**Object Lifecycle**

The life cycle of an object has several stages, which start at creation of the object and end in its destruction. To create an object in your application, you use the new keyword. When the common language runtime (CLR) executes code to create a new object, it performs the following steps:

1. It allocates a block of memory large enough to hold the object.
2. It initializes the block of memory to the new object.

The CLR handles the allocation of memory for all managed objects. However, when you use unmanaged objects, you may need to write code to allocate memory for the unmanaged objects that you create.  Unmanaged objects are those that are not .NET components such as a Microsoft Word object, a database connection, or a file resource.

When you have finished with an object, you can dispose of it to release any resources, such as database connections and file handles, that it consumed. When you dispose of an object, the CLR uses a feature called the garbage collector (GC) to perform the following steps:

1. The GC releases resources.
2. The memory that is allocated to the object is reclaimed.

The GC runs automatically in a separate thread. When the GC runs, other threads in the application are halted, because the GC may move objects in memory and therefore must update the memory pointers.

**Introducing Garbage Collection**

The garbage collector is a separate process the runs in its own thread whenever a managed code application is running. The garbage collection process provides the following benefits:

* Enables you to develop your application without having to worry about freeing 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.

When a .NET application is executed, the garbage collector is initialized by the CLR.  The GC allocates a segment of memory that it will use to store and manage the objects for each .NET application that is running. This memory area is referred to as the managed heap, which differs from a native heap used in the context of the operating system.  
  
There is a managed heap for each managed process that is running and all threads in the process allocate memory for objects, in that process, on the same heap.  This means that each process has its own virtual memory space.  
  
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.

Note: The size of segments allocated by the garbage collector is implementation-specific and is subject to change at any time, including in periodic updates. When writing your app, you should never make assumptions about, or depend on a particular segment size that will be used by the GC.  
  
When a garbage collection is triggered, the process will reclaim memory that is occupied by dead objects, objects no longer referenced in the application code. Reclaiming also compacts live objects so that they are moved together, dead space is removed, which reduces the size of the heap.   
  
The GC does exact a performance hit on the applications because garbage collection is the result of the number of allocations and the amount of memory usage and release on the managed heap.

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

* The system is running low on physical memory.
* The memory that is used by currently allocated objects surpasses an acceptable threshold. This threshold will be continuously adjusted as the process is running.
* The GC.Collect method is called. While you can call this method yourself, typically you do not have to call this method, because the garbage collector runs continuously. Even if you do call this method, there is no guarantee that it will run precisely when you call it.

**Implementing the Dispose Pattern**  
  
The dispose pattern is a design pattern that frees resources that an object has used. The .NET Framework provides the IDisposable interface in the System namespace to enable you to implement the dispose pattern in your applications.

The IDisposable interface defines a single parameterless method named Dispose. You should use the Dispose method to release all of the unmanaged resources that your object consumed. If the object is part of an inheritance hierarchy, the Dispose method can also release resources that the base types consumed by calling the Dispose method on the parent type.

Invoking the Dispose method does not destroy an object. The object remains in memory until the final reference to the object is removed and the GC reclaims any remaining resources.

Many of the classes in the .NET Framework that wrap unmanaged resources, such as the StreamWriter class, implement the IDisposable interface.  The StreamWriter class implements a TextWriter object for the purpose of writing text information to a stream.  The stream could be a file, memory, or network stream.  You should also implement the IDisposable interface when you create your own classes that reference unmanaged types.

**Implementing the IDisposable Interface**

To implement the IDisposable interface in your application, perform the following steps:

1. Ensure that the System namespace is in scope by adding the following using statement to the top of the code file.  
   using System;
2. Implement the IDisposable interface in your class definition.  
   ...

public class ManagedWord : IDisposable  
{  
   public void Dispose()  
   {  
      throw new NotImplementedException();  
   }  
}

1. Add a private field to the class, which you can use to track the disposal status of the object, and check whether the Dispose method has already been invoked and the resources released.   
     
   public class ManagedWord : IDisposable  
   {  
      bool \_isDisposed;  
      ...  
   }
2. Add code to any public methods in your class to check whether the object has already been disposed of. If the object has been disposed of, you should throw an ObjectDisposedException.  
     
