# Lifecycle Management in .Net

# Agenda

- The garbage Collector
- Memory Leaks
- Weak References
- Finalization

# THE GARBAGE COLLECTOR

AKA

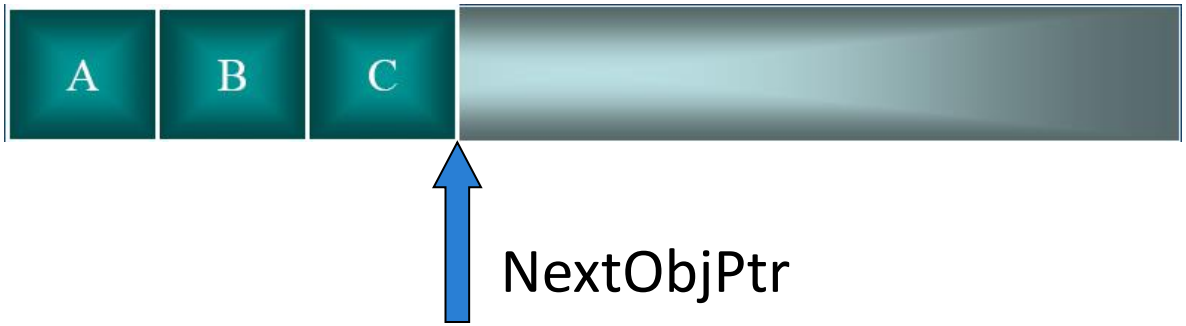
Managed Heap

# Memory Overview

- Every program uses memory
  - Files, memory buffers, screen space, network connections, database resources, and so on.
- A class identifies some type of resource
- Usual programming paradigm
  - Allocate memory for resource (new operator)
  - Initialize memory to make resource usable (constructor)
  - Use the resource (access members) – repeat as necessary
  - Tear-down the resource (destructor)
  - Free the memory
- Common problems
  - Forgetting to free memory → **memory leak**
  - Using memory after freeing it → **memory corruption**
- These bugs are worse than most other bugs
  - Consequence and timing is unpredictable

# Solution

- GC makes these bugs a thing of the past
- All reference types are allocated on the managed heap
  - Your code never frees an object
  - The GC frees objects when they are no longer reachable
- Each process gets its own managed heap
  - Virtual address space region, sparsely allocated
- The new operator always allocates objects at the end
- If heap is full, a GC occurs
  - Reality: GC occurs when generation 0 is full



# The GC Algorithm Overview

- Allocating objects:
  - new operator always allocates objects at the end
  - Almost as fast as a stack allocation
  - Much faster than unmanaged new/malloc/HeapAlloc
  - Large objects are allocated from a special heap
    - Large objects aren't moved in memory
    - Large objects are >= 85,000 bytes (*may change*)
- Garbage Collection:
  - Marking:
    - Objects referred to by app's variables (the Roots) are marked.
  - Compacting:
    - Marked objects are shifted down over unmarked objects
  - If all objects are marked, no compacting occurs
    - new throws OutOfMemoryException

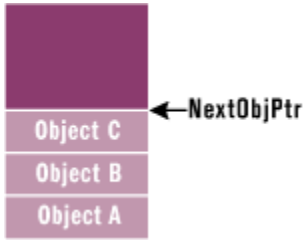
This algorithm is called: "***Mark and Compact***"

# Allocating Memory

- When a process is initialized, the runtime reserves a contiguous region of address space that initially has no storage allocated for it.
  - This address space region is the managed heap.
  - The heap also maintains a pointer, which I'll call the NextObjPtr. This pointer indicates where the next object is to be allocated within the heap.
  - Initially, the NextObjPtr is set to the base address of the reserved address space region.

- An application creates an object using the new operator.
  - This operator first makes sure that the bytes required by the new object fit in the reserved region (committing storage if necessary).
  - If the object fits, then NextObjPtr points to the object in the heap, this object's constructor is called, and the new operator returns the address of the object.

- At this point, NextObjPtr is incremented past the object so that it points to where the next object will be placed in the heap.




