep it alive.  In the code below five instances of the "value" variable are created and given five different values, each bound to a different thread. |  |

Ex-2

|  |  |
| --- | --- |
| Another Example – without local instance variable | Solving the Closure issue with local variable |
|  |  |

Extra Reading -

1) <http://www.codethinked.com/c-closures-explained>

2) <http://richnewman.wordpress.com/2011/08/06/closures-in-c/>

Difference between Equals() versus ==

|  |  |
| --- | --- |
| Equals Method | ==  The == operator is an operator which can be overloaded by classes.  More efficient since its done at compile time (static) |

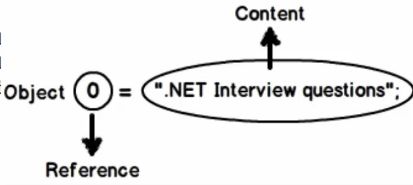
For value types

== and Equals() usually compare two objects by value.

* As you know, all value types inherit from System.ValueType.
* ValueType and all types in turn inherit from System.Object.
* Object offers a Equals() method, used to determine whether two objects are equal. Please do bear in mind that ValueType overrides the Equals() method of Object. When we compare whether two value typed variables are equal, we can call the Equals() method inherited from the ValueType type.
* The default implementation of System.ValueType.Equals uses reflection. Of course, there exists poor performance in reflection. In other word, to call the Equals() method towards the value type will be very expensive.

|  |  |
| --- | --- |
| For example:  int x = 10;  int y = 10;  Console.WriteLine( x == y );  Console.WriteLine( x.Equals(y) ); | **Display:**  True  True |

For Reference types



|  |  |
| --- | --- |
| Scenario 1:  All objects except string | Equals Method   * Reference types derive from System.Object, which defines a virtual Equals method. This method does a simple reference check to see if the two items point to the same object in memory. If they do, it returns true; otherwise, it returns false. * The Equals(Object) method tests for reference equality, and a call to the Equals(Object) method is equivalent to a call to the ReferenceEquals method.   == Operator  For reference types other than string, == returns true if its two operands refer to the same object.  **Technically both Equals() and == on objects other than string do reference equality** |
| Scenario 2:  String Object | Equals Method  System.String overrides the Equals method to also compare the contents of the strings.  == Operator  For the string type, == compares the content of the strings.  **Only for strings both Equals() and == do content equality** |

**Object == and Equals Example**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Two Different References having same value** | **Two exact same references** | |  |  | |  |  | |  |

**String == and Equals Example**

|  |  |
| --- | --- |
| **Example 1** | **Example 2** |
|  |  |
| **Example 3 (mix scenario)** | **Example 4 (weird – don’t bother too much)** |
| In Stringbuilder the Equals method returns true if:   * this instance and sb have equal string (content) * Capacity * and MaxCapacity values;   otherwise, false |  |

Bottom line

* For value types, I'd normally use == for easier-to-read code.
* My rule of thumb is that for almost all reference types, if you want content comparison, you need to override Equals method but be aware that it will only work if the type has overridden the default implementation.
* If you want to perform an identity comparison use the ReferenceEquals() method.
* The exception is for strings - comparing strings with == does make things an awful lot simpler and more readable but you need to remember that both sides of the operator must be expressions of type string in order to get the comparison to work properly.

Best Practices while overriding Equality

Now if you want to do object Equality but lot of times reference types need to define value equality (content based) instead of reference equality.– then you need to override the Equals method and write code yourself. Below are the best practices:

|  |  |
| --- | --- |
| **Step 1: Couple of checks before u start property to property comparison**   1. Check 1: If the object reference is null   First, let's check and see if we're comparing against null. Because we're inside an instance method, "this" is not going to be null, so we only need to check the other side of the comparison (obj).   1. Check 2: Reference Equality   The next step is a simple -- and quick -- reference check. The Object class exposes a static ReferenceEquals method that indicates if the two objects point to the same item in memory. If they're the same object, there's no need to go any further and we can return true.   1. Check 3: If of the same types using reflection GetType()   Next, we do a type check to make sure we're comparing like objects. If the objects aren't of the same type, there's no way they could be equal.  **Step 2:**   1. Finally, we get to our customized logic of checking the ID property. If the two IDs match, it represents the same object in the database and satisfies the equality check for our Customer object.   The validation and checking before we even get to the comparison code is necessary because an Equals override should never throw an exception. The objects are either equal or not equal -- there's nothing else that can make logical sense and hence no exceptions should be thrown.  It is recommended that any class that overrides Equals also override System.Object.GetHashCode. |  |

Relationship between Equals and GetHashCode

* If two things are equal (Equals(...) == true) then they must return the same value for GetHashCode()
* if the GetHashCode() is equal, it is not necessary for them to be the same; this is a collision, and Equals will be called to see if it is a real equality or not.

**By overriding Equals you're basically stating that you are the one who knows better how to compare two instances of a given type, so you're likely to be the best candidate to provide the best hash code.**

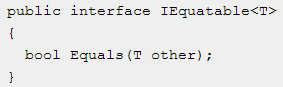
IEquatable<T>

Boxing and Type Safety:

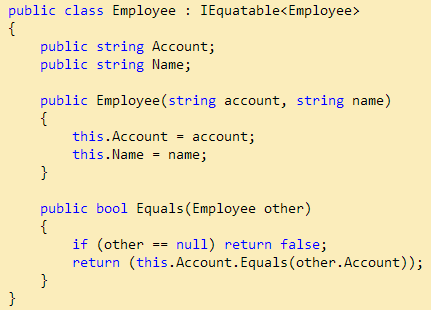
As you probably noticed, the signature for Equals accepts an object. This means two things:

* For reference types, we lose some type-safety flexibility.
* For value types, we pay the cost of boxing and unboxing.

Microsoft realized this, and when the company introduced generics in the .NET Framework 2.0, it gave us a new interface for equality comparison that was type-safe -- IEquatable<T>



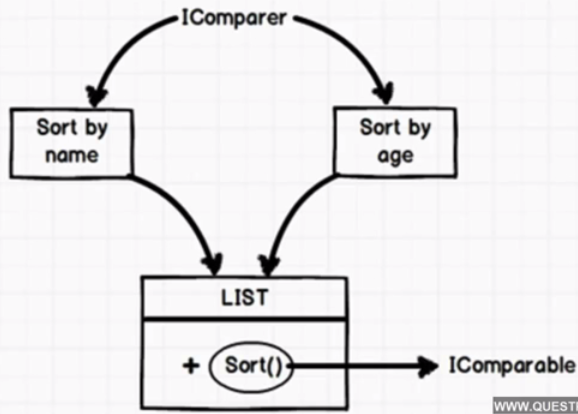
Now we can use this interface to get type safety for our reference types and remove the boxing and unboxing of our value types. Let's revisit the Point3d example and implement IEquatable<T>



Read more - <http://www.codeproject.com/Articles/20592/Implementing-IEquatable-Properly>

Articles from where notes were made:  
<http://visualstudiomagazine.com/articles/2011/02/01/equality-in-net.aspx>  
<http://www.infoq.com/articles/Equality-Overloading-DotNET>

IComparable versus IComparer



IComparable<T>

* Used to provide a default sort order for your objects
* Defines an interface for an object with a CompareTo() method that takes another object of the same type and compares the current object to the passed one.
* It internalizes the comparison to the object, allowing for a more inline comparison operation, and is useful when there's only one logical way, or an overwhelmingly common default way, to compare objects of a type.
* As the name suggests, IComparable<T> reads out I'm comparable
* The CompareTo method is implemented by types whose values can be ordered or sorted. It is called automatically by methods of non-generic collection objects, such as Array.Sort, to order each member of the array.
* Array.Sort internally uses the QuickSort algorithm
* Values which are output from the CompareTo Method

|  |  |
| --- | --- |
| Less than zero | This instance precedes obj in the sort order. |
| zero | This instance occurs in the same position in the sort order as obj. |
| Greater than zero | This instance follows obj in the sort order. |

IComparer<T>

* Defines an interface with a Compare() method that takes two objects of another type (which don't have to implement IComparable) and compares them.
* This externalizes the comparison, and is useful when there are many feasible ways to compare two objects of a type

**Finer Points**

* Use IComparable for value types and IComparer for reference types.
* Also ideally you can have only one IComparable<T> while multiple IComparer<T> is possible based on different criteria.

Specific string Implementation

The CompareTo method was designed primarily for use in sorting or alphabetizing operations. It should not be used when the primary purpose of the method call is to determine whether two strings are equivalent. To determine whether two strings are equivalent, call the Equals method.

