C# Concepts

Delegates

A delegate is a named type that defines a particular kind of method. Just as a class definition lays out all the members for the given kind of object it defines, the delegate lays out the method signature for the kind of method it defines.

Based on this statement, a delegate is a function pointer and it defines what that function looks like.

A delegate can be seen as a placeholder for a/some method(s).

By defining a delegate, you are saying to the user of your class, "*Please feel free to assign, any method that matches this signature, to the delegate and it will be called each time my delegate is called*".

Typical use is of course events. All the OnEventX *delegate* to the methods the user defines.

Delegates are useful to offer to the **user** of your objects some ability to customize their behavior. Most of the time, you can use other ways to achieve the same purpose and I do not believe you can ever be **forced** to create delegates. It is just the easiest way in some situations to get the thing done.

Delegates is a type that hold references to methods with a particular parameter list and return type. Programmers can invoke the method through the delegate instance.

Syntax:

```
<access_modifier> delegate <return_data_type> delegate_name (parameters_list)
```

Where:

- The ***access_modifier*** specifies the accessbility of delegates.
- The ***return_data_type*** specifies the returning data type of output from delegates.
- The ***delegate_name*** specifies the name of delegate.
- The ***parameters_list*** specifies the inputs parameters list of delegates.

Sample:

```
public delegate int PerformCalculation(int x, int y);
```

Key point to remember

- Delegates are like C++ function pointers but are type safe.
- Delegates allow methods to be passed as parameters.
- A method must have the same return type as the delegate.
- Delegates can be used to define callback methods.
- Delegates are especially used for implementing events and the call-back methods.
- Delegates can be chained together; for example, multiple methods can be called on a single event.

Following example demonstrates declaration, instantiation, and use of a delegate that can be used to reference methods that take two integer parameters and returns an integer value.

```
namespace delegatesSample
{
   class Program
   {
      //Declaring a delegate
      public delegate int Calculator(int n1, int n2);

      static void Main(string[] args)
      {
         ////create delegate instances
         Calculator c1 = new Calculator(Add);

        //calling the methods using the delegate objects
         int result = c1(2, 3);

         Console.WriteLine(result);
         Console.Read();
      }


      public static int Add(int a1, int a2)
```

```
    {
        return a1 + a2;
    }
  }
}
```

When the above code is compiled and executed, it produces the following result:

```
5
```

There are three types of delegates that can be used in C#.

- Single delegate
- Multicast delegate
- Generic delegate

Single Delegate

Single delegate can be used to invoke a single method.

In following example, a delegate Calculator invokes a method ***Add()***.

```
namespace delegatesSample
{
   class Program
   {
      //Declaring a delegate
      public delegate int Calculator(int n1, int n2);
      static void Main(string[] args)
      {
         ////create delegate instances
         Calculator c1 = new Calculator(Add);
         //calling the methods using the delegate objects
         int result = c1(2, 3);
         Console.WriteLine(result);
```

```
        Console.Read();
      }
      public static int Add(int a1, int a2)
      {
         return a1 + a2;
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
5
```

Multicast Delegate

Multicast delegate can be used to invoke the multiple methods. The delegate instance can do multicasting (adding new method on existing delegate instance) using the **"+"** operator and **"-"** operator can be used to remove a method from a delegate instance. All methods will invoke in sequence as they are assigned.

In Following example, a delegate instance **MulPlusDel** invokes the methods ***Add()***,***SubDel()*** and ***CrossDel()***.

```
namespace delegatesSample
{
   class Program
   {
      //Declaring a delegate
      public delegate void Calculator(int n1, int n2);
      static void Main(string[] args)
      {
         //create delegate instances
         Calculator addDel = new Calculator(Add);
         Calculator SubDel = new Calculator(Subtract);
         Calculator CrossDel = new Calculator(Multiply);
         Calculator MulPlusDel, multiMinusHiDel;
         // The three delegates, addDel, SubDel and CrossDel, are combined to MulPlusDel.
         MulPlusDel = addDel + SubDel + CrossDel;
```

```
            //Remove SubDel from the MulPlusDel delegate
            multiMinusHiDel = MulPlusDel - SubDel;
            MulPlusDel(5, 2);
            multiMinusHiDel(6, 2);

            Console.Read();
        }
        public static void Add(int a1, int a2)
        {
            Console.WriteLine(a1 + a2);
        }
        public static void Subtract(int a1, int a2)
        {
            Console.WriteLine(a1 - a2);
        }
        public static void Multiply(int a1, int a2)
        {
            Console.WriteLine(a1 * a2);
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
7
3
10
8
12
```

Generic Delegate

Generic Delegate was introduced in .NET **3.5** that don't require to define the delegate instance in order to invoke the methods.

There are three types of generic delegates:

- Action
- Func
- Predicate

Action Delegate

Encapsulates a method that has a single parameter and does not return a value. In other words, Action generic delegate, points to a method that takes up to 16 Parameters and returns void.

In following example, we create one Action delegate "Action<string>" with single input parameter as a string.

```
namespace delegatesSample
{
   class Program
   {
      static void Main(string[] args)
      {
         Action<string> actdel=new action<string<(ModelNumber);

         ActDel("2013");
         Console.Read();
      }
      public static void ModelNumber(string year)
      {
         Console.WriteLine("vehicle Model number = " + year);
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Vehicle model number = 2013
```

Func Delegate

Encapsulates a method that has no parameters and returns a value of the type. In other words, The generic **Func** delegate is used when we want to point to a method that returns a value. **Always remember that the final parameter of Func<> is always the return value of the method**. For examle **Func <int, int, string>**, this version of the Func<> delegate will take 2 int parameters and returns a string value.

Note

Last parameter of func delegate is return value of method.

In following example, we create one Func delegate Func <int, int, string> with two input parameter as an int and return value type is string as its last parameter while instantiate Func delegate.

```
namespace delegatesSample
{
   class Program
   {
      static void Main(string[] args)
      {
         Func<int, int, string> FucDel = new Func<int, int, string>(AddNumber);
         Console.WriteLine(FucDel(4, 5));
         Console.Read();
      }
      public static string AddNumber(int n1, int n2)
      {
         return "Total = " + (n1 + n2);
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Total  = 9
```

Predicate Delegate

The Predicate delegate defines a method that can be called on arguments and always returns **Boolean** type result.

In following example, we create one Predicate delegate Predicate<string> with one input parameter as a string and its return bool type value.

```
namespace delegatesSample
{
   class Program
   {
      static void Main(string[] args)
      {
         Predicate<string> PreDel = new Predicate<string>(IsStringNull);

         Console.WriteLine(PreDel(""));
         Console.Read();
      }


      public static bool IsStringNull(string input)
      {
         if(string.IsNullOrEmpty(input))
         {
            return true;
         }
         else
         {
            return false;
         }
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
true
```

Interview Questions And Answers

Question 1 : What you mean by delegate in C#?

Answer: Delegates are type safe pointers unlike function pointers as in C++. Delegate is used to represent the reference of the methods of some return type and parameters.

Question 2 : What are the types of delegates in C#?

Answer:

Below are the uses of delegates in C# :

- Single Delegate
- Multicast Delegate
- Generic Delegate

Question 3 : What are the three types of Generic delegates in C#?

Answer:

Below are the three types of generic delegates in C# :

- Func
- Action
- Predicate

Question 4 : What are the differences between events and delegates in C#?

Answer: Main difference between event and delegate is event will provide one more of encapsulation over delegates. So when you are using events destination will listen to it but delegates are naked, which works in subscriber/destination model.

Question 5 : Can we use delegates for asynchronous method calls in C#?

Answer: Yes. We can use delegates for asynchronous method calls.

Question 6 : What are the uses of delegates in C#?

Answer:

Below are the list of uses of delegates in C# :

- Callback Mechanism
- Asynchronous Processing
- Abstract and Encapsulate method
- Multicasting

Question 7 : Define Multicast Delegate in C#?

Answer: A delegate with multiple handlers are called as multicast delegate.

http://iqdotnet.com/CSharp/CSharp_Delegates_With_Example

What is the use of ObservableCollection in .net?

ObservableCollection is a collection that allows code outside the collection be aware of when changes to the collection (add, move, remove) occur. It is used heavily in WPF and Silverlight but its use is not limited to there. Code can add event handlers to see when the collection has changed and then react through the event handler to do some additional processing. This may be changing a UI or performing some other operation.

