PXL IT

Garbage Collection

.NET

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Dep. PXL-IT – Elfde-Liniestraat 26 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

This lecture is based on the following Pluralsight Course:
http://www.pluralsight.com/courses/making-dotnet-applications-even-faster

Contents

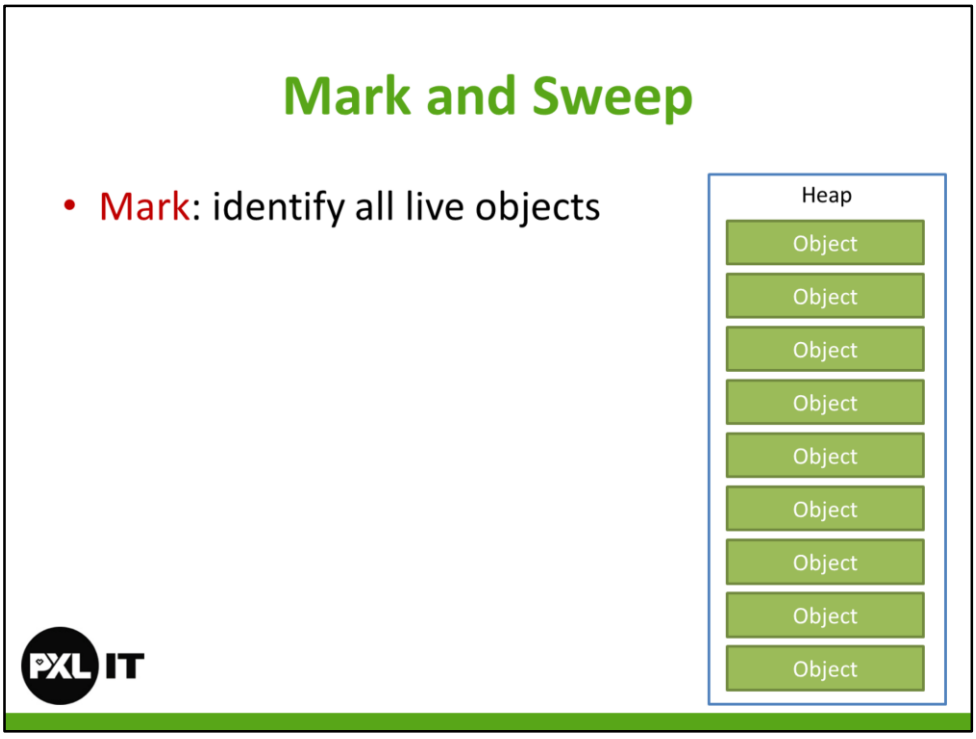
- Basis GC concepts
- GC Flavors
- Generations
- Finalization

Garbage Collection

- GC means: you don't have to manually free memory
- GC isn't free and has performance trade-offs
  - Questionable on real-time systems, mobile devices, etc.
- The CLR garbage collector (GC) is an almost-concurrent, parallel, compacting, mark-and-sweep, generational, tracing GC

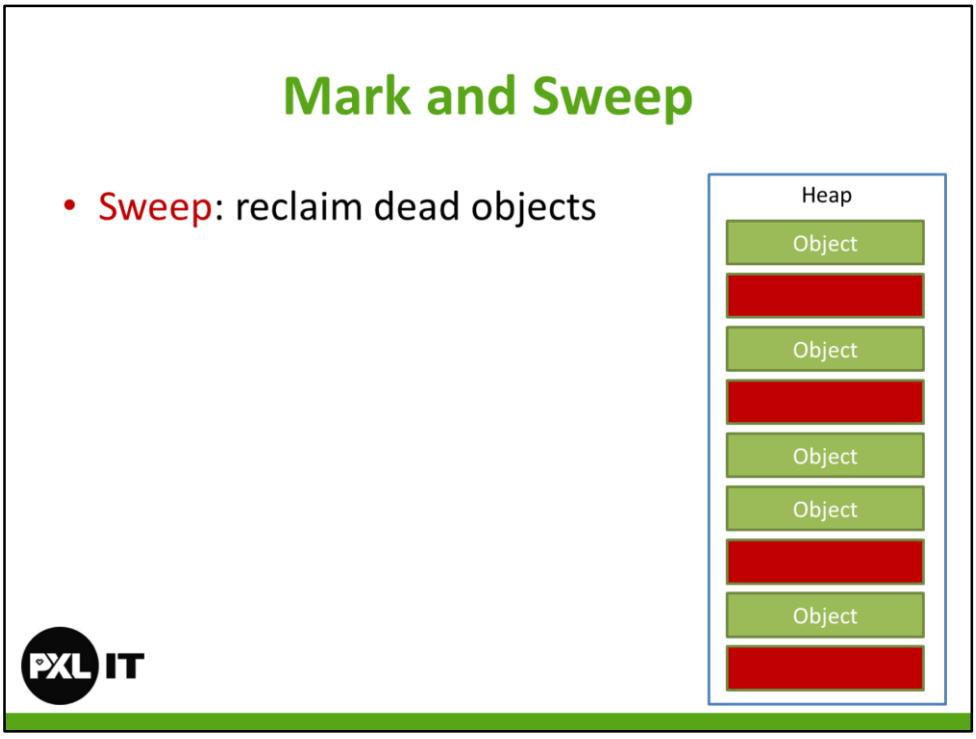
The GC is a run-time component that runs together with your program. It solves an **extremely difficult problem** of identifiying unused objects (or resources in general) and freeing up memory for new allocations.

We will explain the different properties of the GC CLR in the following slides.



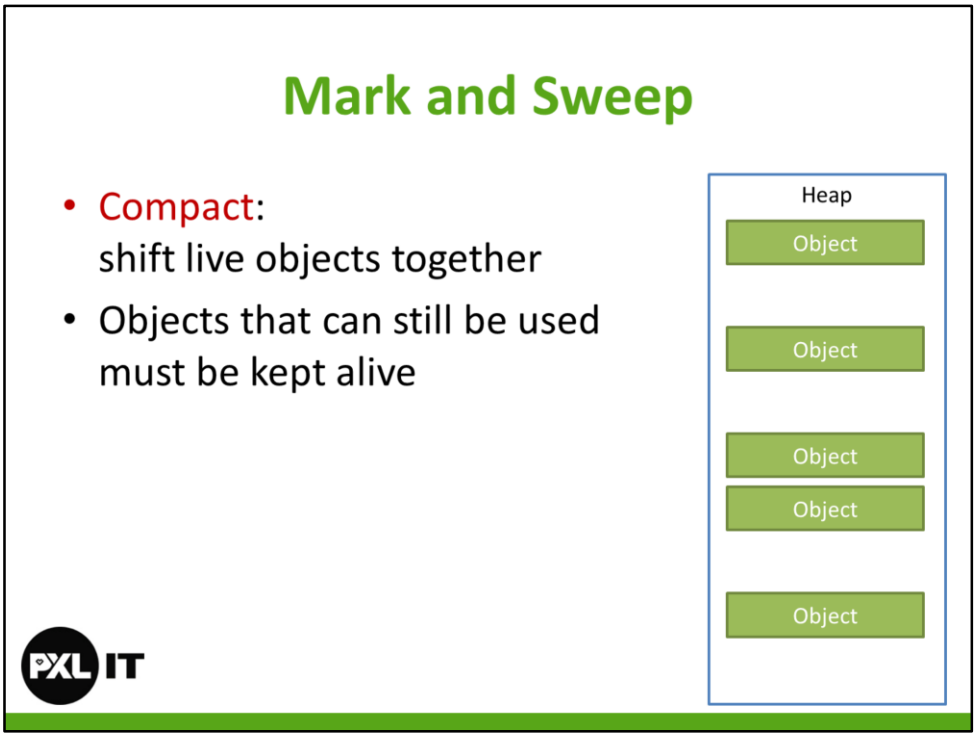
Mark and sweep are the two fundamental actions performed by the garbage collector when it is required to reclaim memory.