The IComparer<T> and IComparable<T> are exactly analogous to IEqualityComparer<T> andIEquatable<T> which are used for testing equality rather than comparing/sorting; a good thread [here](http://stackoverflow.com/questions/9316918/what-is-the-difference-between-iequalitycomparert-and-iequatablet)where I wrote the exact same answer :)

IEnumerable versus IEnumerator

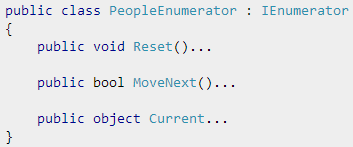
IEnumerable

* IEnumerable is built on top of IEnumertor. It makes syntax shorter and sweeter
* Inheriting from IEnumerable means your class returns an IEnumerator object

|  |  |
| --- | --- |
| Example 1 | Example 2 |
|  |  |

* Thus when your class implements IEnumerable, you are saying that you can call a method (GetEnumerator) and get a new object returned (an IEnumerator) you can use in a loop such as foreach.
* Anything in .Net that you can iterate over implements IEnumerable
* Important Point: State – IEnumerable does not remember which row it is currently iterating through (see code example)

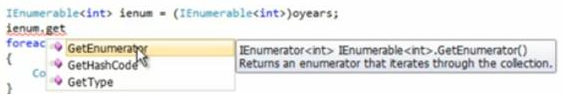
IEnumerator

* Inheriting from IEnumerator means your class returns the methods and properties for iteration:
* Important Point IEnumertor – does remember the state so it remembers the row (see code example)

**Finer Points**

* Implementing IEnumerable means that the object can be iterated over. This doesn't necessarily mean its an array as there are certain lists that can't be indexed but you can enumerate them.
* IEnumerator is the actual object used to perform the iterations. It controls moving from one object to the next in the list.
* Most of the time, IEnumerable & IEnumerator are used transparently as part of a foreach loop. When you write a foreach loop in C#, the code that it generates calls GetEnumerator to create the Enumerator used by the loop.

Interview Question – if you want to make a List just read only and not have the user be able to add/remove/modify elements – then what should you do?



IEnumerable versus IQueryable

IEnumerbale

* IEnumerable exists in System.Collections Namespace.
* IEnumerable is best to query data from in-memory collections like List, Array etc. It is suitable for LINQ to Object and LINQ to XML queries.
  + When you use List<T> and LINQ to Objects, you load the entire "table" of data into memory, then run your query against it.
* LINQ queries against IEnumerable<T> produce delegates (methods) which, when invoked, perform the described query.

IQueryable

* IQueryable exists in System.Linq Namespace.
* IQueryable is best to query data from out-memory (like remote database, service) collections and is suitable for **LINQ to SQL** queries.
* While query data from database, IQueryable **execute select query on server side with all filters**.
* LINQ queries against IQueryable<T> produce expression trees, a data structure which represents the code that produced the query. LINQ providers such as LINQ to SQL interpret these data structures, generating the same query on the target platform (T-SQL in this case).
  + Running the actual query on the server environment is much more efficient since you get the actual results (which are executed on the server) rather than passing in the full dataset back to the caller and then do the filtering on it.

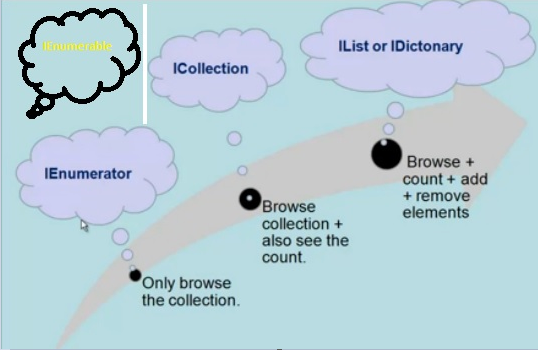
Example 1: The central object here is a DataContext, which represents our database-wrapper. This typically has a property per table (for example, Customers), and a table implements IQueryable<Customer>.

|  |  |
| --- | --- |
|  | which is again interpreted (by the C# compiler) as: |

Example 2:



IEnumerator, ICollection, IDictionary and IList



1. **Encaptulation**: Implement encapsulation on core collections like ArrayList, Hashtable or any generic collections
2. **Polymorphism**: Dynamically point to any collection – Hashtable, ArrayList etc.

|  |  |
| --- | --- |
| Encaptulation | IEnumerator interface – only lets you to browse through the collection and not add, remove. ICollection interface – lets you browse as well as get a count value IList: for ArrayList IDictionary: for Hashtables |
| Polymorphism | IEnumerable: Helps achieve Polymorphism on top of IEnumerator |

What's the difference between IEnumerable<T> and List<T> ?

* IEnumerable is an interface, where as List is one specific implementation of IEnumerable. List is a class.
* FOR-EACH loop is the only possible way to iterate through a collection of IEnumerable, where as List can be iterated using several ways. List can also be indexed by an int index, element can be added to and removed from and have items inserted at a particular index.
* IEnumerable doesn't allow random access, where as List does allow random access using integral index.
* In general from a performance standpoint, iterating thru IEnumerable is much faster than iterating thru a List.

Reflection

What is reflection ?

Reflection helps to inspect content/metadata of an assembly whether they are public or private.

How to Implement Reflection

3 steps process:

1. Import NameSpace
2. Get the Type of the object
   1. Get the reference of the assembly

Get the reference of the class (using GetType)

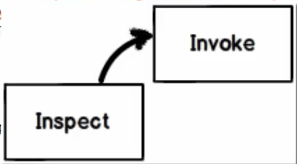
Once you get a type of the object – you can then create instance (when library is referenced directly though Assembly.Load

* 1. Get Class Type

1. Browse Metadata and/or invoke members

Benefits of Reflection

1. Help Inspect Metadata as shown above.
2. Not only does it help inspect metadata, it can help invoke a certain method or set a values on certain members – so complete dynamic invocation (runtime invocation).



Practical Use-Cases of Reflection

* View and traverse the information of the basic types (and their members) and the metadata inside assemblies;
* Late bind methods and properties;
* Dynamically create type instances (and can dynamically invoke the methods, fields, properties in created instances).
* If you are creating application like Visual Studio where you want to show internal of an object by using intellisense.
* In unit testing sometimes we need to invoke private methods. That’s a different thing whether we this is correct way to test private members or not.

Type Class

The Type class plays the core role in .NET reflection, which not only encapsulates information about the objects, but also is the entrance of reflection. As soon as you get the Type object of a specified type, you can acquire all the information (methods, fields, properties, events, parameters, constructors, etc.) of this type using the properties and methods provided by the Type object.

To do this, the first step is to get the Type instance of a specified type. On the whole, there are two forms to get the Type object:

* one is to obtain the type from inside the currently loaded assembly (Runtime);
* the other is from the unloaded one.

Three approaches for runtime related Type:

1. Use the static method GetType() provided by the Type class

For example, if we want to get an instance of Type of the Stream type, we can do like this:

 Note the GetType method accepts a string typed argument.

2. Use the typeof operator  
 

3. Obtain the Type object through instance of a type  
 

Difference between GetType() and typeof

|  |  |
| --- | --- |
|  | GetType() - When you have objects and you want to compare type of those objects.  typeof() – Helps to get from a class. More static |
|  | |

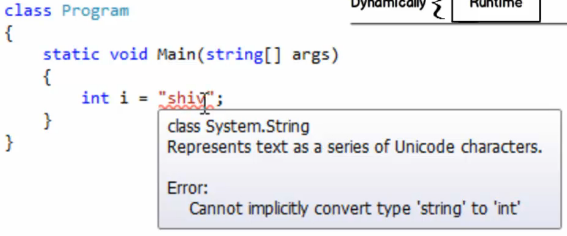
Difference between Reflection and dynamic keyword

|  |  |
| --- | --- |
|  | Dynamic keyword uses reflection internally |

Some background on dynamic keyword:

|  |  |
| --- | --- |
|  |  |

Strongly typed languages where data type checks happen at compile time. See example below – compile time error:



If I use dynamic keyword – during compile time – it will not show error but during runtime – it will complain – hence it will give strongly type during runtime.

|  |  |
| --- | --- |
|  |  |

Static Variables/Methods/Classes

Static Variables

|  |  |
| --- | --- |
| Non Static Example | Static Example |
|  |  |
| Explanation: If you look at the above example I just declared int variable. When you run this code the output will be 10 and 10. Its simple | Explanation: Now when I run above code then the output will be 10 and 15. So static variable value shared among all instances of that class. |
| FYI – Fields of the class are meant to be accessed within the class (As shown above). So having a static field does not mean that they are accessible outside. So  cannot be accessed from Main unless its moved to global Application scope.  The const keyword however can be used to create a method-local constant definition. | |

Static Methods

|  |  |
| --- | --- |
| Static methods cannot access non-static class level members and do not have a 'this' pointer. Instance methods can access those members, but must be called through an instantiated object. This adds another level of indirection.  The example on the side shows an instance class with a static method and how to access such methods directly (without the instance).  The output is below:    Public Static methods use-case: Utility classes often contain public static methods and that is available to every caller.  Internal Static: to describe a method that must only be called in the same assembly as the method implementation.  Private static: methods are also useful as internal logic repositories for the class state. |  |

Static Classes

* Static classes are loaded automatically by the .NET Framework common language runtime (CLR) when the program or namespace containing the class is loaded.
* It is not possible to create instances of a static class using the new keyword.
* A class can be declared static, indicating that it contains only static members.
  + If a class is declared as static then the variables and methods should compulsorily be declared as static.
* They cannot contain Instance Constructors (still possible to declare a static constructor to assign initial values or set up some static state)
* They are sealed and therefore cannot be inherited

*Usage of Static Classes*

Use a static class to contain methods that are not associated with a particular object. For example, it is a common requirement to create a set of methods that do not act on instance data and are not associated to a specific object in your code. You could use a static class to hold those methods.

Creating a static class is therefore much the same as creating a class that contains only static members and a private constructor. A private constructor prevents the class from being instantiated.