The code below doesn't really do anything but demonstrates how you'd attach a handler in a class and then use the event args to react in some way to the changes. WPF already has many operations like refreshing the UI built in so you get them for free when using ObservableCollections

```
class Handler
{

   private ObservableCollection<string> collection;

   public Handler()
   {
      collection = new ObservableCollection<string>();
      collection.CollectionChanged += HandleChange;
   }

   private void HandleChange(object sender, NotifyCollectionChangedEventArgs e)
   {
      foreach (var x in e.NewItems)
      {
         // do something
      }

      foreach (var y in e.OldItems)
```

```
    {
      //do something
    }
    if (e.Action == NotifyCollectionChangedAction.Move)
    {
      //do something
    }
  }
}
```

sealed (C# Reference)

Visual Studio 2015

Other Versions

Updated: July 20, 2015

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.

When applied to a class, the `sealed` modifier prevents other classes from inheriting from it. In the following example, class `B` inherits from class `A`, but no class can inherit from class `B`.

```
class A {}
sealed class B : A {}
```

You can also use the `sealed` modifier on a method or property that overrides a virtual method or property in a base class. This enables you to allow classes to derive from your class and prevent them from overriding specific virtual methods or properties.

Example

In the following example, `Z` inherits from `Y` but `Z` cannot override the virtual function `F` that is declared in `X` and sealed in `Y`.

C#

```
    class X
```

```
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}
class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}
class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("C.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

When you define new methods or properties in a class, you can prevent deriving classes from overriding them by not declaring them as virtual.

It is an error to use the abstract modifier with a sealed class, because an abstract class must be inherited by a class that provides an implementation of the abstract methods or properties.

When applied to a method or property, the `sealed` modifier must always be used with override.

Because structs are implicitly sealed, they cannot be inherited.

For more information, see Inheritance.

For more examples, see Abstract and Sealed Classes and Class Members.

Example

C#

```
sealed class SealedClass
{
    public int x;
    public int y;
}

class SealedTest2
{
    static void Main()
    {
        SealedClass sc = new SealedClass();
        sc.x = 110;
        sc.y = 150;
        Console.WriteLine("x = {0}, y = {1}", sc.x, sc.y);
```

```
        }
    }
    // Output: x = 110, y = 150
```

In the previous example, you might try to inherit from the sealed class by using the following statement:

```
class MyDerivedC: SealedClass {} // Error
```

The result is an error message:

```
'MyDerivedC' cannot inherit from sealed class 'SealedClass'.
```

https://msdn.microsoft.com/en-in/library/88c54tsw.aspx

Partial Classes and Methods (C# Programming Guide)

Visual Studio 2015

Other Versions

Updated: July 20, 2015

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.

It is possible to split the definition of a class or a struct, an interface or a method over two or more source files. Each source file contains a section of the type or method definition, and all parts are combined when the application is compiled.

Partial Classes

There are several situations when splitting a class definition is desirable:

- When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.

- When working with automatically generated source, code can be added to the class without having to recreate the source file. Visual Studio uses this approach when it creates Windows Forms, Web service wrapper code, and so on. You can create code that uses these classes without having to modify the file created by Visual Studio.

- To split a class definition, use the partial keyword modifier, as shown here:

C#

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

The `partial` keyword indicates that other parts of the class, struct, or interface can be defined in the namespace. All the parts must use the `partial`keyword. All the parts must be available at compile time to form the final type. All the parts must have the same accessibility, such as `public`, `private`, and so on.

If any part is declared abstract, then the whole type is considered abstract. If any part is declared sealed, then the whole type is considered sealed. If any part declares a base type, then the whole type inherits that class.

All the parts that specify a base class must agree, but parts that omit a base class still inherit the base type. Parts can specify different base interfaces, and the final type implements all the interfaces listed by all the partial declarations. Any class, struct, or interface members declared in a partial definition are available to all the other parts. The final type is the combination of all the parts at compile time.

Note

The `partial` modifier is not available on delegate or enumeration declarations.

The following example shows that nested types can be partial, even if the type they are nested within is not partial itself.

C#

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }
    partial class Nested
    {
        void Test2() { }
    }
}
```

At compile time, attributes of partial-type definitions are merged. For example, consider the following declarations:

C#

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

They are equivalent to the following declarations:

C#

```
[SerializableAttribute]
[ObsoleteAttribute]
class Moon { }
```

The following are merged from all the partial-type definitions:

- XML comments
- interfaces
- generic-type parameter attributes
- class attributes
- members

For example, consider the following declarations:

C#

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

They are equivalent to the following declarations:

C#

```
class Earth : Planet, IRotate, IRevolve { }
```

Restrictions

There are several rules to follow when you are working with partial class definitions:

- All partial-type definitions meant to be parts of the same type must be modified with `partial`. For example, the following class declarations generate an error:

  C#

  ```
  public partial class A { }
  //public class A { }  // Error, must also be marked partial
  ```

- The `partial` modifier can only appear immediately before the keywords `class`, `struct`, or `interface`.

- Nested partial types are allowed in partial-type definitions as illustrated in the following example:

  C#

  ```
  partial class ClassWithNestedClass
  {
      partial class NestedClass { }
  }

  partial class ClassWithNestedClass
  {
      partial class NestedClass { }
  }
  ```

- All partial-type definitions meant to be parts of the same type must be defined in the same assembly and the same module (.exe or .dll file). Partial definitions cannot span multiple modules.

- The class name and generic-type parameters must match on all partial-type definitions. Generic types can be partial. Each partial declaration must use the same parameter names in the same order.

- The following keywords on a partial-type definition are optional, but if present on one partial-type definition, cannot conflict with the keywords specified on another partial definition for the same type:
    - public
    - private
    - protected
    - internal
    - abstract
    - sealed
    - base class
    - new modifier (nested parts)
    - generic constraints

      For more information, see Constraints on Type Parameters.

Example 1

Description

In the following example, the fields and the constructor of the class, `CoOrds`, are declared in one partial class definition, and the member, `PrintCoOrds`, is declared in another partial class definition.

Code

C#

```
public partial class CoOrds
{
    private int x;
    private int y;

    public CoOrds(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

```
    public partial class CoOrds
    {
        public void PrintCoOrds()
        {
            Console.WriteLine("CoOrds: {0},{1}", x, y);
        }

    }

    class TestCoOrds
    {
        static void Main()
        {
            CoOrds myCoOrds = new CoOrds(10, 15);
            myCoOrds.PrintCoOrds();

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
    // Output: CoOrds: 10,15
```

Example 2

Description

The following example shows that you can also develop partial structs and interfaces.

Code

C#

```
partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}
```

Partial Methods

A partial class or struct may contain a partial method. One part of the class contains the signature of the method. An optional implementation may be defined in the same part or another part. If the implementation is not supplied, then the method and all calls to the method are removed at compile time.

Partial methods enable the implementer of one part of a class to define a method, similar to an event. The implementer of the other part of the class can decide whether to implement the method or not. If the method is not implemented, then the compiler removes the method signature and all calls to the method. The calls to the method, including any results that would occur from evaluation of arguments in the calls, have no effect at run time. Therefore, any code in the partial class can freely use a partial method, even if the implementation is not supplied. No compile-time or run-time errors will result if the method is called but not implemented.

Partial methods are especially useful as a way to customize generated code. They allow for a method name and signature to be reserved, so that generated code can call the method but the developer can decide whether to implement the method. Much like partial classes, partial methods enable code created by a code generator and code created by a human developer to work together without run-time costs.

A partial method declaration consists of two parts: the definition, and the implementation. These may be in separate parts of a partial class, or in the same part. If there is no implementation declaration, then the compiler optimizes away both the defining declaration and all calls to the method.

