Inside the Garbage Collector

DEVELOPMENTOR
DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Agenda

- Applications and Resources
- The Managed Heap
- Mark and Compact
- Generations
- Non-Memory Resources
- Finalization
- Hindering the GC
- Helping the GC

Applications and Resources

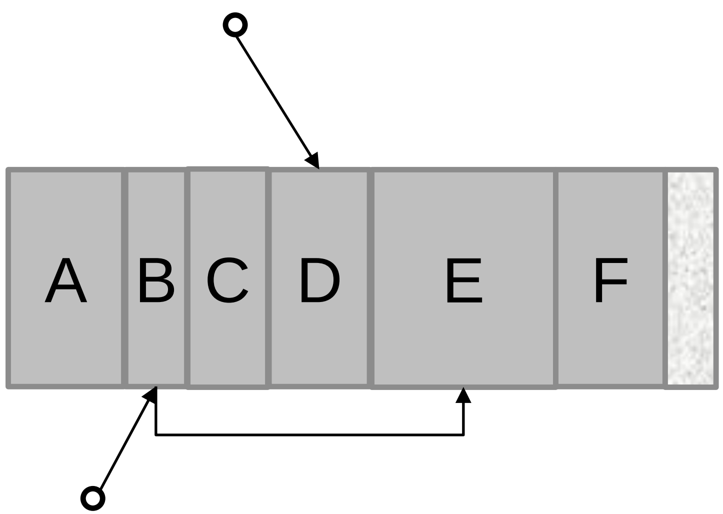
- **Applications use many different kinds of resources when executing**
  - Memory
  - File handles
  - Database connections
  - Sockets
- **.NET provides infrastructure and patterns for managing an applications resources**

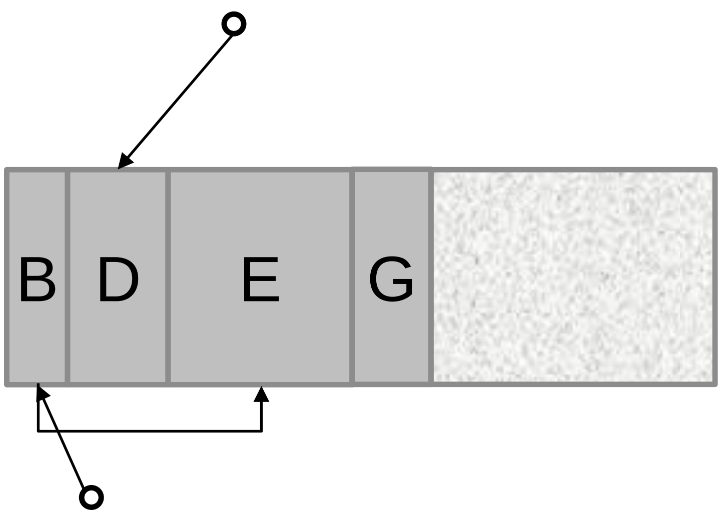
Memory and the Managed Heap

- **.NET has specific functionality to automate memory management**
  - Applications request memory using new keyword
  - Memory is allocated in area called the **Managed Heap**
- **Runtime cleans up memory when necessary using Garbage Collection (GC)**
  - Only managed heap is garbage collected. Unmanaged and stack allocated memory unaffected
  - Garbage Collector removes objects no longer in use

How the Garbage Collector Works

- Applications run as if memory is infinite
- GC is invoked when resource threshold breached
- Assumes all objects are garbage
- Collects those not in use





What Objects are in Use?