The advantage of using a static class is that the compiler can check to make sure that no instance members are accidentally added. The compiler will guarantee that instances of this class cannot be created.

|  |
| --- |
| Can you create static methods without a static class? Why would create static methods?  Yes – you can    A static method, field, property, or event is callable on a class even when no instance of the class has been created. |
| Can we use "this" keyword in static classes ?  Because this refers to the object instance. There is no object instance in a call of a static method. |

Difference between const, readonly and static

Constants

* A constant member is defined at compile time and cannot be changed at runtime.
* Constants are declared as a field, using the const keyword and must be initialized as they are declared.



PI cannot be changed in the application anywhere else in the code as this will cause a compiler error.

* Constants must be a value type (sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, or bool), an enumeration, a string literal, or a reference to null.
* Constants are accessed as if they were static fields, although they cannot use the static keyword.
* Constants can be marked as public, private, protected, internal, or protected internal.

Read-Only

* A read only member is like a constant in that it represents an unchanging value. The difference is that a read-only member can be initialized at runtime, in a constructor as well being able to be initialized as they are declared.

|  |  |
| --- | --- |
|  |  |

* Because a readonly field can be initialized either at the declaration or in a constructor, readonly fields can have different values depending on the constructor used.
* readonly members are not implicitly static, and therefore the static keyword can be applied to a readonly field explicitly if required.
* A readonly member can hold a complex object by using the new keyword at initialization.

Static

* Use of the static modifier to declare a static member, means that the member is no longer tied to a specific object. This means that the member can be accessed without creating an instance of the class.
* Only one copy of static fields and events exists, and static methods and properties can only access static fields and static events.

  
To access a static class member, use the name of the class instead of a variable name to specify the location of the member. For example:  


* The static modifier can be used with classes, fields, methods, properties, operators, events and constructors, but cannot be used with indexers, destructors, or types other than classes.
* static members are initialized before the static member is accessed for the first time, and before the static constructor, if any is called.

**A readonly field can also be used for runtime constants as in the following example:**



Indexer

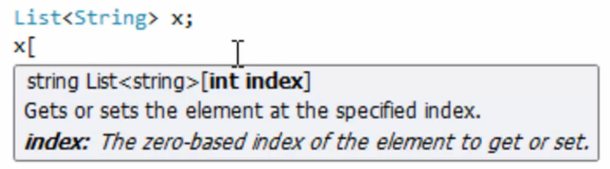
Indexer helps us to simply the way we access the collection from a class.

* Indexer are properties
* They are defined using this keyword
* Since Indexer are properties – we need to define set and get for them
* Simplified sugar syntax

The following is the syntax of an indexer declaration.

|  |  |
| --- | --- |
| <Return type> this [arguments]  {  get  {  }  set  {  } } |  |

* All indexers should accept at least one parameter.
* Indexers cannot be static. This is because static methods do not have access to ‘this’. The ‘this’ keyword indicates an instance of the current class.
* Indexers are heavily used inside .Net Framework:



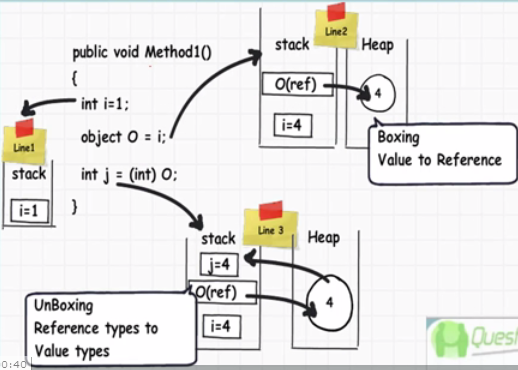
Difference between Control.BeginInvoke and Delegate.BeginInvoke

* Delegate.Invoke: Executes synchronously, on the same thread.
* Delegate.BeginInvoke: Executes asynchronously, on a threadpool thread.
* Control.Invoke: Executes on the UI thread, but calling thread waits for completion before continuing.
* Control.BeginInvoke: Executes on the UI thread, and calling thread doesn’t wait for completion. You should use this approach when you want to update a value on the UI itself (when finished processing).

Control.BeginInvoke and Control.Invoke are used to execute a delegate on the UI thread associated with the control that you call them on. The difference between the two is that BeginInvoke is asynchronous (fire and forget) whereas Invoke blocks until the delegate has been executed.

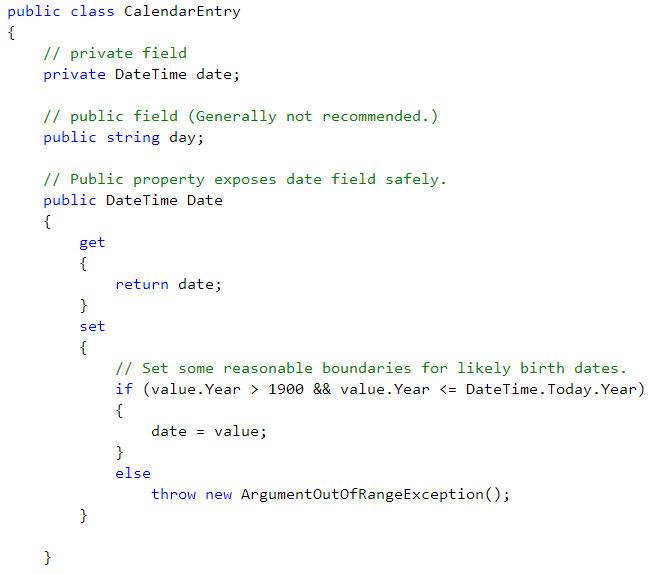
Boxing and UnBoxing

Boxing is the process of converting a value type to the type object or to any interface type implemented by this value type. When the CLR boxes a value type, it wraps the value inside a System.Object and stores it on the managed heap.   
  
Unboxing extracts the value type from the object. Boxing is implicit; unboxing is explicit. The concept of boxing and unboxing underlies the C# unified view of the type system, in which a value of any type can be treated as an object.  

Fields

* A field is a variable of any type that is declared directly in a class or struct.
* A class or struct may have instance fields or static fields or both.
* Generally, you should use fields only for variables that have private or protected accessibility.
* Data that your class exposes to client code should be provided through methods, properties and indexers. A private field that stores the data exposed by a public property is called a backing store or backing field.



Properties

Property acts as a cross link between the field and the method . Actually it behaves as a field. We can retrieve and store data from the field using property.

The compiler automatically translates the field like property into a call like special method called as 'accessor". In property there are two accessor and that are used to save value and retrieve value from the field. The two properties are 'get' and 'set'.

The get property is used to retrieve a value from the field and the set property is used to assign a value to a field .

Depending on their use properties are categorized into three types:

* ReadWrite Property :- When both get and set properties are present it is called as ReadWrite Property.
* ReadOnly Property :- When there is only get accessor it is called as ReadOnly Property.
* WriteOnly Property :- When there is only set accessor, it is called as WriteOnly Property.

Parse & TryParse

**Parse**

|  |  |
| --- | --- |
| * The int.Parse method converts strings into ints. * It throws exceptions on invalid input, which can be slow if they are common. * It does not contain any internal null checks. |  |

**TryParse**

|  |  |
| --- | --- |
| * TryParse is a good method if the string you are converting to an integer is not always numeric. * The TryParse method returns a boolean to denote whether the conversion has been successful or not, and returns the converted value through an out parameter. * The method returns true if it succeeds, and false if it doesn't. |  |

When to use what

* If your input contains non-numeric or invalid characters, use int.TryParse.
* If your input is valid and all numeric, and performance is not extremely critical, use int.Parse.
* If your input is guaranteed to be valid, use custom code if performance is critical. Otherwise, use int.Parse.
* If your input has many digits or decimal places use double.Parse or double.TryParse.

Pass By Value and Pass By Reference

Pass By Value:

* All method parameters are passed by value unless you specifically see ref or out.

Pass By Reference:

* Use the ref keyword while passing the value
* Special case: Arrays
  + Arrays are reference types. This means that you're passing a reference by value.
  + Thus, modifications made to array inside the function are actually applied to the int[] object to which array refers. And so those modifications are visible to all references that refer to that same object. And that includes the reference that the caller holds.
  + If it's necessary to change WHAT array a passed-in array variable points to (e.g. to change the size of the array), the variable must be passed by reference.

Additional

There are four different kinds of parameters in C#:

1. value parameters (the default)
2. reference parameters (which use the ref modifier)
3. output parameters (which use the out modifier)
4. parameter arrays (which use the params modifier).

You can use any of them with both value and reference types. When you hear the words "reference" or "value" used (or use them yourself) you should be very clear in your own mind whether you mean that a parameter is a reference or value parameter, or whether you mean that the type involved is a reference or value type.