**Managed heap**

***This allocation method is much faster than raw memory allocation as in C++***

# .Net Garbage Collection

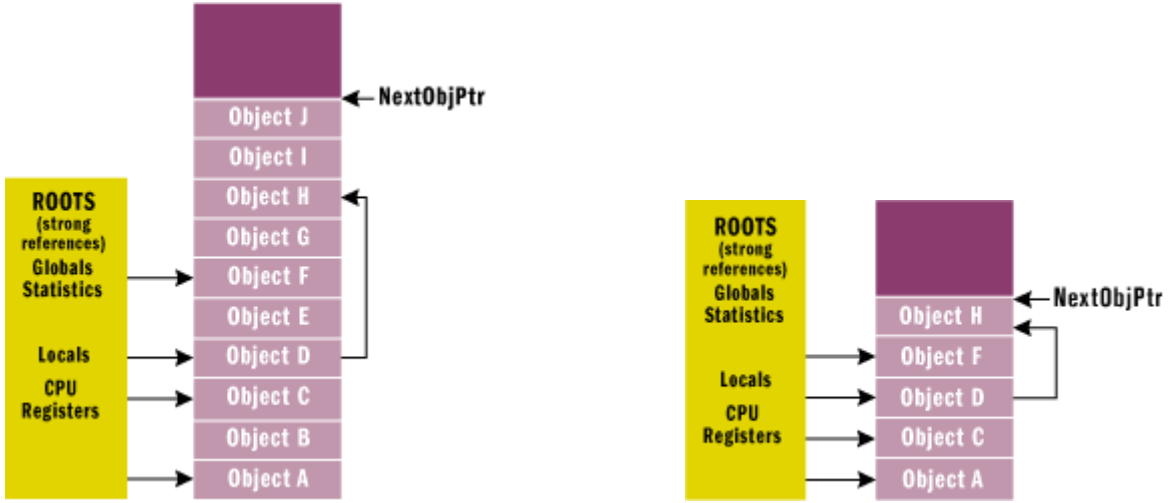
- The garbage collector checks to see if there are any objects in the heap that are no longer being used by the application.
  If such objects exist, then the memory used by these objects can be reclaimed!
- How does the garbage collector know if the application is using an object or not?

# Using Roots to Mark Objects

- When a garbage collection starts all objects in heap are considered garbage
- Marking Phase:
  1. Marks objects reachable from Roots
     - A Root is a memory location that can refer to an object (or be null)
       - Static fields defined within a type
       - Arguments passed to a method
       - Local variables declared within a method
       - CPU registers (enregistered fields, arguments, or variables)
     - Roots are always reference types (never value types)
     - Each method has a root table (produced by JIT compiler)
  2. Each marked object has its fields checked; these objects are marked too (recursively)
  3. GC walks up the thread's call stack determining roots for the calling methods by accessing each method's root table
     - Already marked objects are skipped
       - Improves performance
       - Prevents infinite loops due to circular references
       - Static fields are checked; these objects are marked too

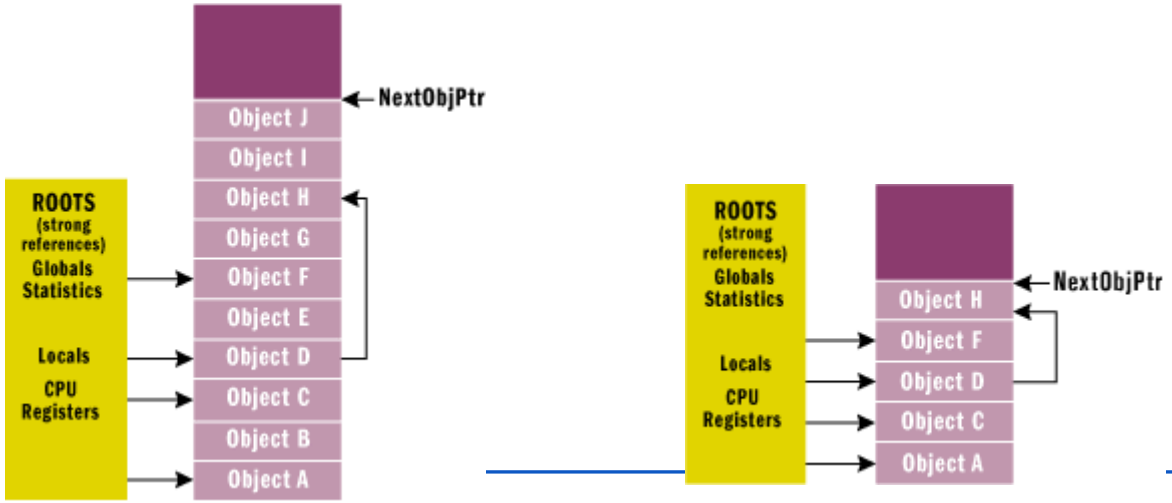
# Collecting Garbage 1

- When the garbage collector starts running, it makes the assumption that all objects in the heap are garbage.
- Now, the garbage collector starts walking the roots and building a graph of all objects reachable from the roots.
- Any objects that are not in the graph are not accessible by the application, and are therefore considered garbage.
- The garbage collector now walks through the heap linearly, looking for contiguous blocks of garbage objects (now considered free space).
- The garbage collector then shifts the non-garbage objects down in memory (using the standard memcpy function), removing all of the gaps in the heap.