In the **mark phase**, the garbage collector identifies all live objects in the heap, and this is done recursively. The GC starts from a group of objects that are certainly live and follows references between the objects until all the referenced objects are explored. The GC keeps track of objects it has already visited to avoid getting into an infinite loop if there is a reference cycle between objects.



Mark and sweep are the two fundamental actions performed by the garbage collector when it is required to reclaim memory.

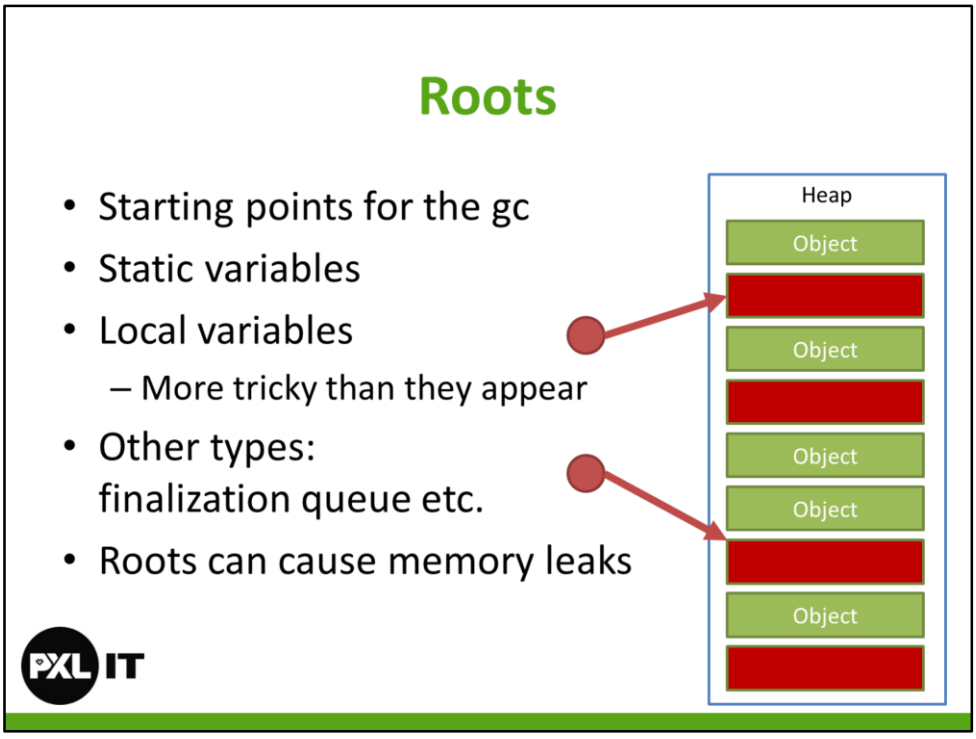
In the **sweep phase**, the garbage collector reclaims all unused memory that's occupied by dead objects.



Mark and sweep are the two fundamental actions performed by the garbage collector when it is required to reclaim memory.

The sweep phase often ends with a **compaction** phase. Compaction means the GC shifts live objects together so that they are consecutive in memory. The free space is then also consecutive, and this can make allocations cheaper.

But how does the garbage collection determine which objects are okay to die and which objects have to remain in memory. It all boils down to whether the program is still using these objects. Any object that the program still has a chance of using in the future has to remain, and it cannot be reclaimed by the GC. Any object that's completely unreachable can be reclaimed, and its memory can be used for future allocations. The problem, then, is that identifying those dead objects is hard, because they are unreachable, and the mark phase really is all about finding the live objects. The rest of the heap are considered dead objects by definition.

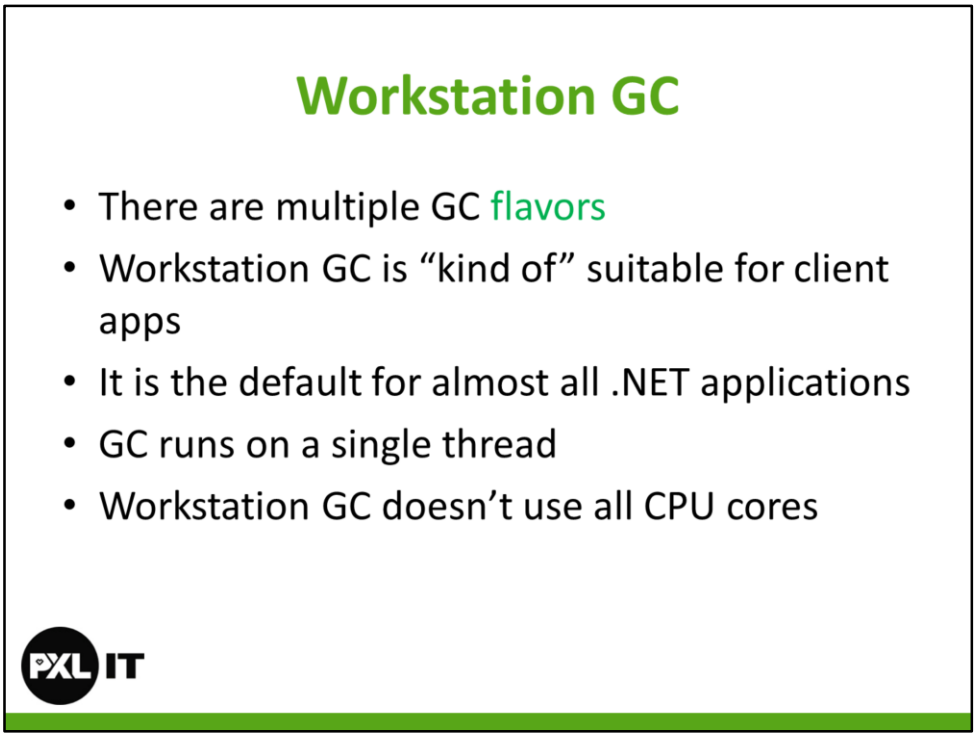


To find live objects, the garbage collector has to start somewhere. Some objects have to be considered alive even if they're not referenced by other objects. References to these root objects are called roots, and there're multiple kinds of roots in a .NET program.