Ref and Out keywords

**Ref & Out**

|  |  |  |
| --- | --- | --- |
| Num | Ref | Out |
| 1 | Ref means in/out which means the value can be read in the callee and then changed. | Out means out only |
| 2 | Ref has to be initialized in the caller function itself before it can be passed to a callee function | A variable you pass as an out parameter doesn't need to be initialized, and the method using the out parameter has to set it to something. |
|  | | |
| 3 | Ref parameters are for data that might be modified | out parameters are for data that's an additional output for the function (eg int.TryParse) that are already using the return value for something. |
| 4 | Properties cannot be passed via out or ref. (Under the hood they are function calls) | |

Examples:

|  |  |
| --- | --- |
| Ref Example | Ref and Out Example (Highlighting differences) |
|  |  |

i++ and ++i

|  |  |
| --- | --- |
|  | What is the outcome of the following code? What is the difference between ++i and i++? How come "i" is 1 after execution?    The result of the i++ expression is the value of i before being incremented. After it is incremented you assign that value to i. Basically i = i++ is equivalent to the following code:    The prefix version works because the result of ++j is the value of j after being incremented. That value is assigned to j and everything works fine. |

Need to write the 3 steps which happen internally as part of i++

Characteristics of Recursion

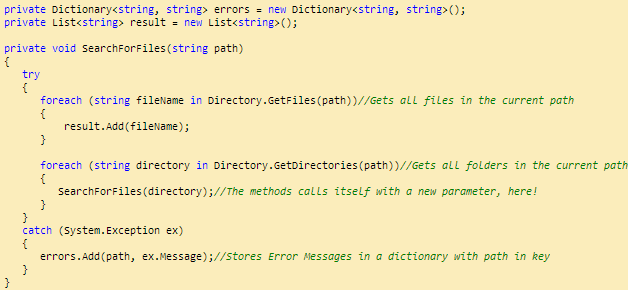
* A Recursive usually, has the two specifications:
  + Recursive method calls itself so many times until being satisfied.
  + Recursive method has parameter(s) and calls itself with new parameter values.
* You don’t use a for loop to simulate recursion rather the method should be called itself (either as part of return or some other mechanism)
* Recursive algorithms are often used for complex searching and simulation.
* Every recursive method sequence must be somehow terminated. Often the first part of the recursive method will have a branch that tests for a condition being met. In this way, the recursive methods continue until the result is attained.

Disadvantages

* Avoid using Recursive when the performance is a very-very important critical subject.

Use-Cases of Recursion

**Use-Case 1**: Finding Files within Directory



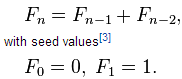
The method seems not to have any satisfy condition because it will be satisfied in each directory automatically, if it iterates all files and doesn't find any subfolder there.

**Use-Case 2**: Composite Design Pattern

**Use-Case 3**: The Fibonacci Series

Definition

* The Fibonacci Sequence is the series of numbers: 
* The next number is found by adding up the two numbers before it.
* By definition, the first two numbers in the Fibonacci sequence are 0 and 1, and each subsequent number is the sum of the previous two.
* In mathematical terms, the sequence Fn of Fibonacci numbers is defined by:



|  |  |
| --- | --- |
| With Recursion | Without Recursion |
|  |  |

**Use-Case 4**: Inner Exception

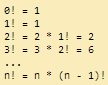
Recursive methods are useful for getting the last innerException:



Why the last innerException?! That's beside the point. The subject of our talk is, if you want to get the last innerException, you can count on Recursive method.

**Use-Case 5**: Factorial

We know that the factorial (!) of a positive integer number is the product of all positive integers less than or equal to the number.



|  |  |
| --- | --- |
| With Recursion | Without Recursion |
|  |  |

Application Domains

Definition - Application domains are an important innovation in .NET and are designed to ease the overhead involved when running applications that need to be isolated from each other but that also need to be able to communicate with each other.

Application domains are designed as a way of separating components without resulting in the performance problems associated with passing data between processes. The idea is that any one process is divided into a number of application domains. The ability to run multiple applications within a single process dramatically increases server scalability.

Example - The classic example of this is a web server application, which may be simultaneously responding to a number of browser requests. It will, therefore, probably have a number of instances of the component responsible for servicing those requests running simultaneously.

|  |  |
| --- | --- |
| **History** | **Current** |
| Up until now, the only means of isolating code has been through processes. When you start a new application, it runs within the context of a process. Windows isolates processes from each other through address spaces. Each process gets a different mapping, with no overlap between the actual physical memories that the blocks of virtual address space map | In the current world – with the introduction of app domain - Each application domain roughly corresponds to a single application, and each thread of execution will be running in a particular app domain. |

Reasons for App Domain

* Although processes are great for security reasons, their big disadvantage is in the area of performance. Often, a number of processes will actually be working together, and therefore need to communicate with each other.
* Few Classic Example –
* A process calls up a COM component, which is an executable and therefore is required to run in its own process.
* The same thing happens in COM when surrogates are used.
* Because processes cannot share any memory, a complex marshaling process must be used to copy data between the processes. This results in a very significant performance hit.
* If you need components to work together and do not want that performance hit, you must use DLL-based components and have everything running in the same address space — with the associated risk that a badly behaved component will bring everything else down.

With App Domain

* If different executables are running in the same process space, then they are clearly able to easily share data, because, theoretically, they can directly see each other's data. However, although this is possible in principle, the CLR makes sure that this does not happen in practice by inspecting the code for each running application to ensure that the code cannot stray outside of its own data areas.
* It is usually possible to do this because of the strong type safety of the IL. In most cases, unless code is using unsafe features such as pointers, the data types it is using will ensure that memory is not accessed inappropriately.
* If a running application does need to communicate or share data with other applications running in different application domains, it must do so by calling on .NET's remoting services.
* Code that has been verified to check that it cannot access data outside its application domain (other than through the explicit remoting mechanism) is said to be memory type safe. Such code can safely be run alongside other type-safe code in different application domains within the same process.

Misc Notes on App Domain

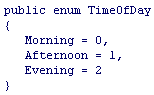
* When an application is started, the CLR creates a default application domain for the assembly with the entry point to start executing. From that point on, application domains are created on demand when the application or a library requests to do so.
* A good example of such a scenario is the use of add-ins in a larger application.
* Maintaining separate processes for every website is far too heavyweight, so historically web servers like Microsoft’s Internet Information Services (IIS) have had ways to run multiple websites in the same process. With the advent of ASP.NET, application domains can be leveraged to achieve this goal. Since IIS 7, the web server itself knows a good deal about managed code, and application pools have become the unit of isolation in IIS. As you might expect, those pools piggyback on application domains under the covers.

Enumeration

An enumeration is a user-defined integer type. When you declare an enumeration, you specify a set of acceptable values that instances of that enumeration can contain. Not only that, but you can give the values user-friendly names. If, somewhere in your code, you attempt to assign a value that is not in the acceptable set of values to an instance of that enumeration, the compiler will flag an error.

Advantages

* They make the code easier to maintain by helping to ensure that your variables are assigned only legitimate, anticipated values.
* They make the code clearer by allowing you to refer to integer values by descriptive names rather than by obscure "magic" numbers.
* Enumerations make your code easier to type, too. When you go to assign a value to an instance of an enumerated type, the Visual Studio .NET IDE will, through IntelliSense, pop up a list box of acceptable values



The real power of enums in C# is that behind the scenes they are instantiated as structs derived from the base class, System.Enum. This means it is possible to call methods against them to perform some useful tasks. Note that because of the way the .NET Framework is implemented there is no performance loss associated with treating the enums syntactically as structs. In practice, after your code is compiled, enums will exist as primitive types, just like int and float.

Yield Keyword

* Custom iteration without temp collection
* Stateful iteration

Deep Copy versus Shallow Copy

<http://stackoverflow.com/questions/16955883/why-is-the-non-generic-stack-class-implemented-as-a-circular-buffer-and-what?rq=1>

Control

The term control is used to speak generally of elements that derive from the System.Windows.Controls.Control base class. Controls have a rich set of functionality that includes features such as mouse and keyboard input, data binding, layout, styles and animation

Initialization of variable  
C# has 2 methods for ensuring that the variables are initialized before use:

* Variables that are fields in a class or struct, if not initialized explicitly are by default zeroed out when they are created
* Variables that are local to a method must be explicitly initialized in the code prior to any statements in which their values are used.

Constructors

What re 2 types of constructors

C# supports two types of constructor:

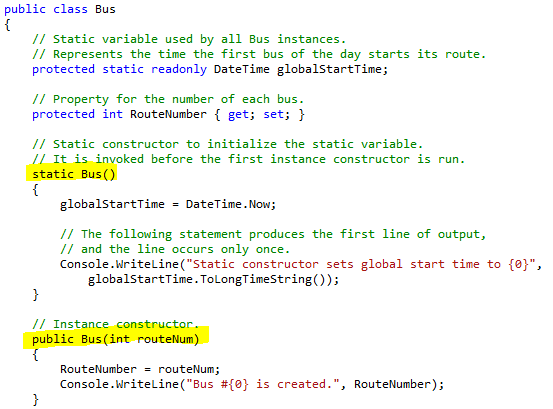
* A class constructor (static constructor) and
* An instance constructor (non-static constructor).

How do they differ

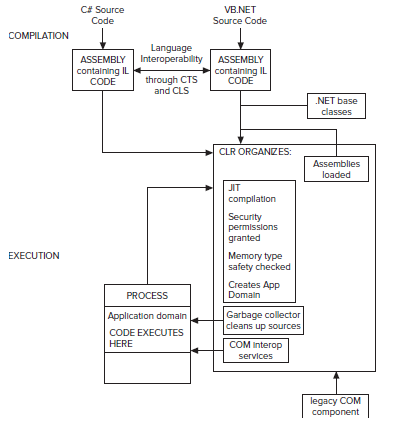
A static constructor is used to initialize any static data, or to perform a particular action that needs to be performed once only. It is called automatically before the first instance is created or any static members are referenced.