# Collecting Garbage 2

- Moving the objects in memory invalidates all pointers to the objects.
  - So the garbage collector must modify the application's roots so that the pointers point to the objects' new locations.
  - In addition, if any object contains a pointer to another object, the garbage collector is responsible for correcting these pointers as well.
- ~~The structure of the object graph must not change during GC, so all the applications threads are suppended during GC!~~ Not in .Net v. 4.0.
  - If a thread was in the middle of modifying a linked list of object (or similar) then it might introduce an error when resumed.
  - To handle this problem, the JIT compiler inserts safe points into the code, meaning points where it is safe to suspend a thread.
- GC generates a significant performance hit! (approx 5% CPU time).
  - this is the major downside of using a managed heap



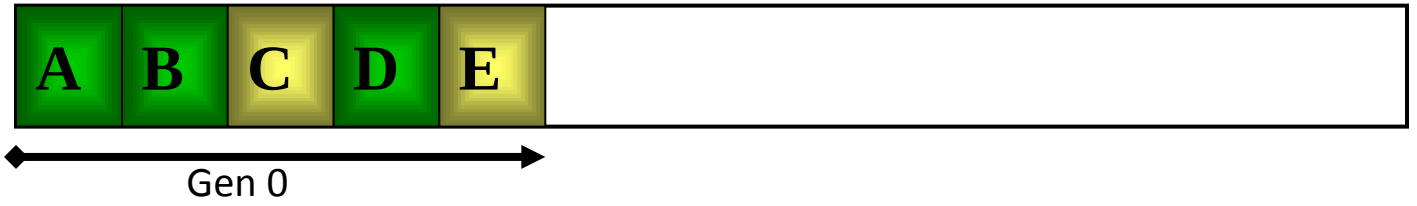
# Generations

- Makes assumptions about your code
  - The newer an object is, the shorter its lifetime will be
    - A garbage collection is likely to free a lot of memory
  - The older an object is, the longer its lifetime will be
    - A garbage collection is unlikely to free a lot of memory
  - New objects have a strong relationship are accessed together
    - Allocated adjacent: locality of reference, reduced working set
- Studies show that assumptions are valid for many apps
- Generational GC improves perf by collecting new objects only
  - Old objects are not marked and walked recursively
  - Only new surviving objects are compacted
    - Only new objects' roots need updating
- .Net has 3 generations – Gen. 0 is the newest.