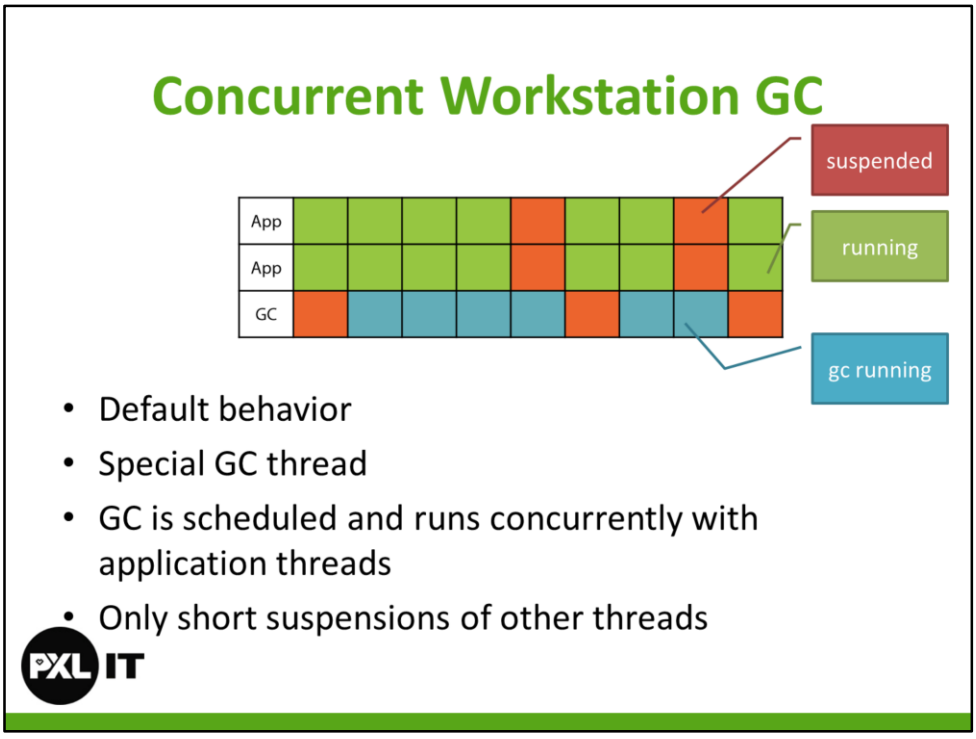
Demo: Mod01_GCRoots

Conclusion: use GC.KeepAlive(obj) to manually extend the lifetime of an object.

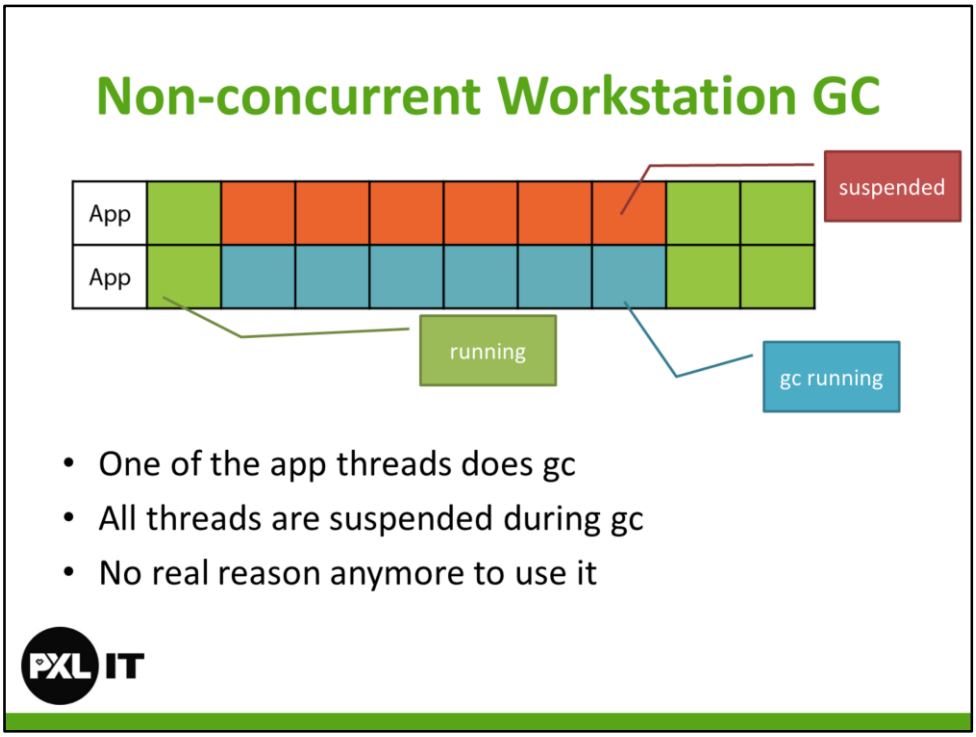
https://msdn.microsoft.com/en-us/library/system.gc.keepalive(v=vs.110).aspx



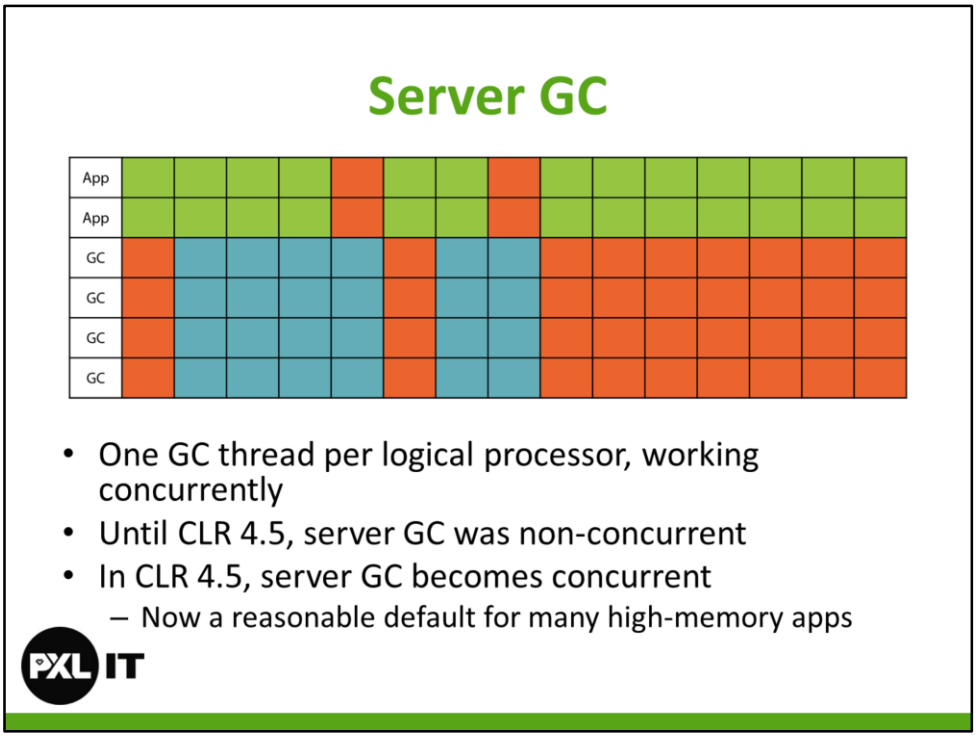
The biggest problem with workstation GC today is that it does not take advantage of modern hardware: it runs on a single thread and doesn’t use all CPU cores.



Let's look at concurrent workstation GC first, because, again, that's the default. In concurrent workstation GC, the CLR creates a special GC thread as soon as your application starts running, and this thread monitors the GC heap and performs garbage collections occasionally. It doesn't wait until all memory's exhausted, which would be wasteful. It tries to schedule garbage collections in a way that takes lots of memory and reclaims it without badly affecting application performance. The key here is that the GC thread runs concurrently with the application threads most of the time. Most parts of the garbage collection process can be done in the background while the other threads are running. Occasionally, the GC thread does have to suspend all the applications threads, but these suspensions are usually very short. And, as a result, if you have lots of memory allocations in your app, the GC thread will work hard in the background and only occasionally stop your other threads.



In non-concurrent workstation GC, there is no special GC thread. One of the application threads is picked out to do the GC when it performs an allocation that would trigger a garbage collection. And that application thread, of course, doesn't do anything else during GC. But neither do the other threads. Non-concurrent GC actually has to suspend all the application threads for the duration of the GC. And that probably sounds like a bad idea, because in most cases, it is a bad idea. If the GC takes two seconds, your whole app stops responding for two seconds.