Static constructors have the following properties:

* A static constructor does not take access modifiers or have parameters.
* A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.
* A static constructor cannot be called directly.
* The user has no control on when the static constructor is executed in the program.
* A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.
* Static constructors are also useful when creating wrapper classes for unmanaged code, when the constructor can call the LoadLibrary method.
* If a static constructor throws an exception, the runtime will not invoke it a second time, and the type will remain uninitialized for the lifetime of the application domain in which your program is running.



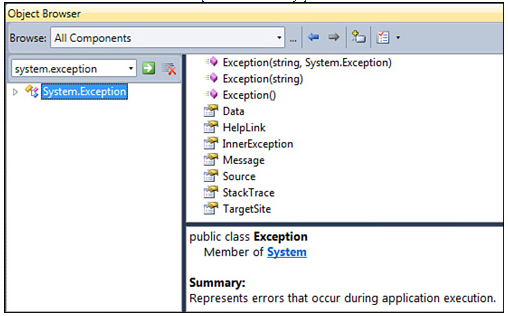
Net Architecture

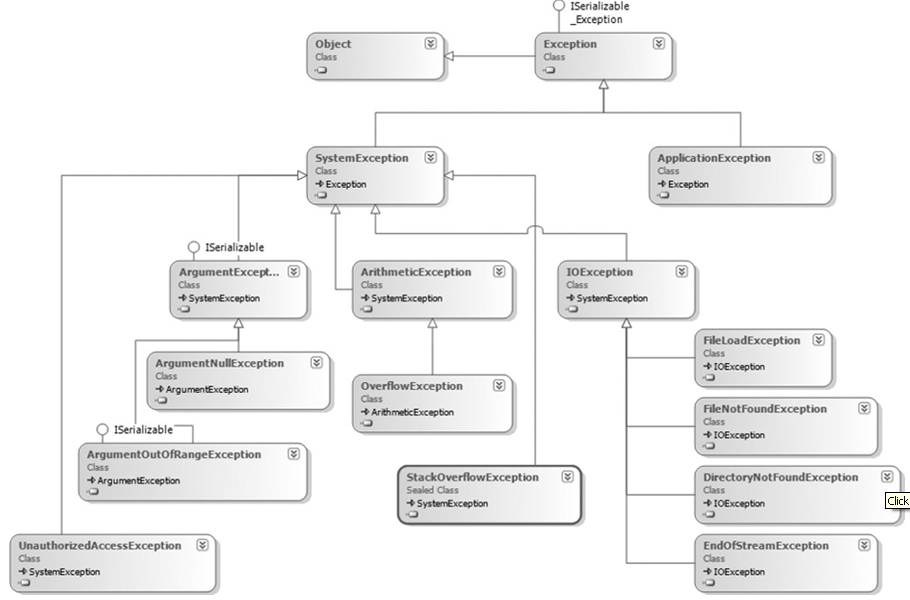


Exception Handling

Exceptions provide a means for detailed error propagation that doesn’t interfere with the method signatures (highly reduced need for the use of output parameters) and allows for a structured way to catch errors and handle them.

Exceptions are objects.





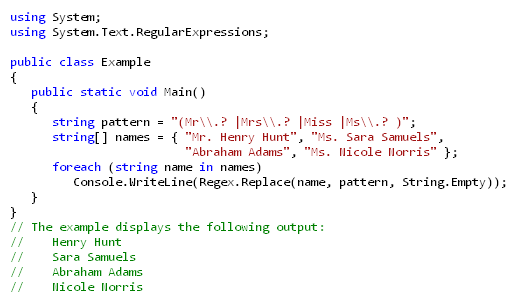
Regular Expressions

* Validate
* Parse
* Split
* Replace

Regular expressions provide a powerful, flexible, and efficient method for processing text. The extensive pattern-matching notation of regular expressions enables you to quickly parse large amounts of text to find specific character patterns; to validate text to ensure that it matches a predefined pattern (such as an e-mail address); to extract, edit, replace, or delete text substrings; and to add the extracted strings to a collection in order to generate a report. For many applications that deal with strings or that parse large blocks of text, regular expressions are an indispensable tool.

The centerpiece of text processing with regular expressions is the regular expression engine, which is represented by the System.Text.RegularExpressions.Regex object in the .NET Framework. At a minimum, processing text using regular expressions requires that the regular expression engine be provided with the following two items of information:

* The regular expression pattern to identify in the text.
* The text to parse for the regular expression pattern



Regex vs. String Methods

* The System.String class includes several search and comparison methods that you can use to perform pattern matching with text.
  + For example, the String.Contains, String.EndsWith, and String.StartsWith methods determine whether a string instance contains a **specified** substring; and the String.IndexOf, String.IndexOfAny, String.LastIndexOf, and String.LastIndexOfAny methods return the starting position of a specified substring in a string.
  + Use the methods of the System.String class when you are **searching for a specific string**. Use the Regex class when you are searching for a **specific pattern** in a string.

Static vs. Instance Methods

After you define a regular expression pattern, you can provide it to the regular expression engine in either of two ways:

* By instantiating a Regex object that represents the regular expression. To do this, you pass the regular expression pattern to a Regex constructor. A Regex object is immutable; when you instantiate a Regex object with a regular expression, that object's regular expression cannot be changed.
* By supplying both the regular expression and the text to search to a static Regex method. This enables you to use a regular expression without explicitly creating a Regex object.

Method for Regex Algorithm

* Always move from left to right
* Exits when first match is found
* Greedy – Reg expressions will try to match the longest length they can

Backtracking

* Big Performance Impact

Interview Questions –

<http://venkatcsharpinterview.blogspot.co.uk/2011/>

<http://tutorials.csharp-online.net/Visual_CSharp_FAQ>

<http://www.dailyfreecode.com/InterviewFAQ/difference-between-object-equals-130.aspx>

<http://www.netsqlinterviewquestions.com/interview_questions/2_-net-interview-questions.aspx?page=0>

<http://www.geekinterview.com/Interview-Questions/Microsoft/C-Sharp>

<http://www.dotnetuncle.com/csharp/csharp_questions.aspx>

<http://dotnetvideos.net/InterviewQandA/tabid/452/Default.aspx?utm_source=idealprogrammer.com&utm_medium=banner&utm_campaign=idealprogrammer>

<http://www.c-sharpcorner.com/interviews/eurl.axd/f21a470439a1f54699ce06fde16a4dc6/>

<http://www.dotnetfunda.com/interview/showcatquestion.aspx?category=32>

<http://joel.inpointform.net/software-development/net-interview-questions-framework-core/>

<http://joel.inpointform.net/software-development/job-interview-preparation-for-senior-software-developers/>

http://oop.megasolutions.net/

<http://www.a2zdotnet.com/Interview.aspx?id=5>

<http://www.freejobsreference.com/dotnet-interview-questions-answers/csharp.aspx>

<http://www.freejobsreference.com/dotnet-interview-questions-answers/netthreefive.aspx>

<http://www.crazyengico.com/QuestionList.aspx?qcatid=2>

<http://dotnetanalyst.com/FAQs/CSharp.aspx>

<http://www.treeknox.com/technical-questions/dotnet/csharp.net/>

<http://www.dotnetfunda.com/interview/exclusive/showcatquestion.aspx?category=32>

<http://www.mindstick.com/Interviewer/QuestionPage.aspx?topicid=1&topic=C%23>

<http://www.questpond.com/>

<http://beyondrelational.com/modules/2/blogs/61/posts/11219/object-oriented-programming-concepts-interview-questions.aspx>

Related Questions –

ILDASM - <http://beyondrelational.com/modules/2/blogs/61/posts/11246/vs2010-tip-how-to-launch-ildasm-from-vs2010-ide.aspx>

Identify a class in software design - <http://beyondrelational.com/modules/2/blogs/61/posts/11233/how-to-identify-a-class-in-software-design.aspx>

Parallel Debugging in VS 2010 -

<http://beyondrelational.com/modules/2/blogs/61/posts/11208/vs2010-parallel-tasks-debugging-enhancement-in-vs2010.aspx>  
<http://beyondrelational.com/modules/2/blogs/61/posts/11209/vs2010-parallel-stacks-debugging-enhancement-in-vs2010.aspx>

SQL Server - <http://beyondrelational.com/modules/12/tutorials/685/getting-started-with-ssms.aspx?tab=tutorials&ts=46&bs=57>

<http://www.freejobsreference.com/dotnet-interview-questions-answers/sqlserver.aspx>

LINQ - <http://anildroisys.blogspot.com/2012/01/linq.html>

Links to read –

<http://msdn.microsoft.com/en-us/magazine/cc188707.aspx>

<http://idealprogrammer.com/>

<http://dotnetvideos.net/PremiumVideos/SOLID/tabid/1046/Default.aspx?utm_source=idealprogrammer.com&utm_medium=banner&utm_campaign=idealprogrammer>