   public void OpenWordDocument(string filePath)  
   {  
      if (this.\_isDisposed)  
         throw new ObjectDisposedException("ManagedWord");  
          ...  
   }
3. Add an overloaded implementation of the Dispose method that accepts a Boolean parameter. The overloaded Dispose method should dispose of both managed and unmanaged resources if it was called directly, in which case you pass a Boolean parameter with the value true. If you pass a Boolean parameter with the value of false, the Dispose method should only attempt to release unmanaged resources. You may want to do this if the object has already been disposed of or is about to be disposed of by the GC.   
     
   public class ManagedWord : IDisposable  
   {  
      ...  
       protected virtual void Dispose(bool isDisposing)  
       {  
           if (this.\_isDisposed)  
               return;  
           if (isDisposing)  
           {  
              // Release only managed resources.  
              ...  
           }  
           // Always release unmanaged resources.  
           ...  
           // Indicate that the object has been disposed.  
           this.\_isDisposed = true;  
       }  
   }
4. Add code to the parameterless Dispose method to invoke the overloaded Dispose method and then call the GC.SuppressFinalize method. The GC.SuppressFinalize method instructs the GC that the resources that the object referenced have already been released and the GC does not need to waste time running the finalization code.  
     
   public void Dispose()  
   {  
      Dispose(true);  
      GC.SuppressFinalize(this);  
   }

After you have implemented the IDisposable interface in your class definitions, you can then invoke the Dispose method on your object to release any resources that the object has consumed. You can invoke the Dispose method from a destructor that is defined in the class.

**Implementing a Destructor**

You can add a destructor to a class to perform any additional application-specific cleanup that is necessary when your class is garbage collected. To define a destructor, you add a tilde (~) followed by the name of the class. You then enclose the destructor logic in braces.  
The following code example shows the syntax for adding a destructor.

// Defining a Destructor  
class ManagedWord  
{  
    ...  
    // Destructor  
    ~ManagedWord  
    {  
        // Destructor logic.  
    }  
}

When you declare a destructor, the compiler automatically converts it to an override of the Finalize method of the object class. However, you cannot explicitly override the Finalize method; you must declare a destructor and let the compiler perform the conversion.

If you want to guarantee that the Dispose method is always invoked, you can include it as part of the finalization process that the GC performs. To do this, you can add a call to the Dispose method in the destructor of the class.

The following code example shows how to invoke the Dispose method from a destructor.

// Calling the Dispose Method from a Destructor  
class ManagedWord  
{  
    ...  
    // Destructor  
    ~ManagedWord  
    {  
        Dispose(false);  
    }  
}

**Managing the Lifetime of an Object**  
  
Using types that implement the IDisposable interface is not sufficient to manage resources. You must also remember to invoke the Dispose method in your code when you have finished with the object. If you choose not to implement a destructor that invokes the Dispose method when the GC processes the object, you can do this in a number of other ways.

One approach is to explicitly invoke the Dispose method after any other code that uses the object. The following code example shows how you can invoke the Dispose method on an object that implements the IDisposable interface.

// Invoking the Dispose Method  
var word = new ManagedWord();  
 // Code to use the ManagedWord object.  
word.Dispose();

Invoking the Dispose method explicitly after code that uses the object is perfectly acceptable, but if your code throws an exception before the call to the Dispose method, the Dispose method will never be invoked. A more reliable approach is to invoke the Dispose method in the finally block of a try/catch/finally or a try/finally statement. Any code in the scope of the finally block will always execute, regardless of any exceptions that might be thrown. Therefore, with this approach, you can always guarantee that your code will invoke the Dispose method.

The following code example shows how you can invoke the Dispose method in a finally block.

// Invoking the Dispose Method in a finally Block  
var word = default(ManagedWord);  
try  
{  
   word = new ManagedWord();  
   // Code to use the ManagedWord object.  
}  
catch  
{  
    // Code to handle any errors.  
}  
finally  
{  
   if(word!=null)  
      word.Dispose();  
}

*Note: When explicitly invoking the Dispose method, it is good practice to check whether the object is not null beforehand, because you cannot guarantee the state of the object.*

Alternatively, you can use a using statement to implicitly invoke the Dispose method. A using block is exception safe, which means that if the code in the block throws an exception, the runtime will still dispose of the objects that are specified in the using statement.

The following code example shows how to implicitly dispose of your object by using a using statement.

// Disposing Of an Object by Using a using Statement  
using (var word = default(ManagedWord))  
{  
   // Code to use the ManagedWord object.  
}

If your object does not implement the IDisposable interface, a try/finally block is an exception-safe approach to execute code to release resources. You should aim to use a try/finally block when it is not possible to use a using statement.