- **The garbage collector does not collect objects in use**
  - How does it know what is in use and what is not?
- **GC has concept of objects being reachable**
  - Objects that could be accessed from code that is yet to run
- **GC walks objects graph under live roots**
  - Non-null static references
  - Local variables on the stack that are still in use
  - A few more exotic kinds, e.g. the native code interop infrastructure
- **Any object reachable from a live root survives the GC**

Spot the Live Roots

- **What references are live roots when the GC runs?**

```
static object o1 = new object();

void RunTest()
{
  object o2 = new object();
  object o3 = new object();

  GC.Collect();

  o1 = null;

  GC.Collect();

  ProcessObject(o3);

  GC.Collect();
}
```

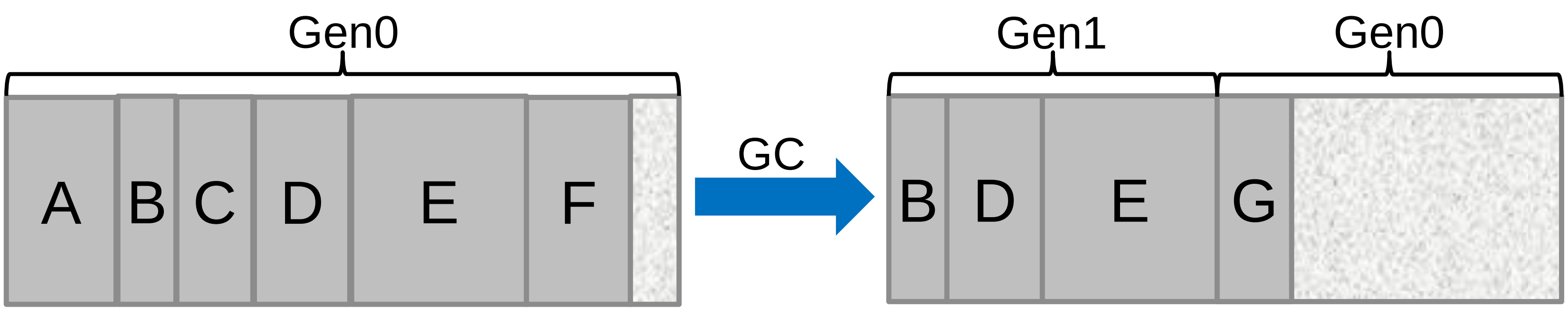
What is garbage here?

What is garbage here?

What is garbage here?

GC Optimization - Generations

- **Examining the entire heap every GC too expensive**
  - Optimization: if object survived last GC then probably still alive
  - By default only checks the most recently allocated objects
- **.NET GC is Generational**
  - Three generations
  - All objects allocated into gen0
  - Any object surviving a collection is promoted to next generation



The Implications of Generations

- **Relative cost of GC**
  - Gen0 is very cheap to collect
  - Gen1 is more expensive to collect that gen0
  - Gen2 can be very expensive to collect
- **Healthy GC**
  - Order of magnitude between number of collections of each successive generation regarded as healthy
  - gen0 : gen1 : gen2 => 100 : 10 : 1
- **Generally don’t try to second guess the GC**