```
// Definition in file1.cs
partial void onNameChanged();

// Implementation in file2.cs
partial void onNameChanged()
{
  // method body
}
```

- Partial method declarations must begin with the contextual keyword partial and the method must return void.
- Partial methods can have ref but not out parameters.
- Partial methods are implicitly private, and therefore they cannot be virtual.
- Partial methods cannot be extern, because the presence of the body determines whether they are defining or implementing.
- Partial methods can have static and unsafe modifiers.

- Partial methods can be generic. Constraints are put on the defining partial method declaration, and may optionally be repeated on the implementing one. Parameter and type parameter names do not have to be the same in the implementing declaration as in the defining one.

- You can make a delegate to a partial method that has been defined and implemented, but not to a partial method that has only been defined.

https://msdn.microsoft.com/en-IN/library/wa80x488.aspx

extern (C# Reference)

Visual Studio 2015

Other Versions

Updated: July 20, 2015

For the latest documentation on Visual Studio 2017 RC, see Visual Studio 2017 RC Documentation.

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

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

The `extern` keyword can also define an external assembly alias, which makes it possible to reference different versions of the same component from within a single assembly. For more information, see extern alias.

It is an error to use the abstract and `extern` modifiers together to modify the same member. Using the `extern` modifier means that the method is implemented outside the C# code, whereas using the `abstract` modifier means that the method implementation is not provided in the class.

The extern keyword has more limited uses in C# than in C++. To compare the C# keyword with the C++ keyword, see Using extern to Specify Linkage in the C++ Language Reference.

Example

Example 1. In this example, the program receives a string from the user and displays it inside a message box. The program uses the `MessageBox`method imported from the User32.dll library.

C#

```
    //using System.Runtime.InteropServices;
    class ExternTest
    {
        [DllImport("User32.dll", CharSet=CharSet.Unicode)]
        public static extern int MessageBox(IntPtr h, string m, string c, int
type);

        static int Main()
        {
            string myString;
            Console.Write("Enter your message: ");
            myString = Console.ReadLine();
            return MessageBox((IntPtr)0, myString, "My Message Box", 0);
        }

    }
```

Example

Example 2. This example illustrates a C# program that calls into a C library (a native DLL).

1. Create the following C file and name it `cmdll.c`:

```
// cmdll.c
// Compile with: /LD
int __declspec(dllexport) SampleMethod(int i)
{
   return i*10;
}
```

Example

2. Open a Visual Studio x64 (or x32) Native Tools Command Prompt window from the Visual Studio installation directory and compile the `cmdll.c`file by typing **cl /LD cmdll.c** at the command prompt.

3. In the same directory, create the following C# file and name it `cm.cs`:

```
// cm.cs
```

```
using System;
using System.Runtime.InteropServices;
public class MainClass
{
   [DllImport("Cmdll.dll")]
   public static extern int SampleMethod(int x);

   static void Main()
   {
      Console.WriteLine("SampleMethod() returns {0}.", SampleMethod(5));
   }
}
```

Example

3. Open a Visual Studio x64 (or x32) Native Tools Command Prompt window from the Visual Studio installation directory and compile the `cm.cs`file by typing:

csc cm.cs (for the x64 command prompt)
—or—
csc /platform:x86 cm.cs (for the x32 command prompt)

This will create the executable file `cm.exe`.

4. Run `cm.exe`. The `SampleMethod` method passes the value 5 to the DLL file, which returns the value multiplied by 10. The program produces the following output:

```
SampleMethod() returns 50.
```

https://msdn.microsoft.com/en-in/library/e59b22c5.aspx

Why we need two trees, visual and logic tree

Here is how the docs explain what it means that ContentElements are not really in the visual tree:
Content elements (derived classes of ContentElement) are not part of the visual tree; they do not inherit from Visual and have no visual representation. In order to appear in a UI at all, a ContentElement must be hosted within a content host that is a Visual, usually a FrameworkElement. You can conceptualize that the content host is somewhat like a "browser" for the content and chooses how to display that content within the screen region the host controls. Once the content is hosted, the content can be made a participant in certain tree processes that are normally associated with the visual tree. Generally the FrameworkElement host class includes implementation code that adds any hosted ContentElement to the event route through subnodes of the content logical tree, even though the hosted content is not part of the true visual tree. This is

necessary so that a ContentElement can source a routed event that routes to any element other than itself.
So what does all this mean? Well, it means that you can't always just use VisualTreeHelper to traverse the visual tree. If you pass a ContentElement to VisualTreeHelper's GetParent or GetChild methods, an exception will be thrown because ContentElement does not derive from Visual or Visual3D. In order to walk up the visual tree, you need to check each element along the way to see if it descends from Visual or Visual3D, and if it does not, then you must temporarily walk up the logical tree until you encounter another visual object. For example, here's some code which walks up to the root element in a visual tree:

Hide Copy Code

```
DependencyObject FindVisualTreeRoot(DependencyObject initial)
{
   DependencyObject current = initial;
   DependencyObject result = initial;

   while (current != null)
   {
      result = current;
      if (current is Visual || current is Visual3D)
      {
         current = VisualTreeHelper.GetParent(current);
      }
      else
      {
         // If we're in Logical Land then we must walk
         // up the logical tree until we find a
         // Visual/Visual3D to get us back to Visual Land.
         current = LogicalTreeHelper.GetParent(current);
      }
   }

   return result;
}
```

This code walks up the logical tree when necessary, as seen in the else clause. This is useful if, say, the user clicks on a Run element within a TextBlock and in your code you need to walk up the visual tree starting at that Run. Since the Run class descends from ContentElement, the Run is not "really" in the visual tree so we need to walk up out of "Logical Land" until we encounter the TextBlock which contains the Run. At that point we will be back in "Visual Land" since TextBlock is not a ContentElement subclass (i.e. it is a real part of a visual tree).

The Logical Tree

The logical tree represents the essential structure of your UI. It closely matches the elements you declare in XAML, and excludes most visual elements created internally to help render the elements you declared. WPF uses the logical tree to determine several things including dependency property value inheritance, resource resolution, and more.

Working with the logical tree is not nearly as clear-cut as the visual tree. For starters, the logical tree can contain objects of any type. This differs from the visual tree, which only contains instances of DependencyObjectsubclasses. When working with the logical tree, you must keep in mind a leaf node in the tree (a terminus) can be of any type. Since LogicalTreeHelper only works with DependencyObject subclasses, you need to be careful about type checking the objects while walking down the tree. For example:

Hide Copy Code

```
void WalkDownLogicalTree(object current)
{
   DoSomethingWithObjectInLogicalTree(current);

   // The logical tree can contain any type of object, not just
   // instances of DependencyObject subclasses.  LogicalTreeHelper
   // only works with DependencyObject subclasses, so we must be
   // sure that we do not pass it an object of the wrong type.
   DependencyObject depObj = current as DependencyObject;