With server GC, the main difference is that there are multiple GC threads. The CLR creates a GC thread for each logical processor in fact. So, for example, if you're on a quad core i7 processor with hyper-threading enabled, Windows thinks you have eight logical processors, and so your .NET app will have eight GC threads. When a GC is needed, these threads are just all awakened at once and start churning away. Garbage collection can be parallelized quite well, so you'd see considerable speedups in most cases if you switched to server GC. To further improve GC performance, server GC also has a separate heap area for each logical processor. So when a thread running on processor 0 allocates memory, it doesn't contend with another thread that happens to be running on processor #1 and also allocating at the same time. And this also improves cache locality for those threads because objects allocated closely in time will also be close together in memory. Now the only practical reason to avoid server GC used to be the fact that it didn't have a concurrent flavor. Until CLR 4.5, server GC was non-concurrent, which means all the application threads were blocked while doing garbage collection, and that could introduce unacceptable delays, especially in client applications again. But even so, it still sometimes made sense to switch to server GC in a client application just because it speeds up the overall collection process so much. If server GC turns a one-second collection into a 200-msec collection speeding it up by a factor of five, it might be worth the pain of suspending all app threads during the collection. In CLR 4.5, however, there now is a concurrent server GC flavor, and that's the best of both worlds. You have multiple threads doing GC, and your application threads can keep

working at the same time. So, if you have an application that spends a lot of time in the garbage collector, I think it's a pretty good idea that you should check server GC out.

Switching GC Flavors

- Configure preferred flavor in app.config
  - Ignored if invalid (e.g. concurrent GC on CLR 2.0)