GC Optimization – GC Styles

- **Three styles of GC**
  - Non-concurrent
  - Concurrent
  - Asynchronous

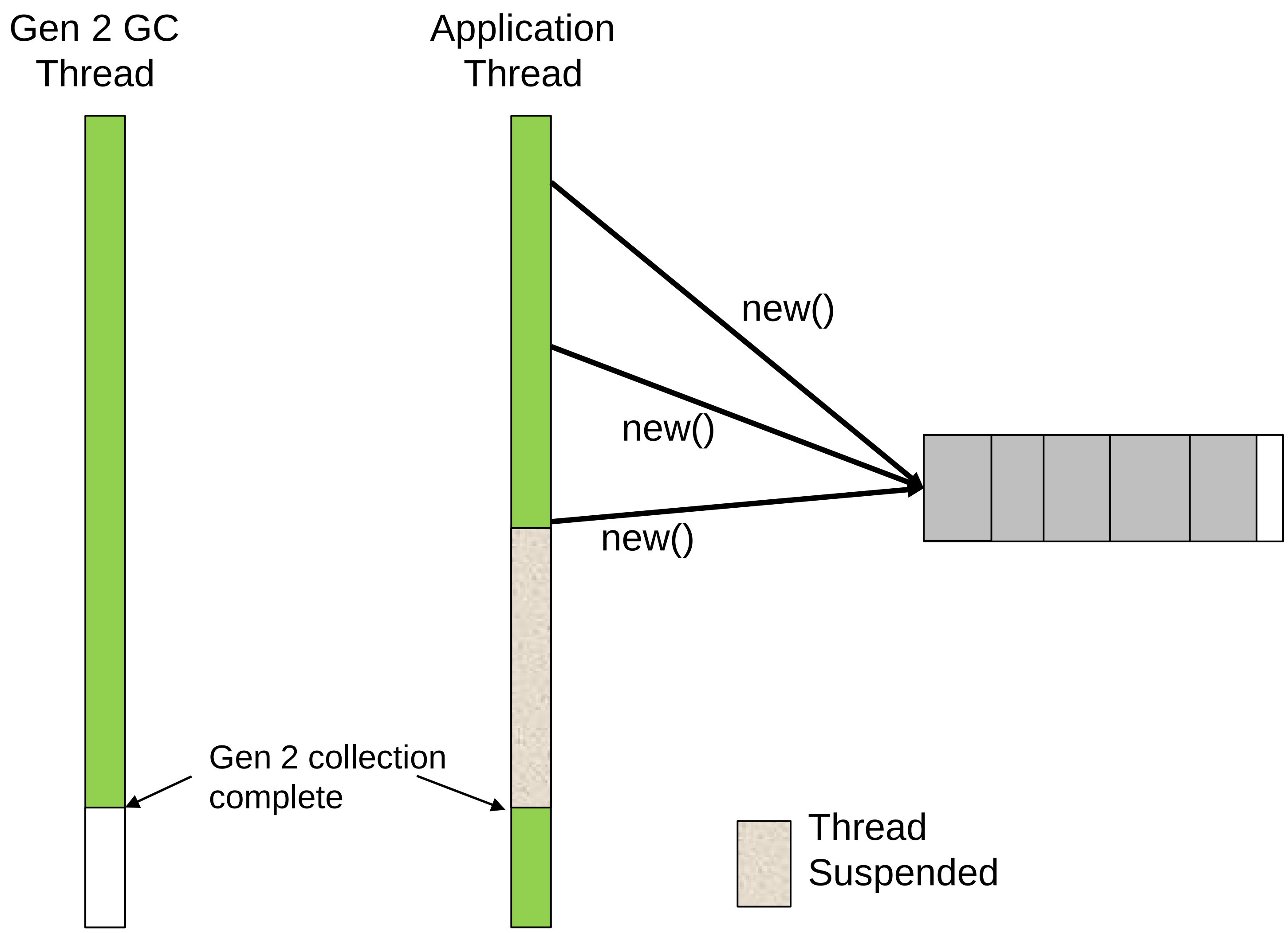
Concurrent and Non-Concurrent

- **Non-concurrent**
  - GC starts and runs through to completion without interruption
  - Always used for Gen 0 and Gen 1 collects
- **Concurrent**
  - Application threads interleaved with collection
  - Keeps application responsive
  - Only used for some Gen 2 collects
  - Can continue application execution until another GC required