   if (depObj != null)
      foreach(object logicalChild in LogicalTreeHelper.GetChildren(depObj))
         WalkDownLogicalTree(logicalChild);
}
```

A given Window/Page/Control will have one visual tree, but can contain any number of logical trees. Those logical trees are not connected to each other, so you cannot just use LogicalTreeHelper to navigate between them. In this article, I refer to the top level control's logical tree as the "main logical tree" and all of the other logical trees within it as "logical islands". Logical islands are really just regular logical trees, but I think that the term "island" helps to convey the fact that they are not connected to the main logical tree.
This weirdness can all be boiled down to one word: **templates**.
Controls and data objects have no intrinsic visual appearance; instead they rely on templates to explain how they should render. A template is like a cookie-cutter which can be "expanded" to create real live visual elements used to render something. The elements that are part of an expanded template, hereafter referred to as "template elements", form their own logical tree which is disconnected from the logical tree of the object for which they were created. Those little logical trees are what I refer to as "logical islands" in this article.

You have to write extra code if you need to jump between logical islands/trees. Bridging those logical islands together, while walking **up** logical trees, involves making use of the TemplatedParent property
of FrameworkElement or FrameworkContentElement. TemplatedParent returns the element which has the template applied to it, and, thus, contains a logical island. Here is a method which finds the TemplatedParentof any element:
https://www.codeproject.com/Articles/21495/Understanding-the-Visual-Tree-and-Logical-Tree-in

http://www.informit.com/articles/article.aspx?p=2115888&seqNum=2

Fundamentals of Garbage Collection

In the common language runtime (CLR), the garbage collector serves as an automatic memory manager. It provides the following benefits:

- Enables you to develop your application without having to free memory.
- Allocates objects on the managed heap efficiently.
- Reclaims objects that are no longer being used, clears their memory, and keeps the memory available for future allocations. Managed objects automatically get clean content to start with, so their constructors do not have to initialize every data field.
- Provides memory safety by making sure that an object cannot use the content of another object.

Fundamentals of memory

The following list summarizes important CLR memory concepts.

- Each process has its own, separate virtual address space. All processes on the same computer share the same physical memory, and share the page file if there is one.
- By default, on 32-bit computers, each process has a 2-GB user-mode virtual address space.
- As an application developer, you work only with virtual address space and never manipulate physical memory directly. The garbage collector allocates and frees virtual memory for you on the managed heap.

  If you are writing native code, you use Win32 functions to work with the virtual address space. These functions allocate and free virtual memory for you on native heaps.
- Virtual memory can be in three states:
  - **Free**. The block of memory has no references to it and is available for allocation.
  - **Reserved**. The block of memory is available for your use and cannot be used for any other allocation request. However, you cannot store data to this memory block until it is committed.
  - **Committed**. The block of memory is assigned to physical storage.

- Virtual address space can get fragmented. This means that there are free blocks, also known as holes, in the address space. When a virtual memory allocation is requested, the virtual memory manager has to find a single free block that is large enough to satisfy that allocation request. Even if you have 2 GB of free space, the allocation that requires 2 GB will be unsuccessful unless all of that space is in a single address block.

Conditions for a garbage collection

Garbage collection occurs when one of the following conditions is true:

- The system has low physical memory.
- The memory that is used by allocated objects on the managed heap surpasses an acceptable threshold. This threshold is continuously adjusted as the process runs.
- The GC.Collect method is called. In almost all cases, you do not have to call this method, because the garbage collector runs continuously. This method is primarily used for unique situations and testing.

The managed heap

After the garbage collector is initialized by the CLR, it allocates a segment of memory to store and manage objects. This memory is called the managed heap, as opposed to a native heap in the operating system.

There is a managed heap for each managed process. All threads in the process allocate memory for objects on the same heap.

To reserve memory, the garbage collector calls the Win32 **VirtualAlloc** function, and reserves one segment of memory at a time for managed applications. The garbage collector also reserves segments as needed, and releases segments back to the operating system (after clearing them of any objects) by calling the Win32 **VirtualFree** function.

> The size of segments allocated by the garbage collector is implementation-specific and is subject to change at any time, including in periodic updates. Your app should never make assumptions about or depend on a particular segment size, nor should it attempt to configure the amount of memory available for segment allocations.

The fewer objects allocated on the heap, the less work the garbage collector has to do.
When you allocate objects, do not use rounded-up values that exceed your needs, such as allocating an array of 32 bytes when you need only 15 bytes.
When a garbage collection is triggered, the garbage collector reclaims the memory that is occupied by dead objects. The reclaiming process compacts live objects so that they are

moved together, and the dead space is removed, thereby making the heap smaller. This ensures that objects that are allocated together stay together on the managed heap, to preserve their locality.
The intrusiveness (frequency and duration) of garbage collections is the result of the volume of allocations and the amount of survived memory on the managed heap.
The heap can be considered as the accumulation of two heaps: the large object heap and the small object heap.
The large object heap contains very large objects that are 85,000 bytes and larger. The objects on the large object heap are usually arrays. It is rare for an instance object to be extremely large.

Generations

The heap is organized into generations so it can handle long-lived and short-lived objects. Garbage collection primarily occurs with the reclamation of short-lived objects that typically occupy only a small part of the heap. There are three generations of objects on the heap:

- **Generation 0**. This is the youngest generation and contains short-lived objects. An example of a short-lived object is a temporary variable. Garbage collection occurs most frequently in this generation.

  Newly allocated objects form a new generation of objects and are implicitly generation 0 collections, unless they are large objects, in which case they go on the large object heap in a generation 2 collection.

  Most objects are reclaimed for garbage collection in generation 0 and do not survive to the next generation.

- **Generation 1**. This generation contains short-lived objects and serves as a buffer between short-lived objects and long-lived objects.

- **Generation 2**. This generation contains long-lived objects. An example of a long-lived object is an object in a server application that contains static data that is live for the duration of the process.

Garbage collections occur on specific generations as conditions warrant. Collecting a generation means collecting objects in that generation and all its younger generations. A generation 2 garbage collection is also known as a full garbage collection, because it reclaims all objects in all generations (that is, all objects in the managed heap).

What happens during a garbage collection

A garbage collection has the following phases:

- A marking phase that finds and creates a list of all live objects.
- A relocating phase that updates the references to the objects that will be compacted.
- A compacting phase that reclaims the space occupied by the dead objects and compacts the surviving objects. The compacting phase moves objects that have survived a garbage collection toward the older end of the segment.

  Because generation 2 collections can occupy multiple segments, objects that are promoted into generation 2 can be moved into an older segment. Both generation 1 and generation 2 survivors can be moved to a different segment, because they are promoted to generation 2.

  Ordinarily, the large object heap is not compacted, because copying large objects imposes a performance penalty. However, starting with the .NET Framework 4.5.1, you can use the GCSettings.LargeObjectHeapCompactionMode property to compact the large object heap on demand.

https://msdn.microsoft.com/en-us/library/ee787088(v=vs.110).aspx

Static class can be inherited from another class?

Static classes are sealed and therefore cannot be **inherited**. They cannot **inherit**from any **class** except Object. **Static classes** cannot contain an instance constructor; however, they **can** have a **static** constructor. For more information, see**Static** Constructors (C# Programming Guide)

This is actually by design. There seems to be no good reason to inherit a static class. It has public static members that you can always access via the class name itself. **The only reasons I have seen for inheriting static stuff have been bad ones, such as saving a couple of characters of typing.**

There may be reason to consider mechanisms to bring static members directly into scope (and we will in fact consider this after the Orcas product cycle), but static class inheritance is not the way to go: It is the wrong mechanism to use, and works only for static members that happen to reside in a static class.

(Mads Torgersen, C# Language PM)

Other opinions from channel9

Inheritance in .NET works only on instance base. Static methods are defined on the type level not on the instance level. That is why overriding doesn't work with static methods/properties/events...

Static methods are only held once in memory. There is no virtual table etc. that is created for them.

If you invoke an instance method in .NET, you always give it the current instance. This is hidden by the .NET runtime, but it happens. **Each instance method has as first argument a pointer (reference) to the object that the method is run on. This doesn't happen with static methods** (as they are defined on type level). How should the compiler decide to select the method to invoke?

Extension Methods

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type. For client code written in C# and Visual Basic, there is no apparent difference between calling an extension method and the methods that are actually defined in a type.

https://msdn.microsoft.com/en-IN/library/bb383977.aspx

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}


string s = "Hello Extension Methods";
int i = s.WordCount();
```

Binding Extension Methods at Compile Time

You can use extension methods to extend a class or interface, but not to override them. An extension method with the same name and signature as an interface or class method will never be called. At compile time, extension methods always have lower priority than

instance methods defined in the type itself. In other words, if a type has a method named `Process(int i)`, and you have an extension method with the same signature, the compiler will always bind to the instance method. When the compiler encounters a method invocation, it first looks for a match in the type's instance methods. If no match is found, it will search for any extension methods that are defined for the type, and bind to the first extension method that it finds. The following example demonstrates how the compiler determines which extension method or instance method to bind to.

Example

The following example demonstrates the rules that the C# compiler follows in determining whether to bind a method call to an instance method on the type, or to an extension method. The static class `Extensions` contains extension methods defined for any type that implements `IMyInterface`. Classes `A`, `B`, and `C` all implement the interface.

The `MethodB` extension method is never called because its name and signature exactly match methods already implemented by the classes.

When the compiler cannot find an instance method with a matching signature, it will bind to a matching extension method if one exists.