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <gcServer enabled="true|false" />
    <gcConcurrent enabled="true|false" />
  </runtime>
</configuration>
```

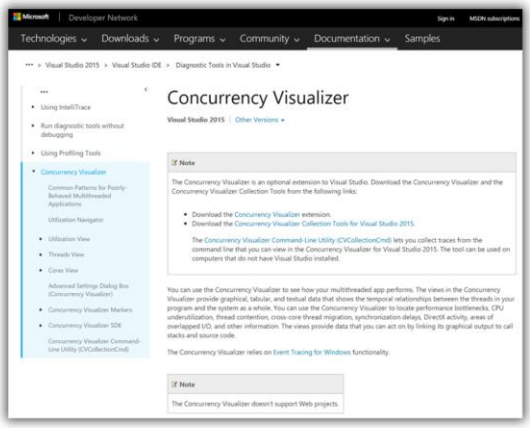
- You can't switch flavors at runtime
  - You can query flavor using GCSettings class

GCSettings

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

Demo: Mod01_GCFlavors

- First, install Concurrency Visualizer

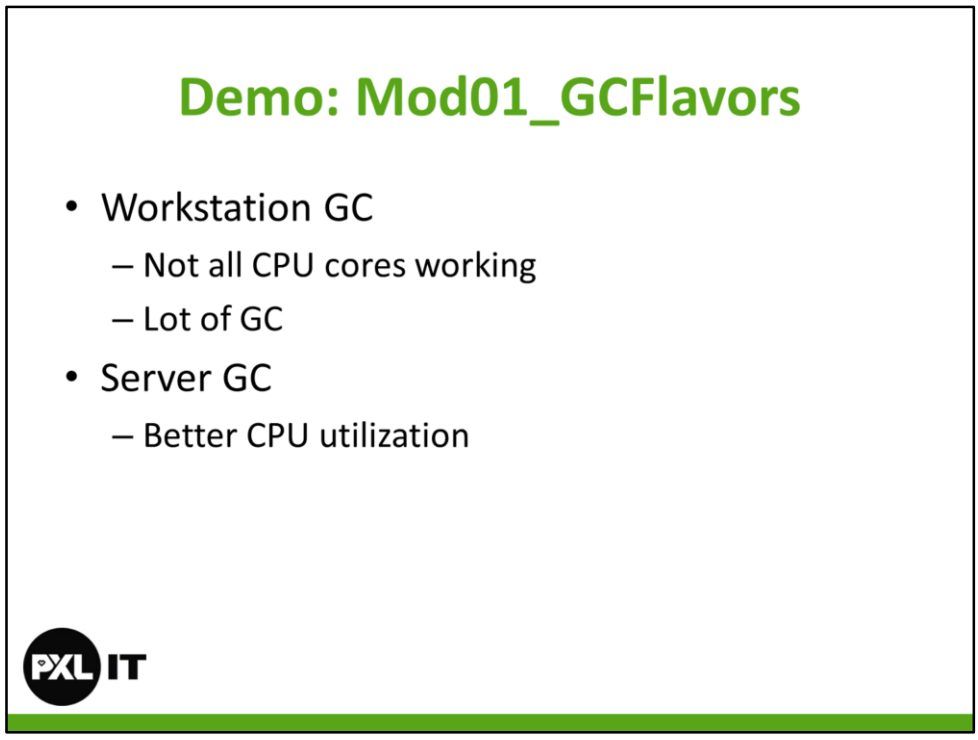


PXL IT

You can download Concurrency Visualizer from:
https://msdn.microsoft.com/en-us/library/dd537632.aspx

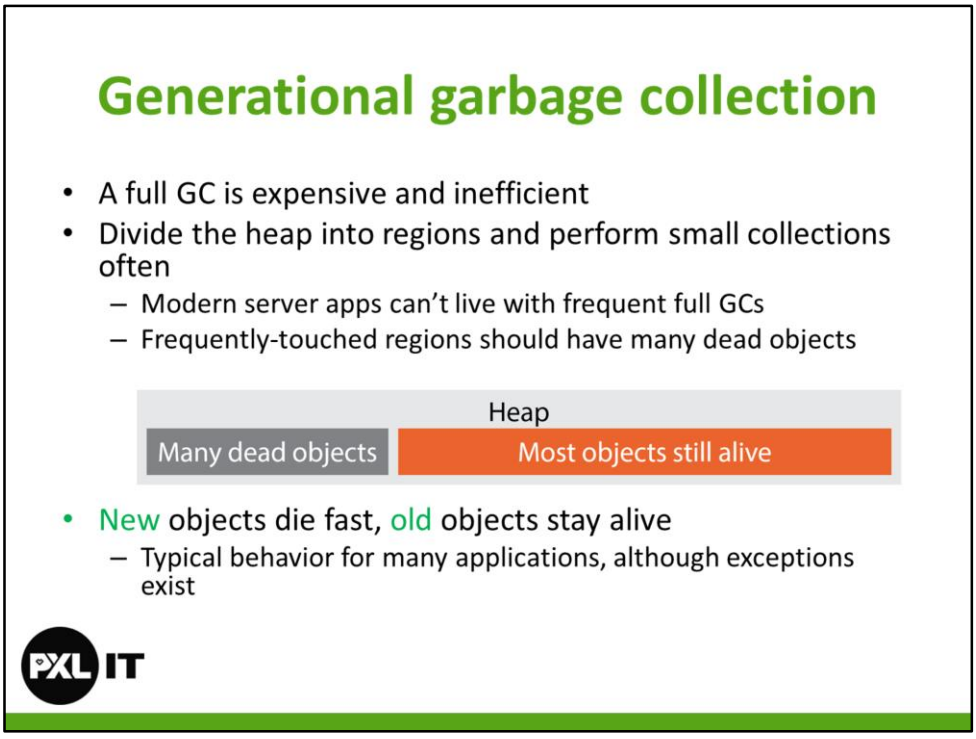
Download and install both:

- The extension for VS
- The collection tools



Demo: Mod01_GCFlavors

Look at the following demo (starting 8:00) on pluralsight: http://www.pluralsight.com/training/player?course=making-dotnet-applications-even-faster&author=sasha-goldshtein&name=making-dotnet-applications-even-faster-m1-gcinternals&clip=3&mode=live



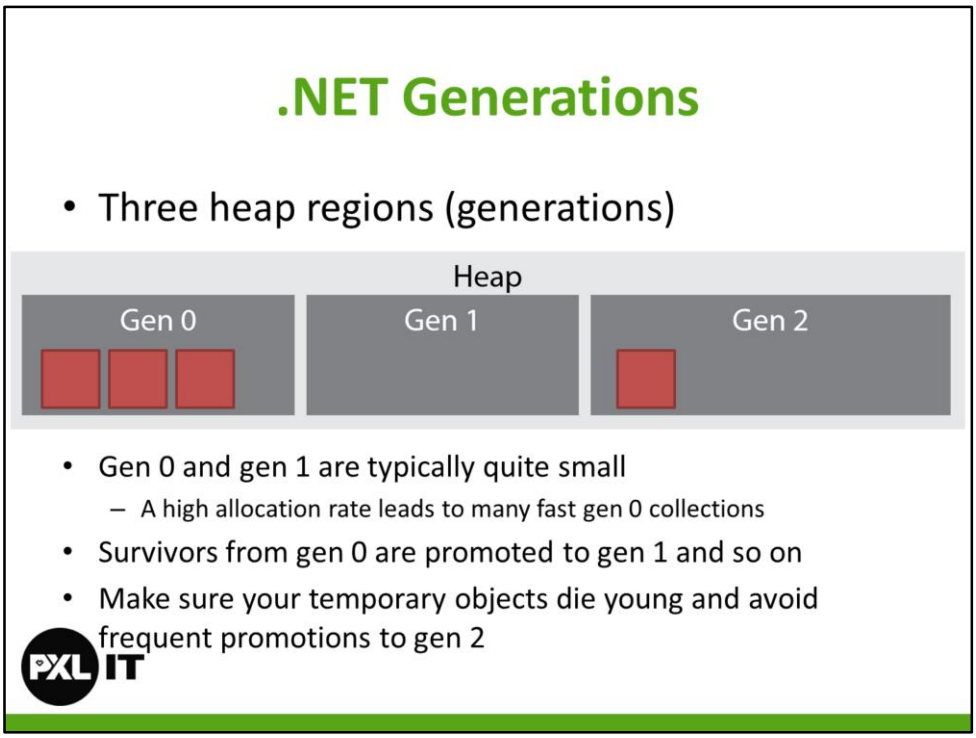
One Big Heap is not efficient: try to avoid full gc.

Divide into regions => many dead objects (gc first) and mostly alive objects (gc not usefull)

Observation: **new** objects die fast, **old** objects stay alive.

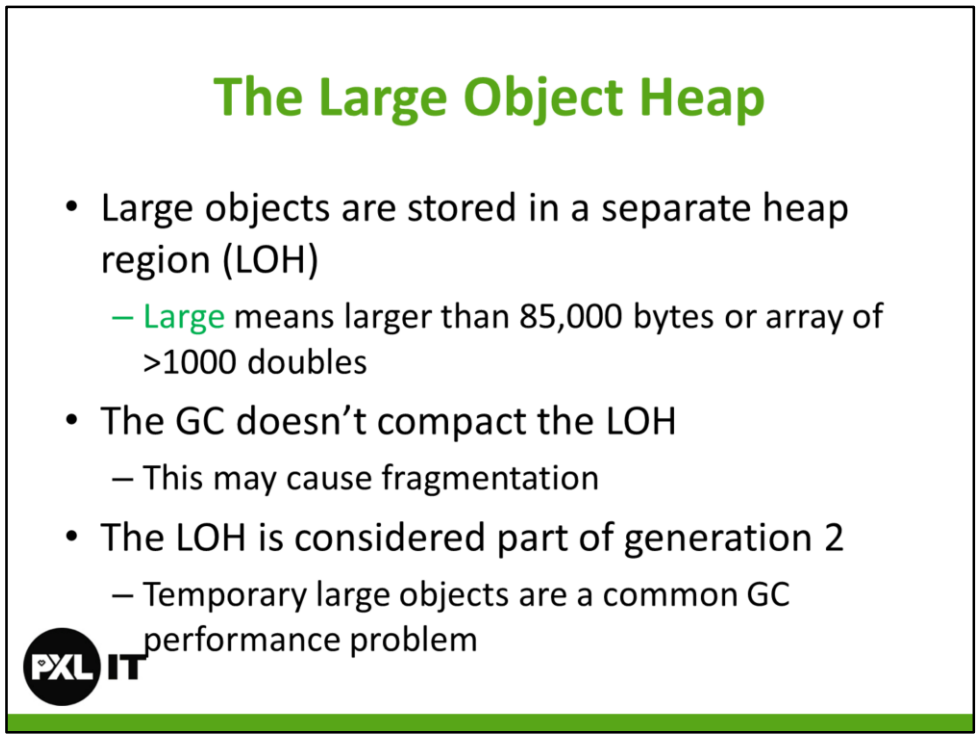
The younger the object, the more likely it will be gc soon.

For example, in a web server, there are many objects created when the server is processing a request. They will all die when the request is finished. On the other hand, old objects that were created during server initialization, like singletons and managers, they're probably going to stay alive until the end of the process. In the .NET garbage collector, there are three areas for objects of various ages.



Generation 0 is where objects are born. Generation one is the nursery and generation two is for tenured old objects that are likely to survive for a long time. The garbage collector will try to perform lots and lots of generation 0 collections and touch generation two only infrequently. Because most objects in generation 0 are young, they die fast, so the generation 0 collections are going to be very efficient and will reclaim lots of unused space. The sizes of the three generations depend on numerous runtime parameters. Generations are larger on 64-bit systems than on 32-bit systems. Generation sizes depend on the CPU cache size and even the amount of physical memory. You can also control the generation size by the application hosting the CLR, such as the ASP.NET host. But a typical generation 0 size on a 32-bit system is going to be a few MB, and that's something that fits in a CPU cache and means a garbage collection can be extremely fast. And most generation 0 garbage collections on modern hardware should complete in less than a msec. So you could have hundreds of them per second. Because generation 0 is so small, it