Done

<http://www.dotnetfunda.com/interview/showcatquestion.aspx?category=34>

C# Questions

|  |  |
| --- | --- |
| Framework Concepts | |
|  | What is the CLR? What tasks is it responsible for?  Common Language Runtime – This is a virtual machine (or execution engine depending on who is defining it) that runs generic code meant for multiple hardware configurations |
|  | What is CLI?  Common Language Infrastructure – a specification that .net languages are built on. The CLR is an implementation of the CLI |
|  | What is CIL?  Common Intermediate Language – .net languages such as C# get compiled to CIL bytecode.  This format is understood by the CLR |
|  | **JIT Compiler –** Why is it there – why does C++ not need one (or have one)  <http://kalitinterviewquestions.blogspot.com/search?updated-max=2009-10-20T23:54:00-07:00&max-results=7&start=10&by-date=false> |
|  | What does the following tools do: ILASM, ILDASM,SN,CORFLAGS |
|  | What does CLS Compliant mean? |
|  | What is the DLR? |
|  | What is JIT?  Just In Time compiler - CIL is compiled at runtime as needed into native code  It executes code by using a hybrid of interpreted and ahead-of-time approach. Code is first interpreted and then the compilied commands are cached for later use. |
|  | How can JIT code be faster than ahead of time compiled code?  The JIT code is interpreted for an computer’s specific hardware configuration whereas ahead-of-time compilied code is written for a set of computers meeting a more general specification. This means the JIT can take advantage of very specific hardware features. |
|  | Explain the path from C# source to native code  C# gets compiled to CIL  CIL is then executed by the CLR. It does this using a JIT compiler which converts the CIL into native code |
|  |  |
| .Net Basics | |
|  | What is a Concrete Object?  A concrete object is any object created with the “new” keyword |
|  | What does the term immutable mean?  The data value may not be changed. Note: The variable value may be changed, but the original immutable data value was discarded and a new data value was created in memory |
|  | What does it mean for an object to be immutable? |
|  | How would you create an immutable class |
|  | Are private class-level variables inherited?  Yes, but they are not accessible. Although they are not visible or accessible via the class interface, they are inherited. |
|  | What is the use of param keyword in C#?  In C# param parameter allows us to create a method that may be sent to a set of identically typed arguments as a single logical parameter. |
|  | Can you allow a class to be inherited, but prevent the method from being over-ridden?  Yes. Just leave the class public and make the method sealed. |
|  | What is the size of a .Net Integer? |
|  | What is late binding and early binding? What is the benefits and disadvantages of using them?  Need to Research |
|  | Can you store a value type on heap - answer is yes (googly)  answer is customer c = new (){id = 1, name="Amit"}  Value type can be stored in a heap if they are part of an object and are stored in inline heap |
|  | What is upcasting and downcasting?  Link to Review - <http://www.c-sharpcorner.com/UploadFile/pcurnow/polymorphcasting06222007131659PM/polymorphcasting.aspx> |
|  | How do you pass arguments to the main () method?  To pass command line arguments to the program, the parameter is string array traditionally called args (although the C# will accept any name).    When you run the compiled executable, you can pass in arguments after the name of the program, as shown in this example: |
|  |  |
|  | Namespace/Alias   * Namespaces are the way that .Net avoids name clashes between classes. * A namespace is a grouping of data types hence effectively prefixes all data within the namespace. * It is possible to nest the namespace within each other – most general purpose .Net base classes are in the namespace System. The base class for Array is in this namespace, so its full name is System.Array |
|  | Can you nest namespaces within each other |
|  | What are 2 types of constructors   * 1. How do they differ   2. Give a use-case of where static constructors are used   Static constructor is used to initialize static data members as soon as the class is referenced first time, whereas an instance constructor is used to create an instance of that class with keyword. A static constructor does not take access modifiers or have parameters and can't access any non-static data member of a class. |
|  | Is it possible to inherit a class that has only private constructor?  When you instantiate a derived class, constructor of the base class is called before constructor of the derived class. If base class constructor is private it is not accessible from derived class and an error is thrown. |
|  | What is an attribute? Give me some possible uses. |
|  | How are the attributes specified in C#  - enclosed with in [ ]  - enclosed with in ( )  - enclosed with in { } They are enclosed with in []. |
|  | What are CustomAttributes |
|  | Can be Serailizable attribute be inherited? |
|  | How to declare a property in a class?  int m\_PersonID = 0;  public int PersonID  {  get { return m\_PersonID; }  set { m\_PersonID = value; }  } |
|  | How to declare a property in an Interface?  DateTime DateOfBirth { get;set;}  int Age { get;set;}  string FirstName { get;set;}  As this is an Interface, so no implementation required only definition of properties are required. Implementation of these properties will be written into the class inherting this interface. |
|  | What first action compiler will take on detection of iterator?  As soon as compiler will detect iterator, it will automatically generate current, MoveNext and Disposemethods of the IEnumerator or IEnumerator(T) type. |
|  | What is default access specifier of a class in C#?   * A class has default modifiers as Internal. * A methods, fields, and properties has default access modifier as "Private" if no modifier is specified. * An interface has default modifier as public * A struct has default modifier as Internal * An enum has default modifier as public |
|  | Can you get reference of a struc type |
|  | What is the use of ?? operator in C#?  This operator allows you to assign a value to a nullable type if the retrieved value is in fact null. |
|  | Where is the “value types” stored if you put them in ArrayList or List class? How? |
|  | Why do you need extension methods. Can you just use plain inheritance? |
|  | Why in Lamda expression (x => x.FirstName). How does it know that x is of what type? |
|  | What is the implicit name of the parameter passed into a property’s ’set’ method? |
|  | “As Keyword”  The as operator is like a cast operation. However, if the conversion isn't possible, as returns null instead of raising an exception. Hence it is used a null check mechanism  <http://en.csharp-online.net/Test_for_an_interface_implementation> |
|  | Constraints on Type Parameters  When you define a generic class, you can apply restrictions to the kinds of types that client code can use for type arguments when it instantiates your class. If client code tries to instantiate your class by using a type that is not allowed by a constraint, the result is a compile-time error. These restrictions are called constraints. Constraints are specified by using the where contextual keyword.  where T : class - The type argument must be a reference type; this applies also to any class, interface, delegate, or array type.  Why use Constraints  If you want to examine an item in a generic list to determine whether it is valid or to compare it to some other item, the compiler must have some guarantee that the operator or method it has to call will be supported by any type argument that might be specified by client code.  Example: The constraint enables the generic class to use the Employee.Name property because all items of type T are guaranteed to be either an Employee object or an object that inherits from Employee. |
|  | Difference between deep copy and shallow copy  Shallow copies duplicate as little as possible. A shallow copy of a collection is a copy of the collection structure, not the elements. With a shallow copy, two collections now share the individual elements.  For value type --> a bit-by-bit copy of the field is performed For Reference type --> the reference is copied but the referred object is not; therefore the original object and its clone refer to the same object.  Deep copies duplicate everything. A deep copy of a collection is two collections with all of the elements in the original collection duplicated.  For Value type --> a bit-by-bit copy of the field is performed.  For Reference type --> a new copy of the referred object is performed. |

|  |  |
| --- | --- |
| Object Oriented Concepts | |
|  | Which of the following statements is not true for interfaces - Interface definitions does not have implementation - Interfaces must be declared as public - Interfaces can be instantiated - Interface does not have constructors Interfaces can only be implemented by inheriting and they cannot be instantiated. |
|  | Why C# does not support multiple class inheritance  Why multiple Inheritance is not possible in C#? (Please do not answer like this-It is possible through Interfaces.)  C# does not support multiple class inheritance because of the diamond problem that is associated, with multiple class inheritance. Let us understand the diamond problem of multiple class inheritance with an example.    As shown in the image above:  1. I have 2 classes - ClassB and ClassC  2. Both of these classes inherit from ClassA  3. Now, we have another class, ClassD which inherits from both ClassB and ClassC  So, if a method in ClassD calls a method defined in ClassA and ClassD has not overriden the invoked method. But both ClassB and ClassC have overridden the same method differently. Now, the ambiguity is, from which class does, ClassD inherit the invoked method: ClassB, or ClassC?  In order not to have these problems, C# does not support multiple class inheritance.  So to avoid this problem we use Interface..... what is interface ... means in interface we just declare a function ...ok and in the derived class we give the definition as per the requirement...means function should be abstract ... and in interface all function[method] should abstract.  So essentially it is to avoid name collision.  Multiple Inheritance can have many classes and methods , and its quite possible that they can have save name " say a save function in two different classes name can be Save() but implementation can be different". The derived class will have problem accessing the correct function. |
|  | What is the purpose of interfaces? I can always force developer to implement the method? (Answer in the direction of polymorphism) |
|  | When should I use the IEquatable interface? |
|  | When should I use the IStructualEquatable? |
|  | What are the 3 places where you use the “new” keyword |
|  | What happens when you have a method with the same name in base and derived class and not declared as virtual/override – what would happen? |
|  | Is Polymorphism possible through Interface? Are virtual/override keywords valid in this scenario |
|  | When you have a derived class inherited from base class - what constructors are called |
|  | Can you have a abstract methods without abstract class? |
|  | can we add events and properties in interfaces ? |
|  | Can you can specify values for variables in interfaces in c#? (In Java you can)  No. I tried this in VS.net and it gave me the following error: D:\VisualStudio\…\Interface.cs(10): Interfaces cannot contain fields |
|  | Can we overload methods by specifying different return types?  No |
|  | How do you inherit from a class in C#?  Place a colon and then the name of the base class. |
|  | Can you declare the override method static while the original method is non-static? No, you can’t, the signature of the virtual method must remain the same, only the keyword virtual is changed to keyword override. |
|  | Can you override private virtual methods? No, moreover, you cannot access private methods in inherited classes, have to be protected in the base class to allow any sort of access.  The .Net compiler does not accept a virtual private method which makes sense since a virtual private method makes no sense. |
|  | Can you prevent your class from being inherited and becoming a base class for some other classes? Yes, that’s what keyword sealed in the class definition is for. The developer trying to derive from your class will get a message: cannot inherit from Sealed class WhateverBaseClassName. It’s the same concept as final class in Java. |
|  | Can you allow class to be inherited, but prevent the method from being over-ridden?  Yes, just leave the class public and make the method sealed. |
|  | When do you absolutely have to declare a class as abstract (as opposed to free-willed educated choice or decision based on UML diagram)?  When at least one of the methods in the class is abstract. When the class itself is inherited from an abstract class, but not all base abstract methods have been over-ridden. |
|  | Use of property in an interface:  An interface can have methods, properties, events and indexers. When a property in an interface is implemented in a class, the advantage is that, through that property, the class can give access to its private member varibles. |
|  | Deep class inheritance hierarchy – why it is bad?   * Fragile base classes (changes to base are a nightmare for the derived) * Increased coupling (with too many base classes comes tight coupling) * Encapsulation weakens * Testing issues (leaf level overridden methods can't just be tested to reproduce end-user behavior correctly always due to multiple chained calls here and there) * Maintenance (comes from strong coupling) |
|  | Why an instance can't be created for an abstract class?  An Instance of Abstract class cannot be created because an abstract class may (not compulsory) have some abstract members i.e. members whose declaration is provided and implementation is not and if the instance of this class is created, that abstract method would be invoked and it doesnt have implementation and this would create a problem. To avoid this kind of problem the compiler gives error if we try to create the instance of abstract class (which can also be said as incomplete class) |
|  |  |
|  |  |