C#

```
// Define an interface named IMyInterface.
namespace DefineIMyInterface
{
    using System;

    public interface IMyInterface
    {
        // Any class that implements IMyInterface must define a method
        // that matches the following signature.
        void MethodB();
    }
}


// Define extension methods for IMyInterface.
namespace Extensions
{
    using System;
    using DefineIMyInterface;

    // The following extension methods can be accessed by instances of any
    // class that implements IMyInterface.
    public static class Extension
    {
        public static void MethodA(this IMyInterface myInterface, int i)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, int i)");
        }
```

```
        public static void MethodA(this IMyInterface myInterface, string s)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, string
s)");
        }

        // This method is never called in ExtensionMethodsDemo1, because each
        // of the three classes A, B, and C implements a method named MethodB
        // that has a matching signature.
        public static void MethodB(this IMyInterface myInterface)
        {
            Console.WriteLine
                ("Extension.MethodB(this IMyInterface myInterface)");
        }
    }
}


// Define three classes that implement IMyInterface, and then use them to test
// the extension methods.
namespace ExtensionMethodsDemo1
{
    using System;
    using Extensions;
    using DefineIMyInterface;

    class A : IMyInterface
    {
        public void MethodB() { Console.WriteLine("A.MethodB()"); }
    }

    class B : IMyInterface
    {
        public void MethodB() { Console.WriteLine("B.MethodB()"); }
        public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)"); }
    }

    class C : IMyInterface
    {
        public void MethodB() { Console.WriteLine("C.MethodB()"); }
        public void MethodA(object obj)
        {
            Console.WriteLine("C.MethodA(object obj)");
        }
    }

    class ExtMethodDemo
    {
        static void Main(string[] args)
        {
            // Declare an instance of class A, class B, and class C.
            A a = new A();
            B b = new B();
            C c = new C();
```

```
            // For a, b, and c, call the following methods:
            //      -- MethodA with an int argument
            //      -- MethodA with a string argument
            //      -- MethodB with no argument.

            // A contains no MethodA, so each call to MethodA resolves to
            // the extension method that has a matching signature.
            a.MethodA(1);           // Extension.MethodA(object, int)
            a.MethodA("hello");     // Extension.MethodA(object, string)

            // A has a method that matches the signature of the following call
            // to MethodB.
            a.MethodB();            // A.MethodB()

            // B has methods that match the signatures of the following
            // method calls.
            b.MethodA(1);           // B.MethodA(int)
            b.MethodB();            // B.MethodB()

            // B has no matching method for the following call, but
            // class Extension does.
            b.MethodA("hello");     // Extension.MethodA(object, string)

            // C contains an instance method that matches each of the
following
            // method calls.
            c.MethodA(1);           // C.MethodA(object)
            c.MethodA("hello");     // C.MethodA(object)
            c.MethodB();            // C.MethodB()
        }
    }
}
/* Output:
    Extension.MethodA(this IMyInterface myInterface, int i)
    Extension.MethodA(this IMyInterface myInterface, string s)
    A.MethodB()
    B.MethodA(int i)
    B.MethodB()
    Extension.MethodA(this IMyInterface myInterface, string s)
    C.MethodA(object obj)
    C.MethodA(object obj)
    C.MethodB()
 */
```

Nested Types

Use a nested class when the class you are nesting is only useful to the enclosing class. For instance, nested classes allow you to write something like (simplified):

```
public class SortedMap {
   private class TreeNode {
      TreeNode left;
      TreeNode right;
   }
}
```

You can make a complete definition of your class in one place, you don't have to jump through any PIMPL hoops to define how your class works, and the outside world doesn't need to see anything of your implementation.

If the TreeNode class was external, you would either have to make all the fields `public` or make a bunch of `get/set` methods to use it. The outside world would have another class polluting their intellisense.
http://stackoverflow.com/questions/48872/why-when-should-you-use-nested-classes-in-net-or-shouldnt-you

Use of Using keyword

"**Using**" **keyword** takes the parameter of type IDisposable.Whenever you are **using**any IDisposable type object you should **use** the "**using**" **keyword** to handle automatically when it should close or dispose. Internally **using keyword** calls the Dispose() method to dispose the IDisposable object.

Dispose and Finalise

Finalize. 1) Used to free unmanaged resources like files, database connections, COM etc. held by an object before that object is destroyed. //At runtime **C#**destructor is automatically Converted to **Finalize method**. 2) Internally, it is called by Garbage Collector and cannot be called by user code

What happens to FReachable Objects if they are found to be reachable?

If suppose an object implements the Finalize method but inside it refers an alive static object of the application (bad design! but very possible).

Now when GC kicks in and finalises the object by putting it in Finalization queue and then move it to FReachable queue where it would call its finalize method.

But whoa! it finds its referring an alive object so it doesnt allow GC to reclaim the memory occupied by the object and marks the object alive again. A zombie object!

At this point where does this object reside?

1. Remains in freachable?
2. Remains in Finalization queue?
3. Remains on managed heap in an indeterminate state (removed from freachable and finalization queues)?

Also what can be the best place to ReRegisterForFinalize() for such object?

it finds its referring an alive object

That does not matter. An outgoing reference is irrelevant for GC.

Another scenario is where the finalizing objects makes itself reachable again, by registering itself into some rooted list.

This is called resurrection. It does not require much special attention from the GC: the finalizers are processed and the reference is removed from fReachable. Note that there is nothing special, objects in fReachable are *not* in an indeterminate state at any time. They have to be rescanned in the next GC collection. One of the costs of a finalizer is needing 2 rounds of the GC.
Usually the object would call `ReRegisterForFinalize(this)` when it resurrects.
But please note that resurrection is far from a common practice.

Dispose

Garbage collector (GC) plays the main and important role in .NET for memory management so programmer can focus on the application functionality. Garbage collector is responsible for releasing the memory (objects) that is not being used by the application. But GC has limitation that, it can reclaim or release only memory which is used by managed resources. There are a couple of resources which GC is not able to release as it doesn't have information that, how to claim memory from those resources like File handlers, window handlers, network sockets, database connections etc. If your application these resources than it's programs responsibility to release unmanaged resources. For example, if we open a file in our program and not closed it after processing than that file will not be available for other operation or it is being used by other application than they can not open or modify that file. For this purpose FileStream class provides Dispose method. We must call this method after file processing finished. Otherwise it will through exception Access Denied or file is being used by other program.

Close Vs Dispose

Some objects expose Close and Dispose two methods. For Stream classes both serve the same purpose. Dispose method calls Close method inside.

```
1.  void Dispose()
2.  {
3.      this.Close();
4.  }
```

Here question comes, why do we need Dispose method in Stream.Having Dispose method will enable you to write below code and implicitly call dispose method and ultimately will call Close method.

```
1.  using(FileStream file = new FileStream("path", FileMode.Open, FileAccess.Read))
```

```
{
    //Do something with file
}
```

But for some classes both methods behave slightly different. For example Connection class. If Close method is called than it will disconnect with database and release all resources being used by the connection object and Open method will reconnect it again with database without reinitializing the connection object. However Dispose method completely release the connection object and cannot be reopen just calling Open method. We will have re-initialize the Connection object.

Creating Dispose

To implement Dispose method for your custom class, you need to implement IDisposable interface. IDisposable interface expose Dispose method where code to release unmanaged resource will be written.

Finalize

Finalize method also called destructor to the class. Finalize method can not be called explicitly in the code. Only Garbage collector can call the the Finalize when object become inaccessible. Finalize method cannot be implemented directly it can only be implement via declaring destructor. Following class illustrate, how to declare destructor. It is recommend that implement Finalize and Dispose method together if you need to implement Finalize method. After compilation destructor becomes Finalize method.

```
public class MyClass: IDisposable {

    //Construcotr
    public MyClass() {
        //Initialization:
    }

    //Destrucor also called Finalize
    ~MyClass() {
        this.Dispose();
    }

    public void Dispose() {
```

```
14.         //write code to release unmanaged resource.
15.     }
16. }
```

Using Finalize

Now question is, When to implement Finalize? There may be any unmanaged resource for example file stream declared at class level. We may not be knowing what stage or which step should be appropriate to close the file. This object is being use at many places in the application. So in this scenario Finalize can be appropriate location where unmanaged resource can be released. It means, clean the memory acquired by the unmanaged resource as soon as object is inaccessible to application.

Finalize is bit expensive to use. It doesn't clean the memory immediately. When application runs, Garbage collector maintains a separate queue/array when it adds all object which has finalized implemented. Other term GC knows which object has Finalize implemented. When the object is ready to claim memory, Garbage Collector call finalize method for that object and remove from the collection. In this process it just clean the memory that used by unmanaged resource. Memory used by managed resource still in heap as inaccessible reference. That memory release, whenever Garbage Collector run next time. Due to finalize method GC will not clear entire memory associated with object in fist attempt.

