# 1. Subscribing to Events

Once you subscribe to an event, that object holds a reference to your class. That is unless you subscribed with an anonymous method that didn’t capture a class member. Consider this example:

public class MyClass

{

public MyClass(WiFiManager wiFiManager)

{

wiFiManager.WiFiSignalChanged += OnWiFiChanged;

}

private void OnWiFiChanged(object sender, WifiEventArgs e)

{

// do something

}

}

Assuming the wifiManager outlives MyClass, you have a memory leak on your hands. Any instance of MyClass is referenced by wifiManager and will never be allocated by the garbage collector.

There are several great pattern to prevent memory leaks from event:

* Unsubscribe from the event.
* Use weak-handler patterns.
* Subscribe if possible with an anonymous function and without capturing any members.

# 2. Capturing members in anonymous methods

While it might be obvious that an event-handler method means an object is referenced, it’s less obvious that the same applies when a class member is captured in an anonymous method.

Here’s an example:

public class MyClass

{

private JobQueue \_jobQueue;

private int \_id;

public MyClass(JobQueue jobQueue)

{

\_jobQueue = jobQueue;

}

public void Foo()

{

\_jobQueue.EnqueueJob(() =>

{

Logger.Log($"Executing job with ID {\_id}");

// do stuff

});

}

}

In this code, the member \_id is captured in the anonymous method and as a result the instance is referenced as well. This means that while JobQueue exists and references that job delegate, it will also reference an instance of MyClass.

The solution can be quite simple – assigning a local variable:

public class MyClass

{

public MyClass(JobQueue jobQueue)

{

\_jobQueue = jobQueue;

}

private JobQueue \_jobQueue;

private int \_id;

public void Foo()

{

var localId = \_id;

\_jobQueue.EnqueueJob(() =>

{

Logger.Log($"Executing job with ID {localId}");

// do stuff

});

}

}

By assigning the value to a local variable, nothing is captured and you’ve averted a potential memory leak.

# 3. Static Variables

Let’s consider how the garbage collector works. The basic idea is that the GC goes over all GC Root objects and marks them as not-to-collect. Then, the GC goes to all the objects they reference and marks as not-to-collect as well. And so on. Finally, the GC collects everything left (great article on garbage collection).

So what is considered as a GC Root?

Live Stack of the running threads.

Static variables.

Managed objects that are passed to COM objects by interop (Memory de-allocation will be done by reference count)

This means that static variables and everything they reference will never be garbage collected. Here’s an example:

public class MyClass

{

static List<MyClass> \_instances = new List<MyClass>();

public MyClass()

{

\_instances.Add(this);

}

}

If, for whatever reason, you decide to write the above code, any instance of MyClass will forever stay in memory, causing a memory leak.

# 4. Caching functionality

Developers love caching. Why do an operation twice when you can do it once and save the result, right?

That’s true enough, but if you cache indefinitely, you will eventually run out of memory. Consider this example:

public class ProfilePicExtractor

{

private Dictionary<int, byte[]> PictureCache { get; set; } =

new Dictionary<int, byte[]>();

public byte[] GetProfilePicByID(int id)

{

// A lock mechanism should be added here, but let's stay on point

if (!PictureCache.ContainsKey(id))

{

var picture = GetPictureFromDatabase(id);

PictureCache[id] = picture;

}

return PictureCache[id];

}

private byte[] GetPictureFromDatabase(int id)

{

// ...

}

}

This piece of code might save some expensive trips to the database, but the price is cluttering your memory.

You can do several things to solve this:

Delete caching that wasn’t used for some time

Limit caching size

Use WeakReference to hold cached objects. This relies on the garbage collector to decide when to clear the cache, but might not be such a bad idea. The GC will promote objects that are still in use to higher generations in order to keep them longer. That means that objects that are used often will stay longer in cache.

# 5. Incorrect WPF Bindings

WPF Bindings can actually cause memory leaks. The rule of thumb is to always bind to a DependencyObject or to a INotifyPropertyChanged object. When you fail to do so, WPF will create a strong reference to your binding source (meaning the ViewModel) from a static variable, causing a memory leak (explanation ).

Here’s an example.