***Improves Performance!***

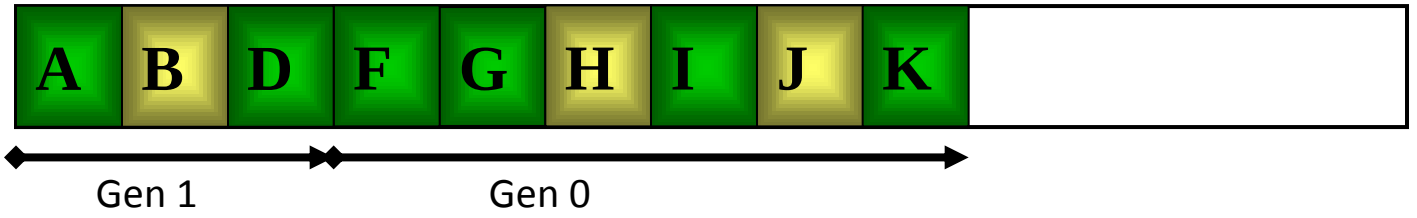
# Generational GC 1

- Initially, the heap is empty. New objects go in Gen 0



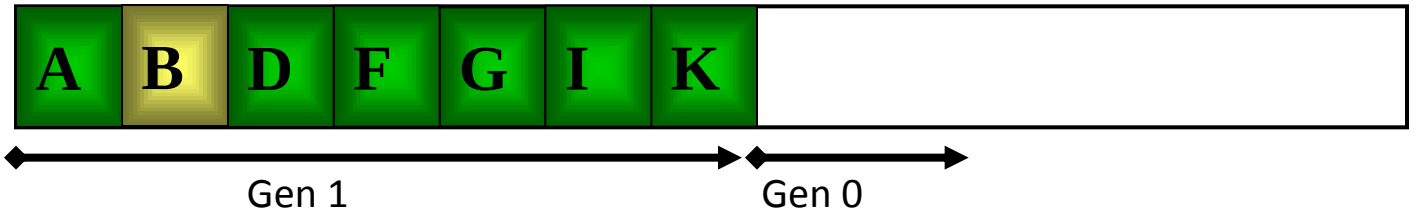

- Gen 0 has 256KB, GC starts. Survivors → Gen 1


- New objects go in Gen 0

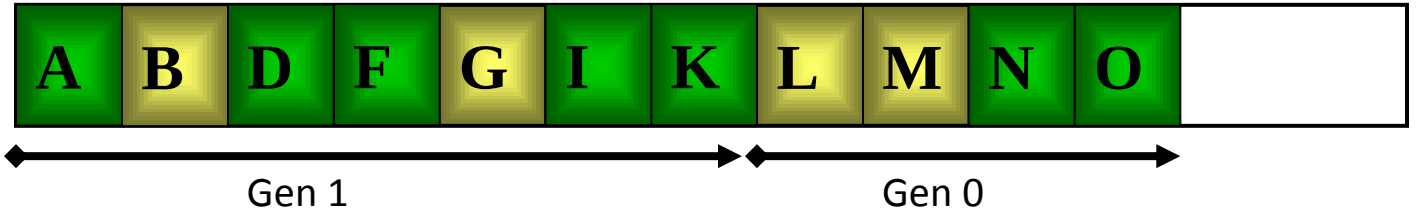


# Generational GC 2

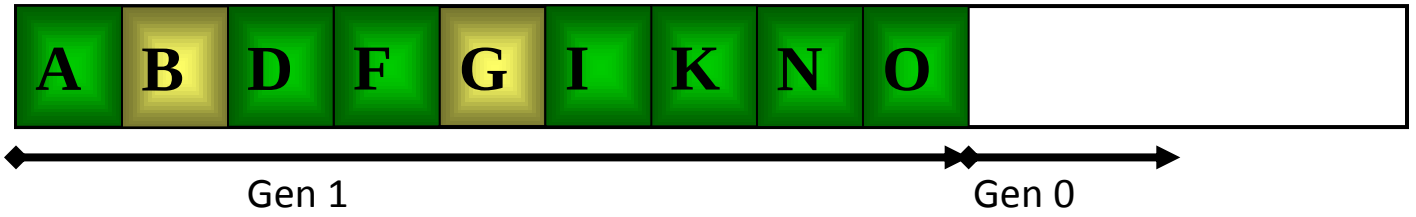
- Gen 0 has 256KB, GC starts. Survivors → Gen 1 (grows)




- New objects go in Gen 0

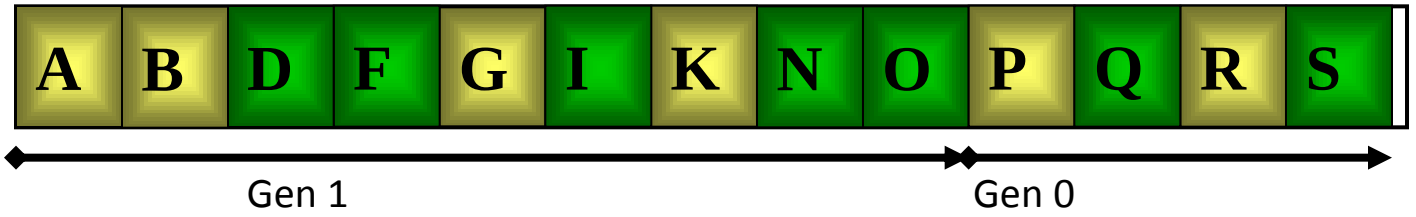



- Gen 0 has 256KB, GC starts. Survivors → Gen 1 (grows)