Conclusion

It is always recommended that, one should not implement the Finalize method until it is extremely necessary. First priority should always be to implement the Dispose method and clean unmanaged as soon as possible when processing finish with that.
http://www.c-sharpcorner.com/UploadFile/nityaprakash/back-to-basics-dispose-vs-finalize/

.Net Garbage Collection and Finalization Queue

Finalize is a special method that is automatically called by the garbage collector (GC) before the object is collected. This method is only called by the GC. Destructor in C# are automatically translated into Finalize. You can see the IL code using IDASM where you will see that destructor is renamed to finalize.

```
1.          public class A
2.          {
3.          ~A()
4.          {
5.          Console.WriteLine("Class A Destructor");
6.          //Clean up unmanaged resources here
7.          }
8.          }
```

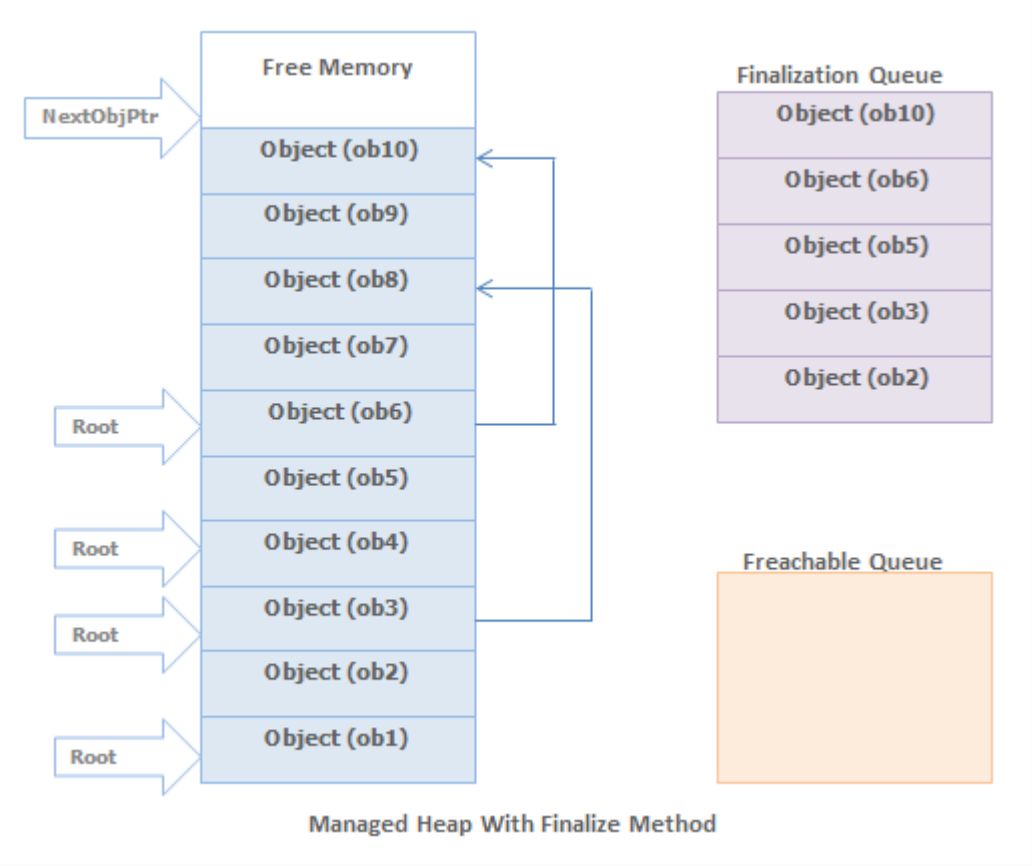
The CLR translate the above C# destructor to like this:

```
1.          public override void Finalize()
2.         {
3.          try
4.          {
5.          Console.WriteLine("Class A Destructor");
6.          //Clean up unmanaged resources here
7.          }
8.          finally
9.          {
10.         base.Finalize();
11.         }
12.        }
```

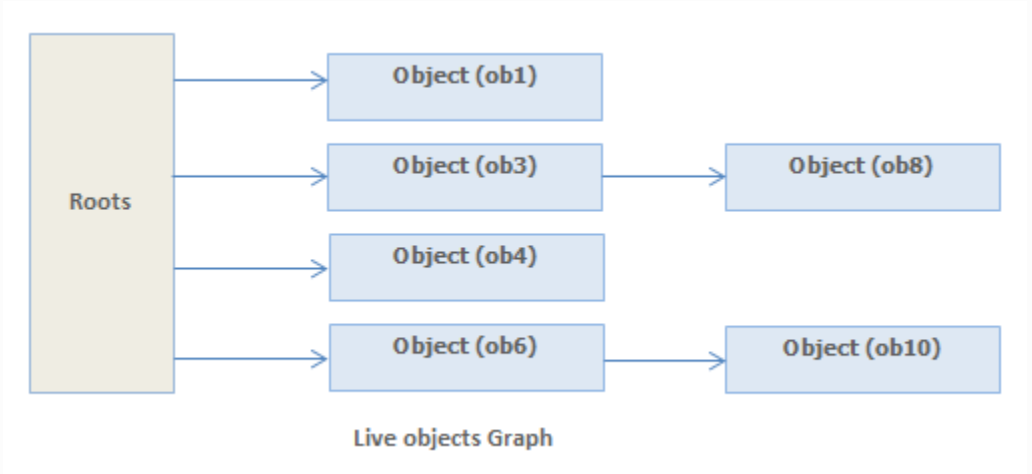
Fundamental of Finalization

When a new object is created, the memory is allocated in the managed heap. If newly created object have a Finalize() method or a destructor then a pointer pointing to that object is put into the finalization queue. Basically, finalization queue is an internal data structure that is controlled and managed by the GC. Hence each pointer in finalization queue points to an object that have its Finalize method call before the memory is reclaimed.

In the below fig. the managed heap contains 10 objects and objects 2,3,5,6,and 10 also contains the Finalize method. Hence pointers to these objects were added to the finalization queue.



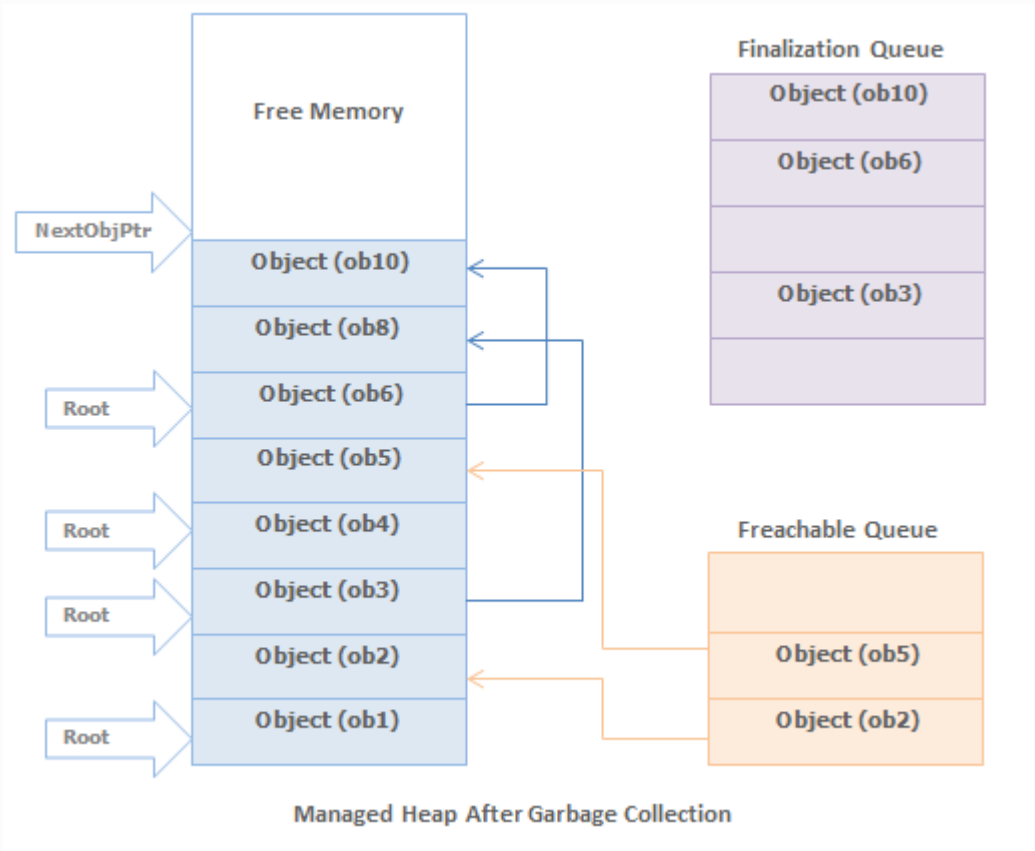
When the garbage collector starts go through the roots, it make a graph of all the objects reachable from the roots. The below fig. shows a heap with allocated objects. In this heap the application roots directly refer to the objects 1,3,4,6 and object 3 & 6 refers to the objects 8 & 10. Hence all these objects will become the part of the live objects graph.