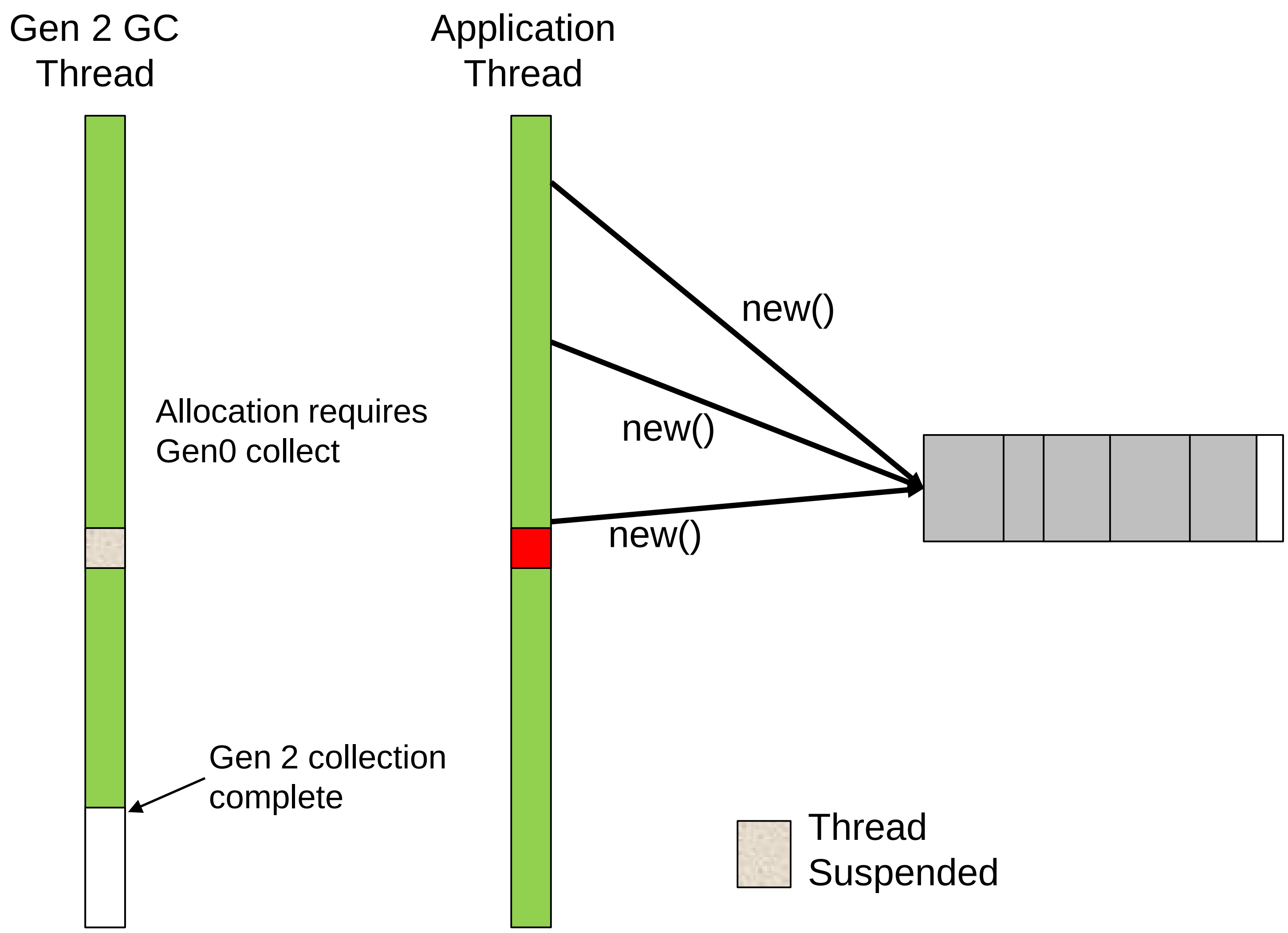
Async

- **Asynchronous**
  - Introduced in version 4.0
  - Similar to concurrent but can perform gen 0 and gen 1 collects during a gen 2 collect

Concurrent GC



Background GC



GC Modes

- **There are three GC modes**
  - Workstation Concurrent (default)
  - Workstation Non-Concurrent
  - Server
- **Controlled by config flags**

```
<configuration>
  <runtime>
    <gcServer enabled="true"/>
    <gcConcurrent enabled="false"/>
  </runtime>
</configuration>
```

Workstation Concurrent

- **Designed to keep UI applications interactive**
  - UI thread paused as short a time as possible
- **Gen 0 and Gen 1 use non-concurrent GC**
  - UI thread pause time would be insignificant
- **Gen 2 can use concurrent**
  - Gen 2 can be expensive
  - Keeps UI thread pumping the message queue
- **Gen 2 can use asynchronous**
  - Means UI thread stays free-running even if another GC is required
  - Only available on .NET 4.0 and later