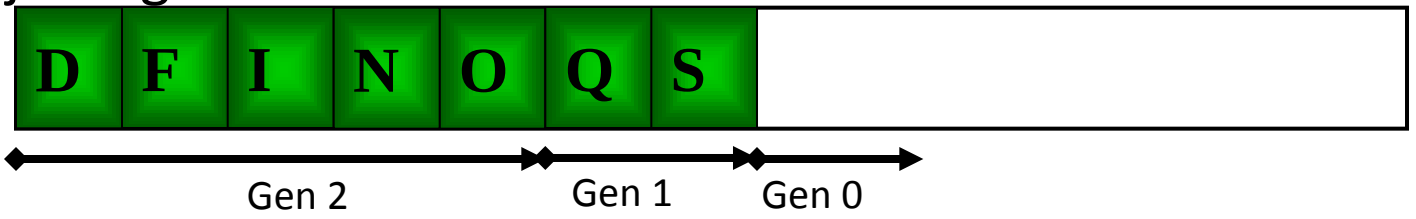


# Generational GC 3

- New objects go in Gen 0




- Gen 0 has 256KB, GC starts
  Since Gen 1 > 2MB, Gen 1 & 0 are collected
  Gen 1 survivors → Gen 2, Gen 0 survivors → Gen 1
  New objects go in Gen 0



# How Generations Improve Performance

- When Gen 0 (~256KB) fills a GC starts
  - 256 because of CPU L2 cache
- The GC must decide what generations to collect
  - If Gen 2 > ~10MB, collect Gen 2, 1, & 0
    - Else if Gen 1 > ~2MB, collect Gen 1 & 0
      - Else Collect Gen 0
- Most collections collect only Gen 0!
  - Most new objects are garbage & enough memory is reclaimed
  - Only objects in Gen 0 have to be examined
  - If root/object refers to older object, walking older objects is not necessary reducing time to build graph
  - Works great for apps that have waiting threads with short stacks
    - Windows Forms, WPF, Web Forms, Web Services, etc.
    - Event causes new objects, all garbage immediately!
- The GC is self tuning → the threshold values are adjusted to optimize performance for the current machine and application!

# No more Memory Problems?

- Does the garbage collector solve all memory problems?
- **NO!**
- The GC does a fine job but:
  - Memory leaks, and
  - Excessive(unneeded) memory usage

  **is still possible!**

- To discover these you use profilers.

# MEMORY LEAKS

# The Profile Of A Leaking Application

- Memory usage slowly increases over time
- Performance degrades
- Application will freeze/crash requiring a restart
- After restart, the application runs without problems again, and the cycle repeats.
- How can we get memory leaks?
  - Memory leaks occur when a reference is retained to an object of which instances are created frequently.
  - Finding a memory leak is simply a matter of finding the object that is being retained, and then determining which reference in your code is causing it to be retained.

# Typical Memory Leak

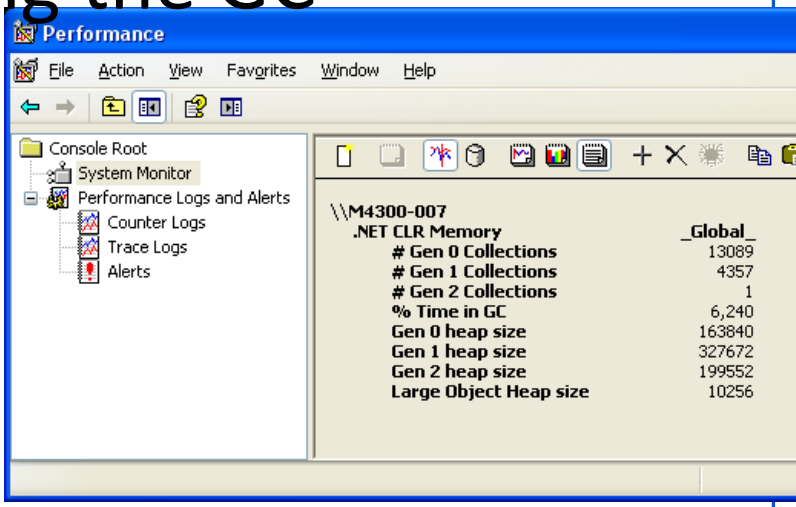
```
// Each Order subscribes to the OnPriceUpdate event
Order newOrder = new Order("EURUSD", DealType.Buy, Price,
                     PriceTolerance, TakeProfit, StopLoss);
newOrder.OnPlaced += OrderPlaced;
m_Currency.OnPriceUpdate += newOrder.OnTick;
m_PendingDeals.Add(newOrder);

// When the price is right an Order completes
void OrderPlaced(Order placedOrder)
{
  m_PendingDeals.Remove(placedOrder);
  m_ActiveDeals.Add(placedOrder);
}
```

***Can you spot the memory leak?***

The OrderPlaced method is missing one line which is needed to avoid a memory leak:

```
m_Currency.OnPriceUpdate -= placedOrder.OnTick;
```