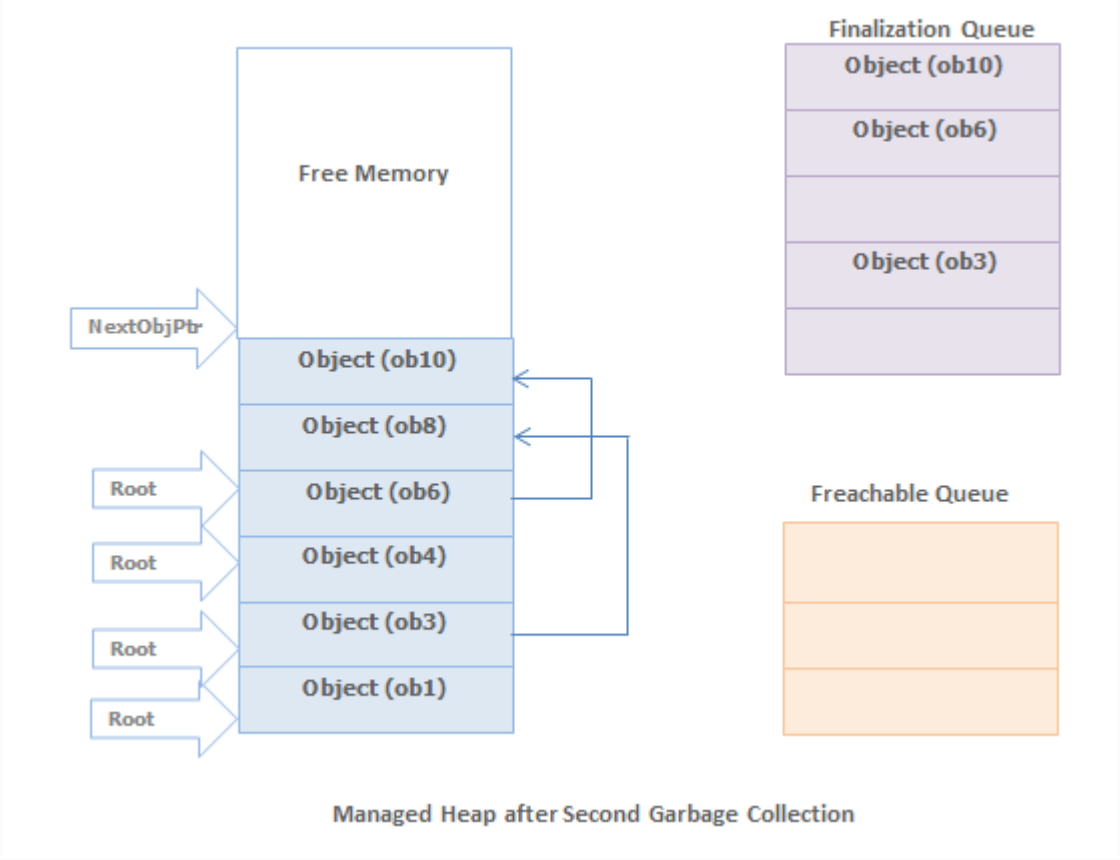
The objects which are not reachable from application's roots, are considered as garbage since these are not accessible by the application. In above heap objects 2,5,7,9 will be considered as dead objects.

Before the collections for dead objects, the garbage collector looks into the finalization queue for pointers identifies these objects. If the pointer found, then the pointer is flushed from the finalization queue and append to the freachable queue .The freachable queue is also an internal data structure and controlled by the garbage collector. Now each and every pointer with in the freachable queue will identify an object that is ready to have its Finalize method called.

After the collection, the managed heap looks like below Fig. Here, you see that the memory occupied by objects 7 and 9 has been reclaimed because these objects did not have a Finalize method. However, the memory occupied by objects 2 and 5 could not be reclaimed because their Finalize method has not been called yet.



When an object's pointer entry move from the finalization queue to the freachable queue, the object is not considered garbage and its memory is not reclaimed. There is a special run-time thread that is dedicated for calling Finalize methods. When there is no entry in the freachable queue then this thread sleeps. But when there is entry in the freachable queue, this thread wakes and removes each entry from the queue by calling each object's Finalize method.



The next time the garbage collector is invoked, it sees that the finalized objects are truly garbage, since the application's roots don't point to it and the freachable queue no longer points to it. Now the memory for the object is simply reclaimed.

Note

1. The two GCs are required to reclaim memory used by objects that have Finalize method.

Reference : http://msdn.microsoft.com/en-us/magazine/bb985010.aspx

http://www.dotnettricks.com/learn/netframework/net-garbage-collection-and-finalization-queue

What's the purpose of GC.SuppressFinalize(this) in Dispose() method?

When implementing the dispose pattern you might also add a finalizer to your class that calls `Dispose()`. This is to make sure that `Dispose()` *always* gets called, even if a client forgets to call it.
To prevent the dispose method from running twice (in case the object already has been disposed) you add `GC.SuppressFinalize(this);`

Freachable queue and Finalization queue

Freachable what? You might ask. Freachable (pronounced F-reachable) is one of CLR Garbage Collector internal structures that is used in a finalization part of garbage collection. You might have heard about the Finalization queue where every object that needs finalization lands initially. This is determined based on whether he has a Finalize method, or it's object type contains a Finalize method definition to speak more precisely. This seems like a good idea, GC wants to keep track of all objects that he needs to call Finalize on, so that when he collects he can find them easily. Why would he need another collection then?

Well apparently what GC does when he finds a garbage object that is on Finalizable queue, is a bit more complicated than you might expect. GC doesn't call the Finalize method directly, instead removes object reference from Finalizable queue and puts it on a (wait for it..) Freachable queue. Weird, huh? Well it turns out there is a specialized CLR thread that is only responsible for monitoring the Freachable queue and when GC adds new items there, he kicks in, takes objects one by one and calls it's Finalize method. One important point about it is that you shouldn't rely on Finalize method being called by the same thread as rest of you app, don't count on Thread Local Storage etc.

But what interest me more is why? Well the article doesn't give an answer to that, but there are two things that come to my mind. First is performance, you obviously want the garbage collection to be as fast as possible and a great deal of work was put into making it so. It seems only natural to keep side tasks like finalization handled by a background thread, so that main one can be as fast a possible. Second, but not less important is that Finalize is after all a client code from the GC perspective, CLR can't really trust your dear reader implementation. Maybe your Finalize will throw exception or will go into infinite loop? It's not something you want to be a part of GC process, it's much less dangerous if it can only affect a background thread.

Implementing Finalize and Dispose to Clean Up Unmanaged Resources

https://msdn.microsoft.com/en-us/library/b1yfkh5e(v=vs.100).aspx

Class instances often encapsulate control over resources that are not managed by the runtime, such as window handles (HWND), database connections, and so on. Therefore, you should provide both an explicit and an implicit way to free those resources. Provide implicit control by implementing the protected Finalize on an object (destructor syntax in C# and C++). The garbage collector calls this method at some point after there are no longer any valid references to the object.

In some cases, you might want to provide programmers using an object with the ability to explicitly release these external resources before the garbage collector frees the object. If an external resource is scarce or expensive, better performance can be achieved if the programmer explicitly releases resources when they are no longer being used. To provide explicit control, implement the Dispose provided by the IDisposable. The consumer of the object should call this method when it is finished using the object. **Dispose** can be called even if other references to the object are alive.

Note that even when you provide explicit control using **Dispose**, you should provide implicit cleanup using the **Finalize** method. **Finalize** provides a backup to prevent resources from permanently leaking if the programmer fails to call **Dispose**.