fills up very fast. An application that allocates small objects at a rate of 500 MB/sec is going to cause a generation 0 collection every few msec. But, again, these collections are very fast. So we can live with that expense. When a garbage collection is complete in generation 0, the surviving objects are promoted to generation one, and that's the nursery. These objects are still considered young. The GC still assumes it's useful to run frequent collections in generation one. Most of the objects should have got here by accident, accidentally surviving a generation 0 collection. Finally, objects that survive a generation one collection are promoted to

generation two, and that's when the fun ends. Generation two is the resting place of tenured objects that should not die soon. The GC really tries its best to never touch objects in generation two. Once in a while, there might be a gen two collection, and these can take very long because generation two does not have a size limit. So, ideally, there should be very few generation two collections in your application. In terms of best practices, it would make sense to say that you should strive to work within the limits set by the generational model. You should try to make sure that your temporary objects die young in generation 0 or generation one in the worst case, and you should be very careful to avoid frequent promotions of temporary objects into generation two.



Large objects go into a separate heap region, which is logically a part of generation two. This region is called the **large object heap**, or **LOH**, as opposed to the rest, which is the small object heap, or **SOH**. Currently, the CLR considers objects that are larger than 85,000 bytes to be large. There is also a special exception for arrays of doubles. An array of more than 1,000 doubles is also considered a large object, even though it can be much smaller than 85,000 bytes. This heuristic dates to .NET 1.0, and it hasn't changed since. What used to be large in 2001 isn't that large today, but the threshold remains as it is.

There is an important difference between the GC behavior in the LOH and the SOH. In the LOH, the GC does not compact the heap. The idea is that compaction requires copying objects around, and copying large objects is pretty expensive. The downside is that there can be fragmentation in the large object heap as a result. How does the large object heap actually affect us in real-world programs? The main thing to understand is that *large objects are automatically considered old because they are placed in generation two*. So if you end up creating lots of temporary large objects, you're going to be putting pressure on the garbage collector.

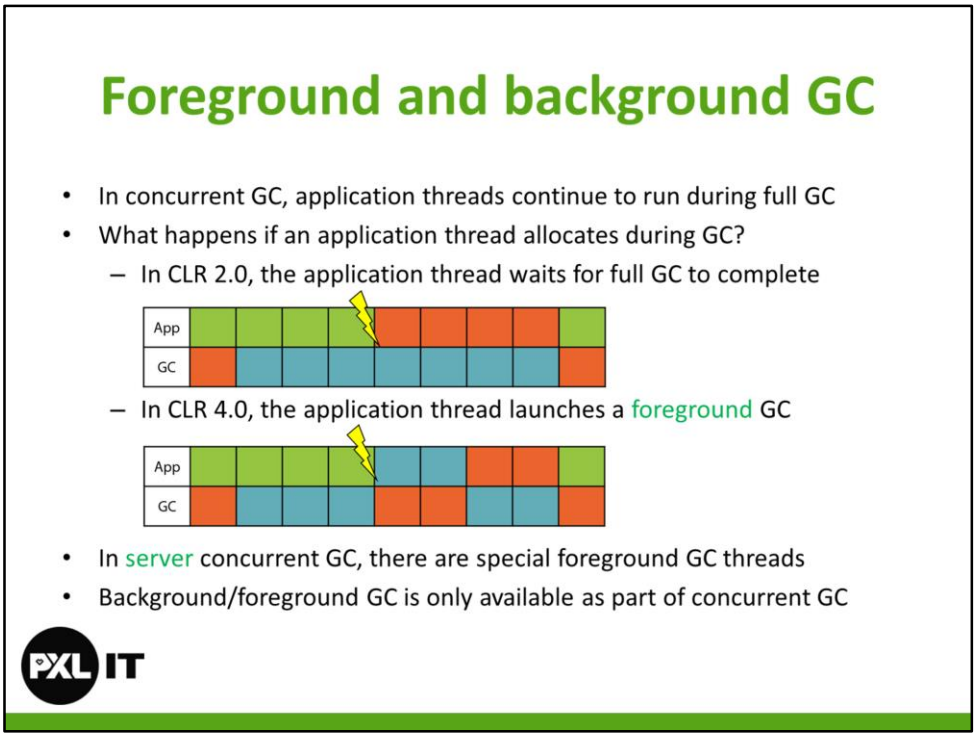
Explicit LOH Compaction

- LOH fragmentation leads to a waste of memory
  - .NET 4.5.1 introduces LOH compaction