|  |  |
| --- | --- |
| Delegates | |
|  | Can a Method level variable can be accessed by the Anonynomus Delegate? if you run Async Delegate what happens to variable when Method goes out of scope and Async delegate is still running? Where is the variable stored in that case? |
|  | How to get a return value from Async Delegates? |
|  | When to use delegates instead of interfaces  Both delegates and interfaces enable 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 be 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 that has 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 it use an interface?  Use a delegate in the following circumstances:  - An eventing design pattern is used.  - The caller has no need to 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  - It is desirable to encapsulate a static method.  Use an interface in the following circumstances:  - There is a group of related methods that may be called.  - A class only needs one implementation of the method.  - 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 the generic version, IComparable<T>. IComparable declares the CompareTo method, which returns an integer that specifies 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. Although 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 does not change at run time, a single-method interface is ideal. |
|  | What are method group conversions in context of delegates? |
|  | delegate inherited from multicast delegate |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| Garbage Collection | |
|  | Why is it called managed heap?  Automatic memory management. The process of automatic memory management involves the following tasks:   * Allocating memory * Releasing memory * Implementing finalizers |
|  | What are the application roots? Example?  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 live local variable/parameter object pointers on thread’s stack. * Any CPU registers containing pointers to objects in managed heap. * Pointers to the objects from Reachable 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. |
|  | What is Garbage Collector?   * The garbage collector is .NET's answer to memory management and in particular to the question of what to do about reclaiming memory that running applications ask for. * Garbage collection is **not deterministic**. In other words, you cannot guarantee when the garbage collector will be called; it will be called when the CLR decides that it is needed, though it is also possible to override this process and call up the garbage collector in your code. |
|  | Terminologies Involved:  Non Deterministic Cleanup  **Desrtuctor/Finalizer Concepts**   * Finalizable objects go through an additional step when the garbage collector detects they’re no longer reachable and subject to deallocation. Instead of deallocating the object during the collection cycle, the object is put in a queue (sometimes referred to as the freachable queue, for finalization-reachable queue). * As a rule of thumb, don’t implement a finalizer method unless you absolutely need it. Violating this rule will put additional stress on the garbage collector, which is strongly discouraged. * This is why empty destructors are evil – makes GC go through 2 step process and causes this to be very expensive   Disadvantages of calling Destructors   * Non-Deterministic * The implementation of a destructor delays the final removal of an object from memory.   Deterministic Cleanup   * Implementing **System.IDisposable interface** * Provides a deterministic mechanism for freeing unmanaged resources and avoids the garbage collector-related problems inherent with destructors. * The IDisposable interface declares a single method named Dispose(), which takes no parameters and returns void. * There is no performance benefit in implementing the Dispose method on types that use only managed resources (such as arrays) because they are automatically reclaimed by the garbage collector. * A Dispose method should call the SuppressFinalize method for the object it is disposing. The method tells the garbage collector that a class no longer needs to have its destructor called. * Implementing **using Method** * This helps in using a certain resource and want to leave the guaranteed cleanup.      * The using statement consists of two different parts. Between the parentheses is the resource-acquisition expression; it indicates the resource that will be used inside the block underneath and needs to be cleaned up no matter how the block is left. The block itself is simply the code that executes with the acquired resource in scope.   **Relationship between Finalizer and IDisposable**  The big question now is how to relate the IDisposable pattern and the use of finalizers with one another. If a user calls the Dispose method explicitly, there’s no need for the finalizer to run anymore because you’ve already got a chance to clean up resources onto which the object holds. On the other hand, if Dispose is never called but the object got unreachable, the finalizer is required to run and do the same cleanup as the Dispose method would have done if it were called properly by the user during the object’s lifetime. So we have the same cleanup logic in both places. Those observations give rise to the pattern shown above (Explanation below):  You can see from this code that there is a second protected overload of Dispose(), which takes one bool parameter — and this is the method that does all the cleaning up. Dispose (bool) is called by both the destructor and by IDisposable.Dispose(). The point of this approach is to ensure that all cleanup code is in one place.   * If a consumer calls IDisposable.Dispose(), that consumer is indicating that all managed and unmanaged resources associated with that object should be cleaned up. * If a destructor has been invoked, all resources still need to be cleaned up. However, in this case, you know that the destructor must have been called by the garbage collector and you should not attempt to access other managed objects because you can no longer be certain of their state. In this situation, the best you can do is clean up the known unmanaged resources and hope that any referenced managed objects also have destructors that will perform their own cleaning up.   The parameter passed to Dispose (bool) indicates whether Dispose (bool) has been invoked by the destructor or by IDisposable.Dispose() — Dispose (bool) should not be invoked from anywhere else in your code.  Dispose/Finalized Pattern   * Microsoft recommends that you implement both Dispose and Finalize when working with unmanaged resources. 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. * Finalize provides a backup to prevent resources from permanently leaking if the programmer fails to call Dispose.   **When to use what**   * As the user of an object, you always use Dispose. Finalize is for the GC. * As the implementer of a class, if you hold managed resources that ought to be disposed, you implement Dispose. If you hold native resources, you implement both Dispose and Finalize, and both call a common method that releases the native resources.   **Additional Comments**   * 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.   --------------------------------------------------------------------------------------------------------------   * IDispose can only free the resources and cannot free the memory consumed by the object itself. * Onus of calling Dispose method on the object still remains with the user. Just implementing IDisposable will not make sure that all resources are released properly. Dispose method will not be called automatically by framework except object is called with “using”. * Avoid throwing exceptions from within Dispose methods * Allow a Dispose method to be called more than once without throwing an exception. The method should do nothing after the first call. * 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. * It is better to consider implementing try..finally block inside the dispose method because if during the execution of dispose method any exception occurs there is no way left to free the resources.   **Additional Concepts**   * GC.Collect Concept * The System.GC class is a .NET class that represents the garbage collector, and the Collect() method forces a garbage collection to occur irrespective of whether it is needed. * The GC class is intended for rare situations in which you know that it's a good time to call the garbage collector; for example, if you have just de-referenced a large number of objects in your code. * Why is calling GC.Collect bad generally * In general, calling GC.Collect will always lead to a performance loss, never a gain. This will introduce an immediate delay in your code, while the GC runs. If you wait for the GC to collect on its own schedule (which will happen later), it will collect all of your memory then, plus more, so it will technically be more efficient. * I personally believe direct calls to GC.Collect are typically signs of a design flaw, or a programmer trying to optimize something they don't fully understand. There are almost no places where this is really necessary. * In which circumstances would you call GC.Collect   Consider calling GC.Collect() if some non-recurring event has just happened and this event is highly likely to have caused a lot of old objects to die.   1. If you have an application that has a lot of objects that are going into Generation 2 and then dying suddenly 2. A classic example of this is if you're writing a client application and you display a very large and complicated form that has a lot of data associated with it. Your user has just interacted with this form potentially creating some large objects... things like XML documents, or a large DataSet or two. When the form closes these objects are dead and so GC.Collect() will reclaim the memory associated with them. 3. Another reason to call it would be to force the delay to happen when you want it to happen, so there is no chance that there will be a delay from GC during a critical operation. I have some code that is time sensitive, and have wanted to make sure the GC would not collect during this (short) operation, since it can cause some hangs. Calling GC.Collect before the time critical code helps reduce the likelihood of a collection occuring during the operation, which could cause a hang that was bad. |
|  | Garbage Collection Modes  The garbage collector is self-tuning and can work in a wide variety of scenarios. The only option you can set is the type of garbage collection, based on the characteristics of the workload. The CLR provides the following types of garbage collection:  Garbage Collection Modes  Workstation GC  For all client workstations and stand-alone PCs. Default Setting in runtime  Server GC  Intended for server applications that need high throughput and scalability  Concurrent  Enables managed threads to continue operations during a garbage collection.  Non-Concurrent  Background GC  Starting with the .NET Framework 4, background garbage collection replaces concurrent garbage collection     * <gcServer> element  Attribute set to false (default) - CLR performs workstation garbage collection Attribute set to true - CLR performs server garbage collection * Workstation concurrent is the default * Gen 0 and Gen 1 are always non-concurrent because they are very quick and small (hence doesn’t have effect from a user perspective) * Enables managed threads to continue operations during a garbage collection on Gen 2   In the .NET Framework 4, it is supported only for workstation garbage collection. Starting with the .NET Framework 4.5, background garbage collection is available for both workstation and server garbage collection.  More Details - <http://msdn.microsoft.com/en-us/library/ee787088.aspx#concurrent_garbage_collection> |
|  | Do I have any control over the garbage collection algorithm?  A little. For example the System.GC class exposes a Collect method, which forces the garbage collector to collect all unreferenced objects immediately. |
|  | 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 |
|  | Is it true that objects don't always get destroyed immediately when the last reference goes away?  Yes. The garbage collector offers no guarantees about the time when an object will be destroyed and its memory reclaimed. |
|  | Object.Finalize  Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.  The Finalize method is used to perform cleanup operations on unmanaged resources held by the current object before the current object is destroyed. The method is protected and therefore is accessible only through this class or through a derived class.  This method is automatically called after an object becomes inaccessible, unless the object has been exempted from finalization by a call to GC.SuppressFinalize. During shutdown of an application domain, Finalize is automatically called on objects that are not exempt from finalization, even those that are still accessible. Finalize is automatically called only once on a given instance. |
|  | Can finalized objects be resurrected?  Yes. Finalize can take any action, including resurrecting an object (that is, making the object accessible again) after it has been cleaned up during garbage collection. However, the object can only be resurrected once; Finalize cannot be called on resurrected objects during garbage collection.  Use Case  The best use I can think of is a "recycling" pattern. Consider a Factory that produces expensive, practically immutable objects; for instance, objects instantiated by parsing a data file, or by reflecting an assembly, or deeply copying a "master" object graph. The results are unlikely to change each time you perform this expensive process. It is in your best interest to avoid instantiation from scratch; however, for some design reasons, the system must be able to create many instances (no singletons), and your consumers cannot know about the Factory so that they can "return" the object themselves; they may have the object injected, or be given a factory method delegate from which they obtain a reference. When the dependent class goes out of scope, normally the instance would as well. |
|  | Do finalizers always execute?  A finalizer may not run, for example, if:   * Another finalizer throws an exception. * Another finalizer takes more than 2 seconds. * All finalizers together take more than 40 seconds. * An AppDomain crashes or is unloaded (though you can circumvent this with a critical finalizer (CriticalFinalizerObject, SafeHandle or something like that) * No garbage collection occurs * The process crashes |
|  | What is a CriticalFinializer object?  Ensures that all finalization code in derived classes is marked as critical.  The CriticalFinalizerObject class does add some reliability to the Dispose/Finalize process. Like preventing Thread.Abort(). The reclaim of native reources will Always occur |
|  | Can you raise a event in finalizer  The problem here is that as long as that event handler is referencing the object, it will not be eligible for garbage collection, as least as long as that other object that owns the event is alive.  As such, putting anything in the destructor is pointless, as either:   * The event handler has already been removed, thus the object became eligible for garbage collection * The event handler is not removed, the owning object is not eligible for garbage collection, and thus the finalizer will never get called * Both objects are eligible for garbage collection, in which case you should not access that other object at all in the finalizer since you don't know its internal state   In short, don't do this. |
|  | What are the sizes of Generations in Garbage Collector  Gen 0 threshold – 256 K  Gen 1 threshold – 2 Mb  Gen 2 threshold – 10 Mb |
|  | What if you write an empty finalize method?  Evil – Empty Destructor |
|  | Why would someone want to implement IDisposable?  One common reason is to ensure efficient use of limited resources like database connections  Another use would be to wrap processes that have specific code that needs to be called at a start and finish point (ie, a database transaction block) |
|  | What is LOH? |
|  | When does memory get compacted? How can I prevent an object to move in memory? |
|  | How does .Net handle garbage collection for a thread that is currently executing unmanaged code? |
|  |  |
|  |  |