For more information about implementing **Finalize** and **Dispose** to clean up unmanaged resources, see Garbage Collection. The following example illustrates the basic design pattern for implementing **Dispose**. This example requires the System namespace.

```
// Design pattern for a base class.
public class Base: IDisposable
{
    private bool disposed = false;

    //Implement IDisposable.
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // Free other state (managed objects).
            }
            // Free your own state (unmanaged objects).
```

```
            // Set large fields to null.
            disposed = true;
        }
    }

    // Use C# destructor syntax for finalization code.
    ~Base()
    {
        // Simply call Dispose(false).
        Dispose (false);
    }
}

// Design pattern for a derived class.
public class Derived: Base
{
    private bool disposed = false;

    protected override void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // Release managed resources.
            }
            // Release unmanaged resources.
            // Set large fields to null.
           // Call Dispose on your base class.
            disposed = true;
        }
        base.Dispose(disposing);
    }
    // The derived class does not have a Finalize method
    // or a Dispose method without parameters because it inherits
    // them from the base class.
}
```

The following code expands the previous example to show the different ways **Dispose** is invoked and when **Finalize** is called. The stages of the disposing pattern are tracked with output to the console. The allocation and release of an unmanaged resource is handled in the derived class.

```
using System;
using System.Collections.Generic;
using System.Runtime.InteropServices;

// Design pattern for a base class.
public abstract class Base : IDisposable
{
    private bool disposed = false;
    private string instanceName;
    private List<object> trackingList;
```

```
    public Base(string instanceName, List<object> tracking)
    {
        this.instanceName = instanceName;
         trackingList = tracking;
         trackingList.Add(this);
    }

    public string InstanceName
    {
        get
        {
            return instanceName;
        }
    }

    //Implement IDisposable.
    public void Dispose()
    {
        Console.WriteLine("\n[{0}].Base.Dispose()", instanceName);
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // Free other state (managed objects).
                Console.WriteLine("[{0}].Base.Dispose(true)", instanceName);
                trackingList.Remove(this);
                Console.WriteLine("[{0}] Removed from tracking list: {1:x16}",
                    instanceName, this.GetHashCode());
            }
            else
            {
                Console.WriteLine("[{0}].Base.Dispose(false)", instanceName);
            }
            disposed = true;
        }
    }

    // Use C# destructor syntax for finalization code.
    ~Base()
    {
        // Simply call Dispose(false).
        Console.WriteLine("\n[{0}].Base.Finalize()", instanceName);
        Dispose(false);
    }
}

// Design pattern for a derived class.
public class Derived : Base
{
    private bool disposed = false;
```

```
    private IntPtr umResource;

    public Derived(string instanceName, List<object> tracking) :
        base(instanceName, tracking)
    {
         // Save the instance name as an unmanaged resource
         umResource = Marshal.StringToCoTaskMemAuto(instanceName);
    }

    protected override void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                Console.WriteLine("[{0}].Derived.Dispose(true)",
InstanceName);
                // Release managed resources.
            }
            else
            {
                Console.WriteLine("[{0}].Derived.Dispose(false)",
InstanceName);
            }
            // Release unmanaged resources.
            if (umResource != IntPtr.Zero)
            {
                Marshal.FreeCoTaskMem(umResource);
                Console.WriteLine("[{0}] Unmanaged memory freed at {1:x16}",
                    InstanceName, umResource.ToInt64());
                umResource = IntPtr.Zero;
            }
            disposed = true;
        }
        // Call Dispose in the base class.
        base.Dispose(disposing);
    }
    // The derived class does not have a Finalize method
    // or a Dispose method without parameters because it inherits
    // them from the base class.
}

public class TestDisposal
{
    public static void Main()
    {
        List<object> tracking = new List<object>();

        // Dispose is not called, Finalize will be called later.
        using (null)
        {
            Console.WriteLine("\nDisposal Scenario: #1\n");
            Derived d3 = new Derived("d1", tracking);
        }

        // Dispose is implicitly called in the scope of the using statement.
        using (Derived d1 = new Derived("d2", tracking))
```

```
        {
            Console.WriteLine("\nDisposal Scenario: #2\n");
        }

        // Dispose is explicitly called.
        using (null)
        {
            Console.WriteLine("\nDisposal Scenario: #3\n");
            Derived d2 = new Derived("d3", tracking);
            d2.Dispose();
        }

        // Again, Dispose is not called, Finalize will be called later.
        using (null)
        {
            Console.WriteLine("\nDisposal Scenario: #4\n");
            Derived d4 = new Derived("d4", tracking);
        }

        // List the objects remaining to dispose.
        Console.WriteLine("\nObjects remaining to dispose = {0:d}",
tracking.Count);
        foreach (Derived dd in tracking)
        {
            Console.WriteLine("    Reference Object: {0:s}, {1:x16}",
                dd.InstanceName, dd.GetHashCode());
        }

        // Queued finalizers will be exeucted when Main() goes out of scope.
        Console.WriteLine("\nDequeueing finalizers...");
    }
}

// The program will display output similar to the following:
//
// Disposal Scenario: #1
//
//
// Disposal Scenario: #2
//
//
// [d2].Base.Dispose()
// [d2].Derived.Dispose(true)
// [d2] Unmanaged memory freed at 000000000034e420
// [d2].Base.Dispose(true)
// [d2] Removed from tracking list: 0000000002bf8098
//
// Disposal Scenario: #3
//
//
// [d3].Base.Dispose()
// [d3].Derived.Dispose(true)
// [d3] Unmanaged memory freed at 000000000034e420
// [d3].Base.Dispose(true)
// [d3] Removed from tracking list: 0000000000bb8560
//
// Disposal Scenario: #4
```

```
//
//
// Objects remaining to dispose = 2
//    Reference Object: d1, 000000000297b065
//    Reference Object: d4, 0000000003553390
//
// Dequeueing finalizers...
//
// [d4].Base.Finalize()
// [d4].Derived.Dispose(false)
// [d4] Unmanaged memory freed at 000000000034e420
// [d4].Base.Dispose(false)
//
// [d1].Base.Finalize()
// [d1].Derived.Dispose(false)
// [d1] Unmanaged memory freed at 000000000034e3f0
// [d1].Base.Dispose(false)
```

For an additional code example illustrating the design pattern for implementing **Finalize** and **Dispose**, see Implementing a Dispose Method.

Customizing a Dispose Method Name

Occasionally a domain-specific name is more appropriate than **Dispose**. For example, a file encapsulation might want to use the method name **Close**. In this case, implement **Dispose** privately and create a public **Close** method that calls **Dispose**. The following code example illustrates this pattern. You can replace **Close** with a method name appropriate to your domain. This example requires the System namespace.

C#

VB

```
// Do not make this method virtual.
// A derived class should not be allowed
// to override this method.
public void Close()
{
   // Call the Dispose method with no parameters.
   Dispose();
}
```

Finalize

The following rules outline the usage guidelines for the **Finalize** method:

- Implement **Finalize** only on objects that require finalization. There are performance costs associated with **Finalize** methods.

- If you require a **Finalize** method, consider implementing **IDisposable** to allow users of your class to avoid the cost of invoking the **Finalize** method.
- Do not make the **Finalize** method more visible. It should be **protected**, not **public**.
- An object's **Finalize** method should free any external resources that the object owns. Moreover, a **Finalize** method should release only resources that the object has held onto. The **Finalize** method should not reference any other objects.
- Do not directly call a **Finalize** method on an object other than the object's base class. This is not a valid operation in the C# programming language.
- Call the **base class's Finalize** method from an object's **Finalize** method.

> **Note**
>
> The base class's **Finalize** method is called automatically with the C# and C++ destructor syntax.

Dispose

The following rules outline the usage guidelines for the **Dispose** method:

- Implement the dispose design pattern on a type that encapsulates resources that explicitly need to be freed. Users can free external resources by calling the public **Dispose** method.
- Implement the dispose design pattern on a base type that commonly has derived types that hold onto resources, even if the base type does not. If the base type has a **Close** method, often this indicates the need to implement **Dispose**. In such cases, do not implement a **Finalize** method on the base type. **Finalize** should be implemented in any derived types that introduce resources that require cleanup.
- Free any disposable resources a type owns in its **Dispose** method.
- After **Dispose** has been called on an instance, prevent the **Finalize** method from running by calling the GC.SuppressFinalize. The exception to this rule is the rare situation in which work must be done in **Finalize** that is not covered by **Dispose**.
- Call the base class's **Dispose** method if it implements **IDisposable**.

- Do not assume that **Dispose** will be called. Unmanaged resources owned by a type should also be released in a **Finalize** method in the event that **Dispose** is not called.

- Throw an **ObjectDisposedException** from instance methods on this type (other than **Dispose**) when resources are already disposed. This rule does not apply to the **Dispose** method because it should be callable multiple times without throwing an exception.

- Propagate the calls to **Dispose** through the hierarchy of base types.
  The **Dispose** method should free all resources held by this object and any object owned by this object. For example, you can create an object such as a **TextReader** that holds onto a **Stream** and an **Encoding**, both of which are created by the **TextReader** without the user's knowledge. Furthermore, both the **Stream** and the **Encoding** can acquire external resources. When you call the **Dispose** method on the **TextReader**, it should in turn call **Dispose** on the **Stream** and the **Encoding**, causing them to release their external resources.

- Consider not allowing an object to be usable after its **Dispose** method has been called. Re-creating an object that has already been disposed is a difficult pattern to implement.

- Allow a **Dispose** method to be called more than once without throwing an exception. The method should do nothing after the first call.