```
GCSettings.LargeObjectHeapCompactionMode =
                GCLargeObjectHeapCompactionMode.CompactOnce;
GC.Collect();
```

- You can test for LOH fragmentation using the !dumpheap -stat SOS command

Try to avoid as much as possible.



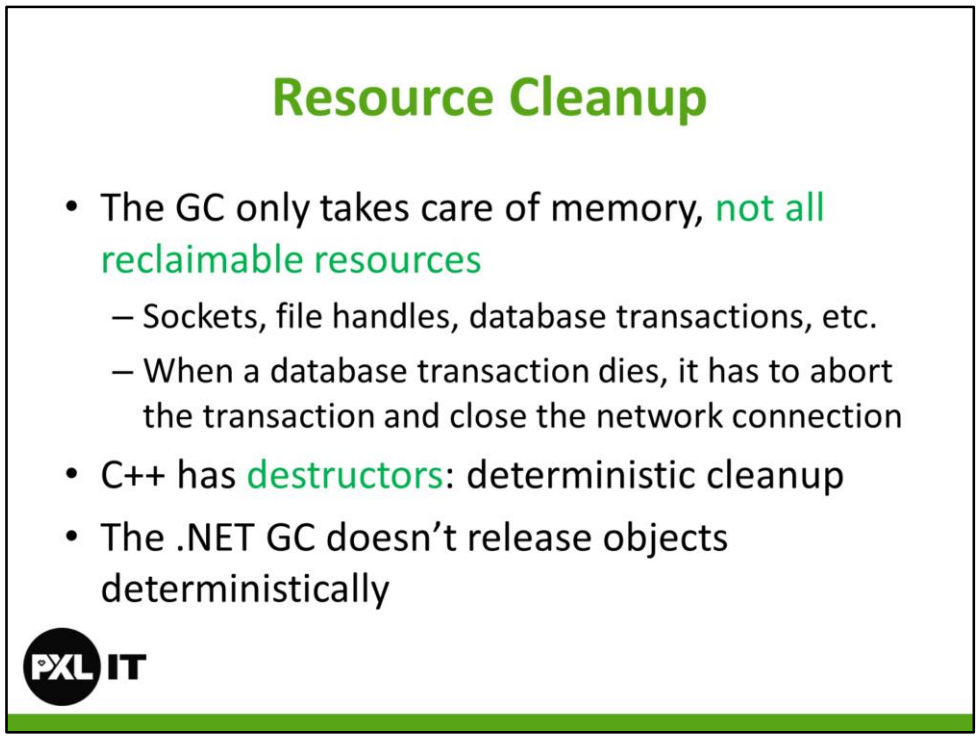
CLR 4.0 introduced another optimization that has to do with garbage collection and generations.

The problem it's trying to solve is the following: suppose you have a concurrent garbage collection working in the background and collecting generation two. Because garbage collection is concurrent, application threads are still allowed to run most of the time, and they can even allocate additional memory. But what happens if an application thread hits the generation 0 cap (= full)? There is already a garbage collection in progress, and it's cleaning up generation two. So, if you have a thread that hits the generation 0 limit, it has to suspend and wait for the full GC to complete, and that can take a while. And that makes concurrent garbage collection look much less attractive.

So CLR 4.0 introduces **background garbage collection for workstation GC** as an extension for concurrent garbage collection. It's in a separate GC flavor. It's just how concurrent GC works after CLR 4.0. When you have background GC, the background GC thread performs generation two collections and doesn't suspend application threads. But if you have an application thread that hits the generation 0 allocation cap, it actually launches a foreground collection on the application thread. So, the foreground GC waits for an acknowledgement from the background GC thread. The background GC thread says, "Hey, I just suspended myself," which usually happens quite quickly, and then the

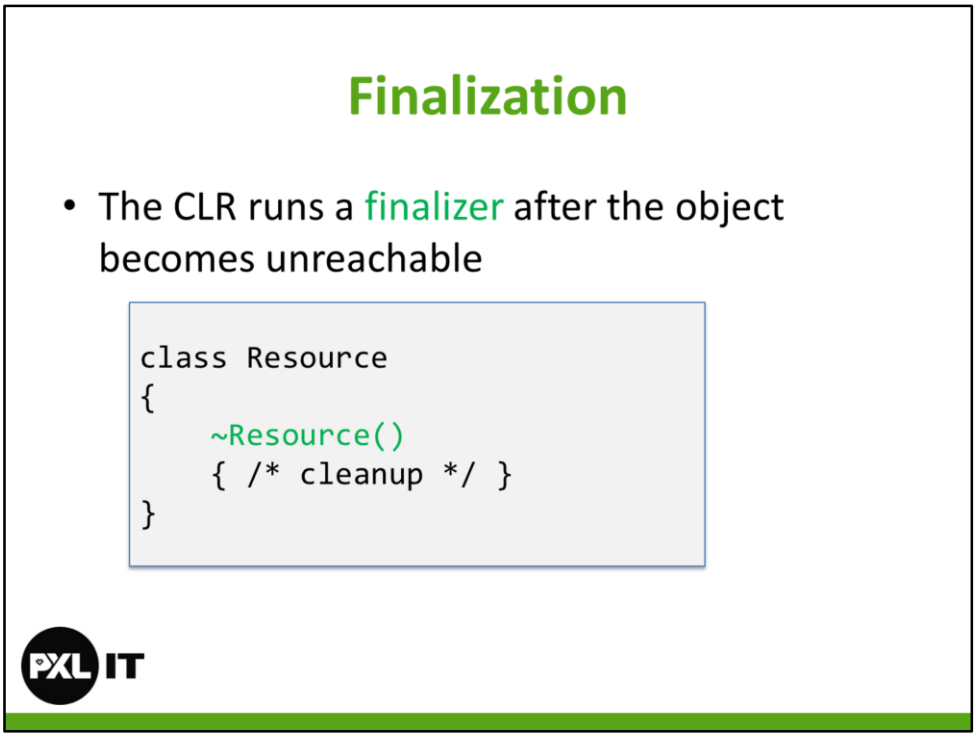
foreground GC cleans up generation 0, possibly also generation one, and then it resumes the background GC thread. And this solves the problem of allowing threads to exceed the generation 0 limit, even many times during a generation two collection that's still happening in the background.

In CLR 4.5, there is **background GC for the server** as well. There are some slight differences. The major one being that on the server, there is a background GC thread per logical processor and a foreground GC thread per logical processor.

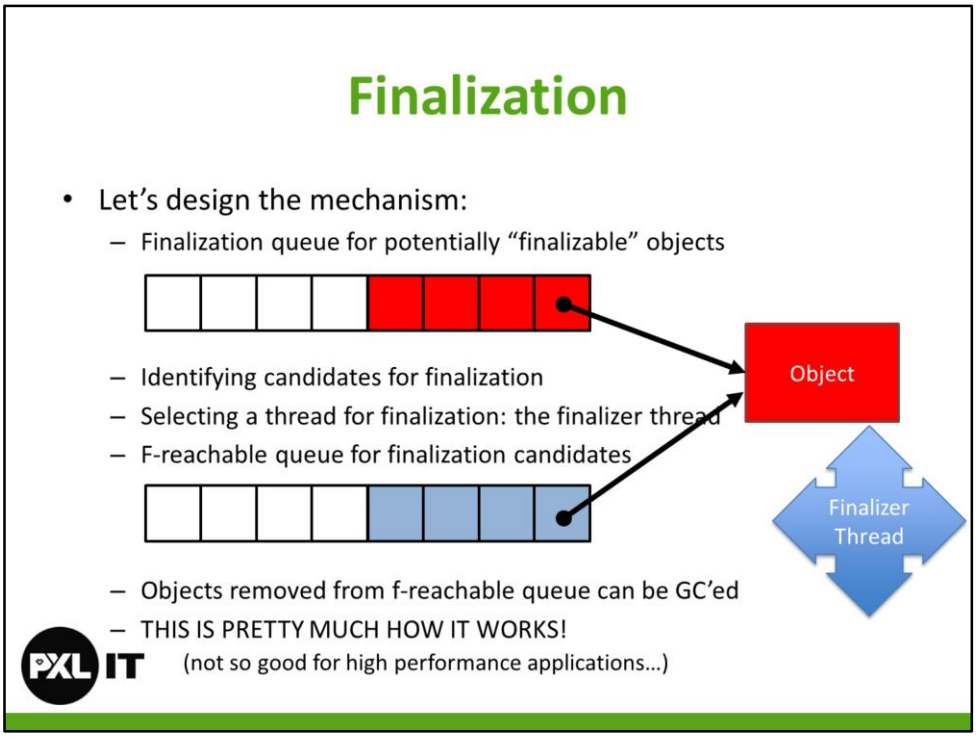


In an ideal world, the garbage collector could take care of all the resources our application needs. In the real world, the GC can only take care of memory. It's very, very good about reclaiming memory, but it's not so good at reclaiming other kinds of resources. Anything else that needs to be closed or disposed or cleaned up explicitly that isn't memory. So, we're talking about file handles, network sockets, database connections, database transactions. These are all examples of resources that have to be cleaned up because they refer to something that isn't just memory. For example, when a database transaction dies, it has to tell the database to either commit or abort the transaction. And it also possibly has to close an open network connection to the database server. Both of these cleanup operations must be explicitly performed. They're not a part of normal memory management, which is what the GC is good at.

Now, in languages without garbage collection, such as C++, you often have deterministic resource cleanup in the form of destructors. In C#, objects do not die at the deterministic point in time. The garbage collector reclaims unused memory at some point in the future after the object has become unreachable. And this actually makes it hard to plan for the releasing of non-memory resources.



The .NET GC has the ability to run cleanup code automatically at some point in the future after an object becomes unreachable. The programmer has to write a special method called the **finalizer**, which the garbage collector invokes. This method is the class's opportunity to release any non-memory resources, such as network sockets and database transactions.



But how exactly does the GC run the finalizer if the object is already unreachable? There is no way to reach unreachable objects, you know, by definition, not to mention the call methods on unreachable objects. So, let's try to design this ourselves and see that what we get is pretty close to how CLR finalization actually works.

First, we need a data structure for keeping track of all the objects that *potentially* require finalization. Let's call this the **finalization queue** and put an object in it when it's created. Of course, we should only put an object in the queue if it has a finalizer. The finalization queue should be a GC root, so we don't accidentally reclaim memory for an object that hasn't been finalized yet.

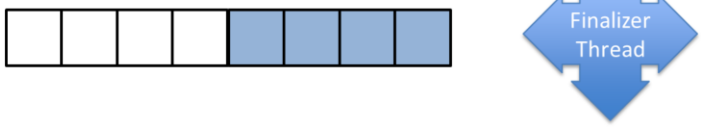
Next, we need a way to know that an object is *eligible for finalization*. And this is easy. We start marking the heap from all the roots except the finalization queue. And then, we'll look at the finalization queue. Any objects that weren't marked yet are now eligible for finalization because they're only referenced by the finalization queue.

Now suppose we have an object that has to be finalized. Just to remind you, we're in the middle of a garbage collection right now, so we can't really afford to stop in the middle of a GC and start calling user defined methods, which can take, you know, an unbounded