# Tools for Monitoring the GC




- **PerfMon**
  - Graphs many .NET-related counters
  - Comes with Windows

- **CLR Profiler**
  - Download from http://www.microsoft.com/download/en/confirmation.aspx?id=16273

- **EQATEC Profiler** *"use our free profiler to spot you app's slow code"* http://www.eqatec.com/tools/profiler


- Plus several commercial products like:
  - Red Gates **Ants Profiler**: http://www.red-gate.com/
  - JetBrains **dotTrace:** **http://www.jetbrains.com/profiler/index.html**

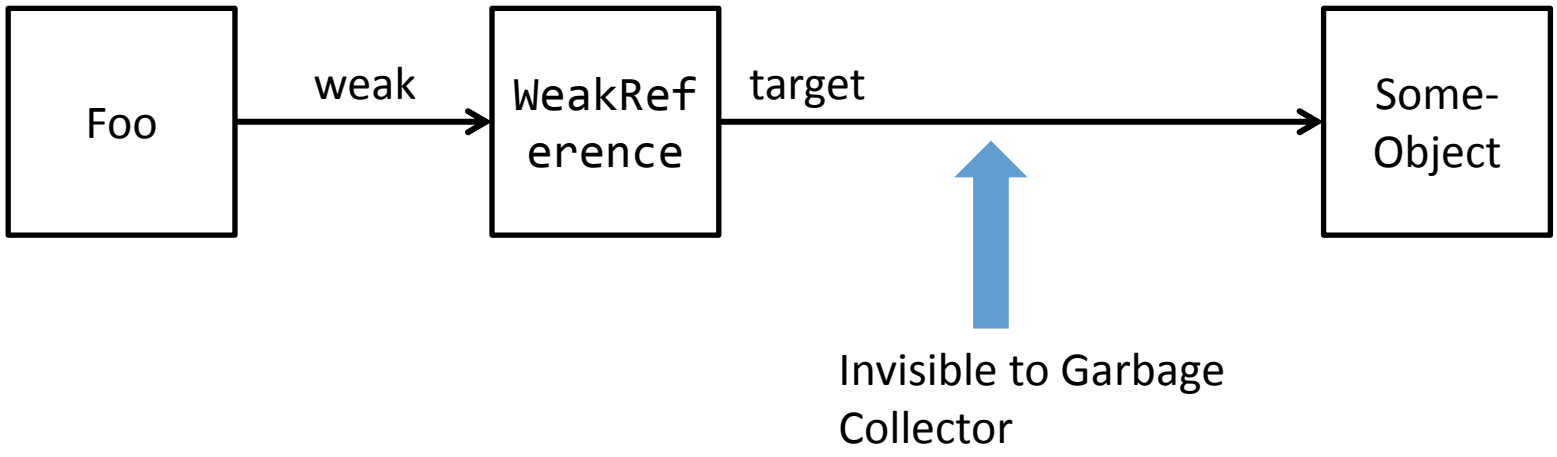
# WEAK REFERENCES

When and How to Use Them

# The WeakReference Class

- An instance of the WeakReference class references an object while still allowing that object to be reclaimed by garbage collection.

```
WeakReference weak = new WeakReference(someObject);
```



# WeakReference Dummy Demo

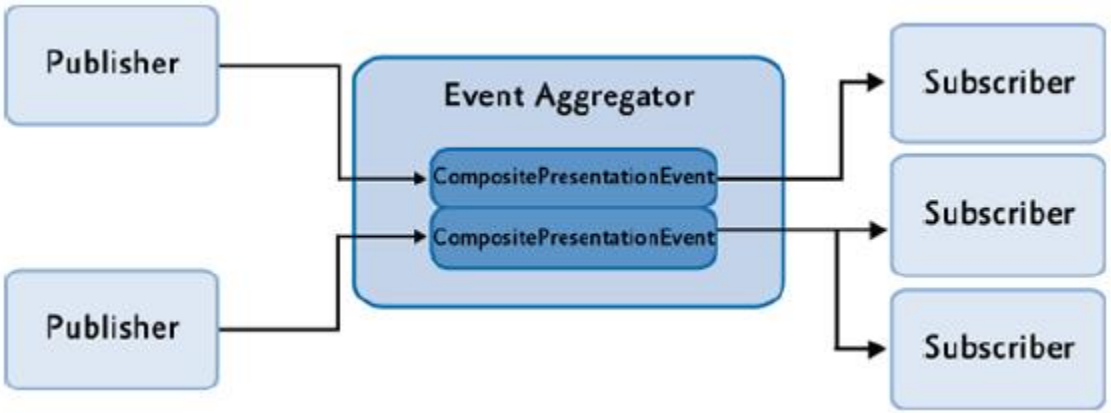
```
static WeakReference _weak;

static void Main() {
// Assign the WeakReference.
_weak = new WeakReference(new StringBuilder("Dummy still alive"));

// See if weak reference is alive.
if (_weak.IsAlive) {
    Console.WriteLine((_weak.Target as StringBuilder).ToString());
}

GC.Collect();

// Check alive.
if (_weak.IsAlive) {
    Console.WriteLine((_weak.Target as StringBuilder).ToString());
}
```