Workstation non-concurrent

- **Whole GC is non-concurrent**
  - Optimized for non-UI code
  - GC kidnaps allocating thread
  - Fewer CPU cycles than workstation concurrent

Server GC

- **Tuned for multiple processors**
  - Each processor gets own block of managed heap for allocation / GC called arena
  - Allows lock free allocation across multiple cores
- **Always non-concurrent**
  - GC takes place on all processors at the same time

GC Optimization – Self Tuning

- **GC behavior dependent on your application**
  - Uses heuristics to work out efficient strategy
  - If GC of a generation is effective then performs more
  - If GC of a generation reclaims little memory then performs less
- **GC self tuning normally produces optimal results**
  - Best strategy for your application
- **Avoid GC.Collect**
  - Disrupts GC statistics and so self tuning less effective

Large Objects

- **Moving large objects too expensive**
  - Large object over 85000 bytes
- **Allocated in Large Object Heap**

Memory is not the only Resource

- **Only memory management is automated**
  - Runtime does not know when use of other resources is complete
- **Types that control non-memory resources follow standard pattern**
  - Implement IDisposable
  - Consumer calls Dispose when they are finished

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

When Should You Implement IDisposable?

- **Most classes do not need to implement IDisposable**
  - Do not consume resources other than memory
- **Classes that directly control resources acquired via interop must implement IDisposable**
  - No other managed type can release those resources
- **Classes that aggregate other types that implement IDisposable should implement IDisposable themselves**
  - Provides a hook to Dispose aggregated objects

What if I don’t Call Dispose?

- **Only an issue for types that directly control unmanaged resources**
  - FileStream
  - SqlConnection
  - Icon
- **Need last chance mechanism for freeing up resources**
  - No one else can free up resources
- **Runtime has concept of Finalization**
  - Very unlikely that your classes will need a finalizer

How Finalization Works

- **Types that support finalization override Finalize virtual method on System.Object**
  - C# syntax looks like a C++ destructor

```
class SharedMemory
{
  ~SharedMemory()
  {
  }
}
```

- **Runtime sees type has finalizer when instantiated**
  - Puts object in finalization queue

Finalization and GC

- **When GC decides to collect object it sees it's finalizable**
  - Moves the object on to freachable queue
  - freachable queue acts as live root (**f**inalization **reachable**)
- **Separate finalization thread processes freachable queue**
  - Calls finalizer and removes object from queue
  - Object can be collected next time GC looks at it

Consequences of Finalization

- **Finalizable objects will always survive one GC**
  - Always promoted to gen1 or gen2
  - Consequently more expensive to collect
- **Timing of finalizer execution non-deterministic**
  - After GC notices it is garbage
  - When Finalization thread gets round to it
- **Limited functionality in finalizer**
  - Should not call other objects as may already have been finalized
- **Finalizable classes should always implement IDisposable**
  - Can suppress finalization in Dispose

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

Hindering and Helping the GC

- **Three main issues for GC**
  - Unnecessary promotion
  - Reference heavy data structures
  - Pinning

Unnecessary Promotion

- **Want to minimise Gen2 collections**
  - Ideally all Gen2 objects live for ever
  - Not realistic
- **Promote then die is worst scenario**
  - Badly tuned caches
  - Finalizers
  - “ad-hoc” caching
  - Not releasing dead objects

Unnecessary Promotion

- **Tune cache timeouts**
  - Drop cache in Gen1
  - Make it long lived to make the hit worthwhile
- **Remove unnecessary finalizers**
  - If absolutely required refactor code into SafeHandle

```
public abstract class SafeHandle : CriticalFinalizerObject,
                                   IDisposable
{
   ...
}
```

"Ad-hoc" Caching

"Probably need this object again soon so I'll hang on to it"

- **Consider WeakReference**
  - Object accessible unless GC'd

```
WeakReference wr = new WeakReference(new object());

object temp = wr.Target;
if(temp == null)
{
    Console.WriteLine("Collected");
}
else
{
    Console.WriteLine("Still alive");
}
```