amount of time. So, here's an idea. Let's postpone finalization and schedule it on a different thread. Which thread exactly? It can't be an application thread. It can't be the GC thread. **So, let's create a new thread, the finalizer thread**. Now to put work on the finalizer thread, we need another queue of objects that are **ready for finalization**. We'll probably call this the finalizable queue, but the CLR calls it the **F-reachable queue**. Objects on this queue are live only because they are waiting for finalization. Finally, the finalizer thread picks objects from the F-reachable queue, runs their finalizers, and objects with finalizers that have run can be removed from the F-reachable queue and can be collected by the garbage collector. So, the next GC cycle is going to pick them up.

Performance problems with Finalization

- Finalization extends object lifetime
- The f-reachable queue might fill up faster than the finalizer thread can drain it



  - Can be addressed deterministic finalization (Dispose)
- It's possible for a finalizer to run while an instance method hasn't returned yet

PXL IT

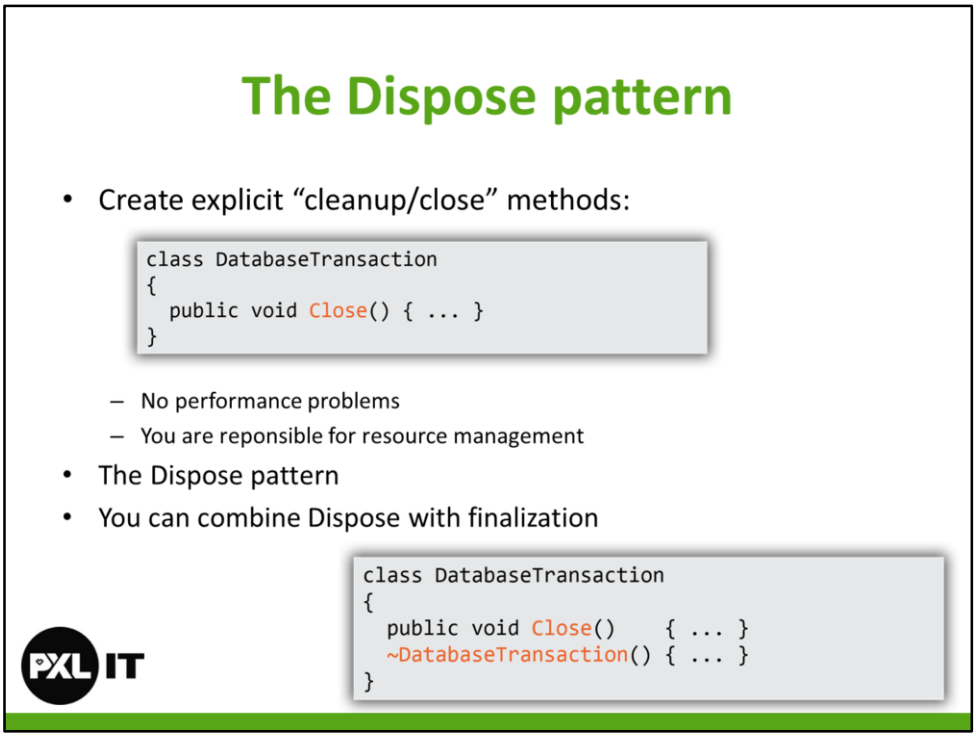
The *first problem* with finalization is that **it extends object lifetime**. An object will survive at least one garbage collection if it has a finalizer. Objects that used to die in generation 0 might die in generation one. Objects that used to slip into generation one will now slip into generation two. Now take many of these objects, and you've got yourself a performance problem, because there will be many generation two garbage collections.

The *second problem* with finalization is that it is **subject to an unbounded queue problem**. The F-reachable queue can fill at a faster rate than the finalizer thread can drain it. For example, suppose it takes 20 msec to finalize a database transaction object. Now if the application creates a hundred of these objects per second, there is an immediate memory leak. One hundred of these objects can be queued to the F-reachable queue every second, but the finalizer thread can only clean up 50 of these objects every second, because each object takes 20 msec to clean. This kind of memory leak can be pretty hard to discover during development. It might only manifest under heavy load or when finalizers run slower in the production environment than in development. It's a very unfortunate situation to be in, and we'll talk about ways to address this problem by moving finalization to the application thread and making it deterministic. And that's the dispose pattern in a nutshell.

The *third problem* I'd like to show you is rather subtle. The underlying problem is that finalization happens on a separate thread, which can introduce **nontrivial race conditions**. The crux of the matter is that it's possible for a finalizer of an object to run while the method of that very object is still executing (see demo).

Demo: race condition finalization

- Watch the demo on Pluralsight
  - Module 1, 6:46 – 13:26
- Key points:
  - Local variables can get cleanup before the end of their scope
  - Finalizer gets run while the object is still executing methods
  - Stay away from finalization and use deterministic cleanup

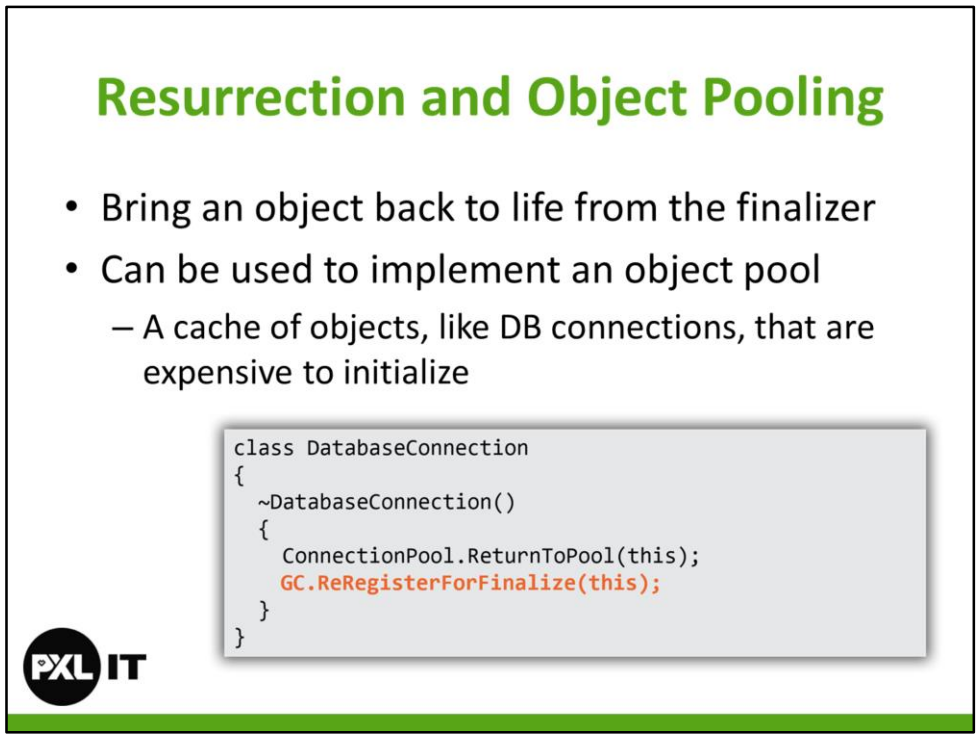


Just create a method that does the cleanup and call it when necessary. The biggest disadvantage? You have to call it!

This pattern is known as the **Dispose** pattern. In the Dispose pattern, you call the cleanup method Dispose, and you make the class implement the **IDisposable** interface. It's merely a convention, though. No one calls the Dispose method automatically for you. It's entirely your responsibility.

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

Sometimes, it might also make sense to combine the Dispose pattern with a finalizer. The finalizer serves as a fallback in case the user forgot to call Dispose. It seems like the best of both worlds again. The finalizer could log a wording message or even fire an assertion to make sure next time, the developer isn't going to forget to call Dispose. For all the nitty-gritty details, consult the MSDN documentation on the Dispose pattern.



A pretty cool and twisted thing that you can do from inside a finalizer is to bring the object back to life. After all, nothing stops you from creating a new root reference to the current object. The GC won't be able to reclaim it in that case. This technique is called **resurrection**, and it's often used to create object pools. The idea of an object pool is that instead of recreating objects every time you need them, your objects sit in a cache, and you can take them from the cache when you need them and return them back to the cache, or to the pool, when you're done with them. This can save the initialization costs for some expensive objects.

Database frameworks usually use this approach for pulling database connections instead of creating a new connection for every operation. Now what does it have to do with finalization? One way to return the object to the pool is by using a finalizer that will do this automatically. The finalizer can put the object back in the pool even if the user forgot to. But if you do this, consider what happens the next time the objects become unreferenced. There is no reference to the object from the finalization queue, so it's going to die right away. And to prevent this, we have the **GC.ReRegisterForFinalize** method, one of the most obscure in the .NET Framework that puts the object back in the finalization queue. You should call ReRegisterForFinalize when you return the object back to the pool, and then the next pool user is not going to lose the object forever.