# Using WeakReferences

- The WeakReference class has two main use cases:
  1. Loosely coupled events
     - In Prism the EventAggregator uses WeakReferences.
     - In MVVMLight the Messenger uses WeakReferences.

  2. Cache of objects that can be recreated if needed:
     - Weak references may be useful for objects that use a lot of memory, but can be recreated easily if they are reclaimed by garbage collection.

# Communications Between Loosely Coupled Components In Prism




- The event aggregator allows publishers and subscribers to communicate through events and still do not have a direct reference to each other.
- By default, CompositePresentationEvent maintains a **weak delegate reference** to the subscriber's handler.
  - This means the reference that CompositePresentationEvent holds on to will not prevent garbage collection of the subscriber.
  - Using a weak delegate reference relieves the subscriber from the need to unsubscribe and allows for proper garbage collection.
- Maintaining this weak delegate reference is slower than a corresponding strong reference.
  - For most applications, this performance will not be noticeable, but if your application publishes a large number of events in a short period of time, you may need to use strong references with CompositePresentationEvent.

# Finalization
# and
# Dispose

Aka **Implicit and Explicit Cleanup**

# Finalization

- What about non-managed ressources?
- The garbage collector offers an additional feature that you may want to take advantage of: finalization.
- Finalization allows a resource to gracefully clean up after itself when it is being collected.
- By using finalization, a resource representing a file or network connection is able to clean itself up properly when the garbage collector decides to free the resource's memory.
- Here is an oversimplification of what happens:
  - when the garbage collector detects that an object is garbage, the garbage collector calls the object's Finalize method (if it exists) and then the object's memory is reclaimed.

# Finalization in C#

- In C# you are not allowed to write the Finalizer yourself, but must write what looks like a destructor:

```
class Foo    {
  ~Foo()
  {
    Console.WriteLine("In destructor");
  }
}
```

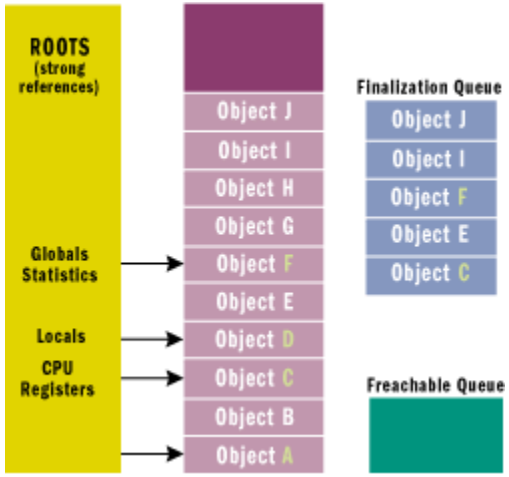
- When the compiler sees this syntax, it turns it into this code :

```
class Foo {
  protected override void Finalize()
  {
    try
    {
      Console.WriteLine("In destructor");
    }
    finally
    {
      base.Finalize();
    }
  }
}
```

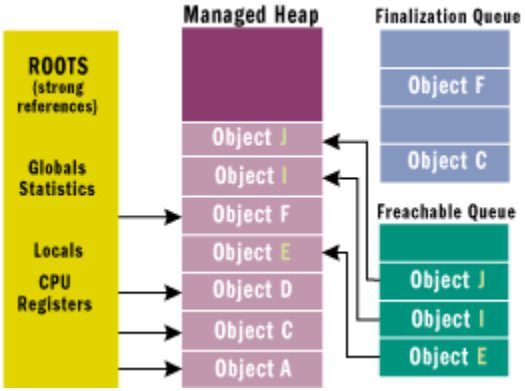
# Finalization in C#

- It is easy to prove that the **Finalize** method is not called in any particular order.
  - This could cause a problem if your object contains other objects. It is possible that the inner object could be finalized before the outer object.
- If you implement a **Finalize** method, don't access contained members!!!
- Adding a **Finalize** method to a class can cause allocations of that class to take significantly longer.
  - When the C# compiler detects that your object has a **Finalize** method, it needs to take extra steps to make sure that your **Finalize** method is called.