Hanging on to Dead Objects

- **Set member and static references to null when done**
  - GC can clean up as soon as possible
- **Especially important when replacing object**
  - Be careful of threading issues

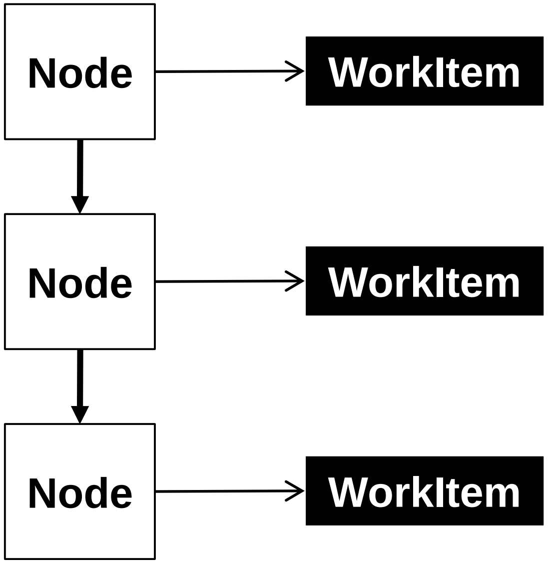
```
class ReferenceData
{
   XElement data;

   public void ReloadData(string filename)
   {
       data = null;
       data = XElement.Load(filename);
   }
}
```

Reference Heavy Structures

- **Mark phase made harder by many references**
  - GC has to chase down embedded object graphs
- **Think of revised or optimised ways of representing data**
  - Arrays
  - Adjacency Matrices
- **Example: Thread Pool Queue before 4.0**

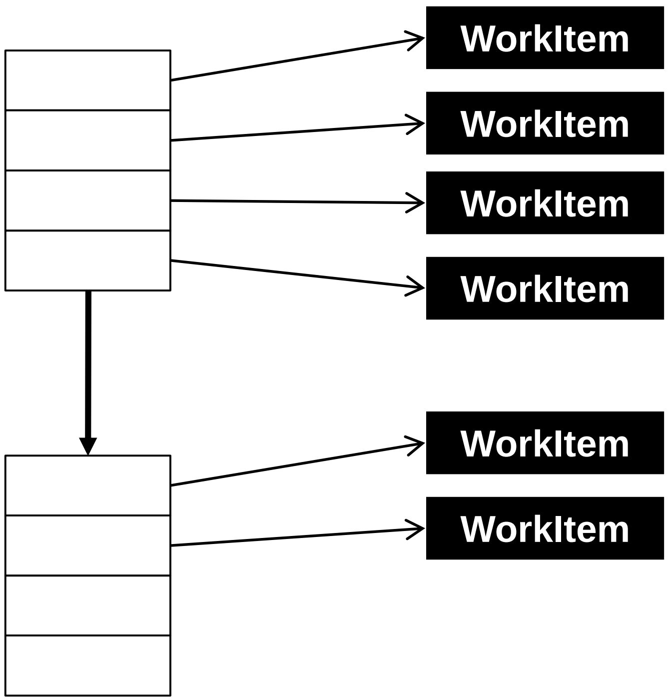
Thread Pool Queue Before 4.0

- **Linked list of work items**
  - Root is never garbage
  - Lots of work = lots of references



Thread Pool Queue After 4.0

- **Linked list of arrays of work items**
  - Lots of work = few references



Pinning

- **Pinning means GC can't move object**
  - Required in interop scenarios
- **Consider inflating large buffers to be allocated in LOH**
  - Pinning irrelevant in LOH
  - Reuse LOH buffers

Summary

- GC automates memory reclamation
- GC is heavily optimized
- Release mode GC much more aggressive than debug
- Non memory resources freed up using IDisposable pattern
- Finalization exists as a last resort for framework classes. You should not need it in your code
- You can help the GC
- You can hinder the GC