<UserControl x:Class="WpfApp.MyControl"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

<TextBlock Text="{Binding SomeText}"></TextBlock>

</UserControl>

This View Model will stay in memory forever:

public class MyViewModel

{

public string \_someText = "memory leak";

public string SomeText

{

get { return \_someText; }

set

{

\_someText = value;

}

}

}

Whereas this View Model won’t cause a memory leak:

public class MyViewModel : INotifyPropertyChanged

{

public string \_someText = "not a memory leak";

public string SomeText

{

get { return \_someText; }

set

{

\_someText = value;

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(nameof (SomeText)));

}

}

It actually doesn’t matter if you invoke PropertyChanged or not, the important thing is that the class derives from INotifyPropertyChanged. This tells the WPF infrastructure not to create a strong reference.

The memory leak occurs when the binding mode is OneWay or TwoWay. If the binding is OneTime or OneWayToSource, it’s not a problem.

Another WPF memory leak issue occurs when binding to a collection. If that collection doesn’t implement INotifyCollectionChanged, then you will have a memory leak. You can avoid the problem by using ObservableCollection which does implement that interface.

# 6. Threads that Never Terminate

We already talked about how the GC works and about GC roots. I mentioned that the Live Stack is considered as a GC root. The Live Stack includes all local variables and members of the call stacks in the running threads.

If for whatever reason, you were to create an infinitely-running thread that does nothing and has references to objects, that would be a memory leak. One example of how this can easily happen is with a Timer. Consider this code:

public class MyClass

{

public MyClass()

{

Timer timer = new Timer(HandleTick);

timer.Change(TimeSpan.FromSeconds(5), TimeSpan.FromSeconds(5));

}

private void HandleTick(object state)

{

// do something

}

If you don’t actually stop the timer, it will run in a separate thread, referencing an instance of MyClass, preventing it from being collected.

# 7. Not de-allocating unmanaged memory

Up to now, we only talked about managed memory. That is, memory that’s managed by the garbage collector. Unmanaged memory is a whole different matter – Instead of just avoiding unnecessary references, you will need to de-allocate the memory explicitly.

Here’s a simple example:

public class SomeClass

{

private IntPtr \_buffer;

public SomeClass()

{

\_buffer = Marshal.AllocHGlobal(1000);

}

// do stuff without freeing the buffer memory

}

In the above method, we’ve used the Marshal.AllocHGlobal, which allocates a buffer of unmanaged memory (documentation ). Under the hood, AllocHGlobal calls the LocalAlloc function in Kernel32.dll. Without explicitly freeing the handle with Marshal.FreeHGlobal, that buffer memory will be considered as taken in the process`s memory heap, causing a memory leak.

To deal with such issues you can add a Dispose method that frees any unmanaged resources, like so:

public class SomeClass : IDisposable

{

private IntPtr \_buffer;

public SomeClass()

{

\_buffer = Marshal.AllocHGlobal(1000);

// do stuff without freeing the buffer memory

}

public void Dispose()

{

Marshal.FreeHGlobal(\_buffer);

}

}

Unmanaged memory leaks are in a way worst than managed memory leaks due to memory fragmentation issues. Managed memory can be moved around by the garbage collector, making space for other objects. Unmanaged memory, however, is forever stuck in place.

# 8. Adding Dispose without Calling it

In the last example, we added the Dispose method to free any unmanaged resources. That’s great, but what happens when whoever used the class didn’t call Dispose?

One thing you can do is to use the using statement in C#:

using (var instance = new MyClass())

{

// ...

}

This works on IDisposable classes and translates by the compiler to this:

MyClass instance = new MyClass();;

try

{

// ...

}

finally

{

if (instance != null)

((IDisposable)instance).Dispose();

}

This is very useful because even if an exception was thrown, Dispose will still be called.

Another thing you can do is utilize the Dispose Pattern . Here’s an example of how you would implement it:

public class MyClass : IDisposable

{

private IntPtr \_bufferPtr;

public int BUFFER\_SIZE = 1024 \* 1024; // 1 MB

private bool \_disposed = false;

public MyClass()

{

\_bufferPtr = Marshal.AllocHGlobal(BUFFER\_SIZE);

}

protected virtual void Dispose(bool disposing)

{

if (\_disposed)

return;

if (disposing)

{

// Free any other managed objects here.

}

// Free any unmanaged objects here.

Marshal.FreeHGlobal(\_bufferPtr);

\_disposed = true;

}

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

~MyClass()

{

Dispose(false);

}

}

This pattern makes sure that even if Dispose wasn’t called, then it will eventually be called when the instance is garbage collected. If, on the other hand, Dispose was called, then the finalizer is suppressed. Suppressing the finalizer is important because finalizers are expensive and can cause performance issues.

The dispose-pattern is not bulletproof, however. If Dispose was never called and your class wasn’t garbage collected due to a managed memory leak, then the unmanaged resources will not be freed.