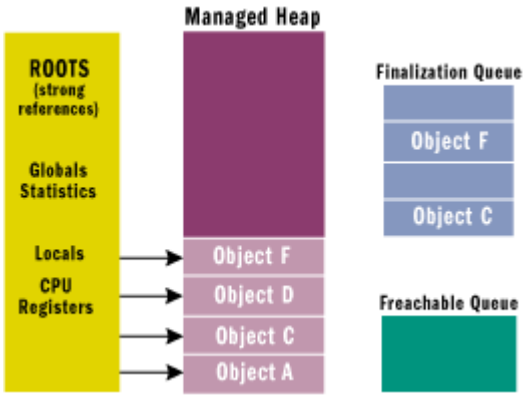
# Finalization Internals



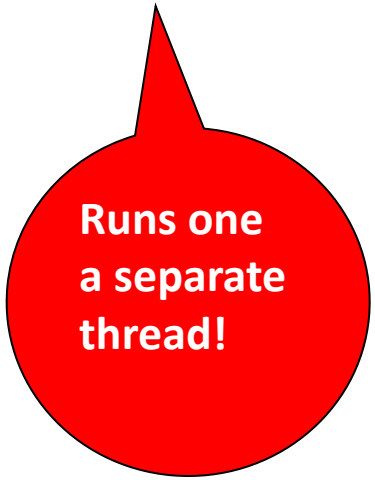

Managed Heap before Garbage Collection




Managed Heap after Garbage Collection




Managed Heap after Second Garbage Collection



# Don’t wait for the GC to Finalize

- Do the clean up your self.
- Use **Explicit Cleanup**
  - Use the Open(), Close() pattern (where appropriate)
  - Use the Dispose() pattern
  - **Use the IDisposable() pattern**

# Managing Resources with a Disposable Object

- A **Finalize** method is meant to be part of a scheme to support deterministic finalization, not as a solution on its own.
- The scheme involves implementing an **IDisposable** interface and uses **Finalize** as a backup in case the object is not explicitly finalized as expected.
- To make an **IDisposable** object, you need to derive from the **IDisposable** interface.
  - This ensures that you implement the proper methods
  - and provides a means of discovering whether this object supports the deterministic finalization model by querying the object for the **IDisposable** interface.

# Implementing IDisposable

```
public class DisposeObject : IDisposable    {
     private bool disposed = false;
     public DisposeObject()       {
        // Do some init…
     }
     public void Close()       {
        Dispose();
     }
     public void Dispose()       {
        // Dispose managed and unmanaged resources.
        Dispose(true);
        GC.SuppressFinalize(this);
     }
     protected virtual void Dispose(bool disposing)       {
        // Check to see if Dispose has already been called.
        if (!this.disposed)          {
           // If disposing equals true, dispose all managed
           // and unmanaged resources.
           if (disposing)             {
              // Dispose managed resources.
           }
           // Release unmanaged resources. If disposing is false,
           // only the following code is executed.
        }
        // Make sure that resources are not disposed
        // more than once
        disposed = true;
     }
     ~DisposeObject()       {
        Dispose(false);
     }
}
```

**This code is NOT thread safe!**

**In multithreaded apps use the "The Ultimate Dispose Pattern Guide" as template**

# Using a Disposable Object

- With C#, it is possible to use an object that implements an **IDisposable** interface and be assured that the Dispose method is called on that object when execution passes beyond the scope that is defined by the **using** keyword:

```
using(f)
{
  // Use the object
}
```

- This is roughly equivalent to something like this:

```
IDisposable d = (IDisposable)f
try
{
    // Use the object
}
finally
{
    d.Dispose();
}
```

# References


- **Fundamentals of Garbage Collection**
  http://msdn.microsoft.com/en-us/library/ee787088.aspx

- The Ultimate Dispose Pattern Guide
  http://www.bluebytesoftware.com/blog/PermaLink.aspx?guid=88e62cdf-5919-4ac7-bc33-20c06ae539ae

- How to detect and avoid memory and resources leaks in .NET applications
  http://msdn.microsoft.com/en-us/library/ee658248.aspx

- **SafeHandle**
  http://www.codeproject.com/KB/dotnet/safehandle.aspx

- What Your Mother Never Told You About Resource Deallocation
  http://www.codeproject.com/Articles/29534/IDisposable-What-Your-Mother-Never-Told-You-About