|  |  |
| --- | --- |
| Exception Handling | |
|  | What is an arithmetic overflow?  How can I be notified when this occurs? |
|  | What is the ThreadStack? How much space does it have by default? Explain what a StackOverFlowException is? |
|  | It is perfectly legitimate to throw exceptions from catch and finally blocks  a) True b) False  Catch and finally blocks are just like other code blocks, and you can write any legitimate C# code, including try, throw, catch and finally. So - it is legitimate to throw exception from finally block but it should be avoided as much as possible. |
|  | Why would I do throw in a single catch block? if the exception occurs, there will be implicit throw anyways? |
|  | How do debug code in production environment? Exception logging in windows/wpf applications? |
|  | how are exceptions expensive ? |
|  | what happens if you have an exception in the process of cleaning up COM objects ? |
|  | Exceptions – having multiple level try catch (at each function) is good or bad?  Need to find a good answer |
|  | What is better – throwing an exception or catching an exception  Need to find a good answer |
|  | Difference between try/finally and using blocks – which is better |
|  | What are the memory considerations when using recursion with many levels?  <http://joel.inpointform.net/software-development/explanation-of-stack-heap-and-recursion-causing-stack-overflow/> |
|  | What is the correct order for Catch clauses when handling different exception types?  Order from most general to most specific.  Order from most likely to least likely to occur.  Order from most specific to most general.  Order from least likely to most likely to occur. |
|  | Why should you close and dispose of resources in a Finally block instead of a Catch block?  It keeps you from having to repeat the operation in each Catch.  Finally blocks run whether or not an exception occurs.  The compiler throws an error if resources are not disposed of in the Finally  You cannot dispose of resources in a Catch block. |
|  | If there is no valid conversion between two types, what should you do when implementing the IConvertible interface?  Delete the ToType member that performs the conversion.  Throw an InvalidCastException.  Throw a new custom exception reporting the error.  Leave the member body empty. |
|  | What does assert() method do?  In debug compilation, assert takes in a Boolean condition as a parameter, and shows the error dialog if the condition is false. The program proceeds without any interruption if the condition is true. |
|  | A try block having 4 catch block will fire all catch block or not?  No, A try having more than one catch block will fire the first relevant catch block after that cursor will be moved to the finally block (if exists) leaving all remaining catch blocks. So in all cases only one catch block will fire. |
|  | Can we throw exception from catch block ?  Yes. The exceptions which cant be handled in the defined catch block are thrown to its caller. |
|  | Restrictions of yield in try-catch.  While using yield keyword, mainly two restrictions are observed. First is , we can’t use yield in finally. Second is , we can’t place yield keyword in the catch block if try contains more than one catch blocks. |
|  | Difference between throw and rethrow -reset stack trace – is provided by rethrow  Catch (Exception ex)  {  throw;  throw ex; (this is rethrow – will reset the stack trace)  } |
|  | Exception bubbling  <http://www.dotnetinterviewquestions.in/article_c-interview-questions:-what-is-exception-bubbling-in-c_117.html> |
|  |  |

|  |  |
| --- | --- |
| Misc | |
|  | How are Winforms events different from WPF events? |
|  | What classes/interfaces does .Net offer to enable object serialization? Which would you use? |
|  | How to bind a dataset and display conditional formatted rows based on the datavalue? |
|  | Write a function that would take a string value like "10.54" and convert it into integer? |
|  | Write a function that would take a string and return a boolean whether it is palindrome? |
|  | create a custom collection of Employee Class and use it in your UI layer to bind it to a datagrid ? |
|  |  |
|  | What are the different design patterns you have used? What is decorator/facade pattern? |
|  | write a psuedo code to implement singleton design pattern |
|  | Define Visitor pattern. Is is a valid pattern in C#? |
|  | Define Observer pattern |
|  | Define Flyweight pattern and how you would use it in a Security Market data service |
|  | Which design pattern is used in StreamReader class |
|  | Memory profiler – tools you may have used |
|  | How do you improve performance of your code |

Tower of hanovy

Programmer Matrix - <http://www.indiangeek.net/wp-content/uploads/Programmer%20competency%20matrix.htm>

Performance Profiling Links

Link 1 - <http://msdn.microsoft.com/en-us/library/ms182372.aspx>

Link 2 - <http://www.jetbrains.com/profiler/index.html>

Link 3 - <http://www.jetbrains.com/profiler/webhelp/Studying_Profiling_Results__Comparing_Profiling_Data.html>

Link 4 - <http://msdn.microsoft.com/en-us/magazine/cc337887.aspx>

Link 5 - <http://stackoverflow.com/questions/3927/what-are-some-good-net-profilers?rq=1>

Link 6 – WPF Performance Suite - <http://msdn.microsoft.com/en-us/library/aa969767.aspx>

Link 7 - <http://msdn.microsoft.com/en-us/magazine/ee336027.aspx>

Link 8 - <http://blogs.msdn.com/b/visualstudio/archive/2010/03/02/wpf-in-visual-studio-2010-part-2-performance-tuning.aspx>