Improve garbage collection performance (Windows Store apps using C#/VB)

<http://msdn.microsoft.com/en-us/library/windows/apps/hh994643.aspx>

2 out of 5 rated this helpful - [Rate this topic](http://msdn.microsoft.com/en-us/library/windows/apps/hh994643.aspx#feedback)

**Note**  This topic applies to C# and Visual Basic only.

Windows Store apps written in C# and Visual Basic get automatic memory management from the .NET garbage collector. This section summarizes the behavior and performance best practices for the .NET garbage collector in Windows Store apps. For more info on how the .NET garbage collector works and tools for debugging and analyzing garbage collector performance, see [Garbage collection](http://msdn.microsoft.com/en-us/library/windows/apps/0xy59wtx.aspx).

The garbage collector determines when to run by balancing the memory consumption of the managed heap with the amount of work a garbage collection needs to do. One of the ways the garbage collector does this is by dividing the heap into generations and collecting only part of the heap most of the time. There are three generations in the managed heap:

* Generation 0. This generation contains newly allocated objects unless they are 85,000 bytes or larger, in which case they are part of the large object heap. The large object heap is collected with generation 2 collections. Generation 0 collections are the most frequently occurring type of collection and clean up short-lived objects such as local variables.
* Generation 1. This generation contains objects that have survived generation 0 collections. It serves as a buffer between generation 0 and generation 2. Generation 1 collections occur less frequently than generation 0 collections and clean up temporary objects that were active during previous generation 0 collections. A generation 1 collection also collects generation 0.
* Generation 2. This generation contains long-lived objects that have survived generation 0 and generation 1 collections. Generation 2 collections are the least frequent and collect the entire managed heap, including the large object heap which contains objects that are 85,000 bytes or larger.

You can measure the performance of the garbage collector in 2 aspects: the amount of time it takes to do the garbage collection and the memory consumption of the managed heap. As mentioned earlier, the garbage collector strives for the best balance between the two. If you have a small app with a heap size that’s less than 100MB, focus on reducing memory consumption. If you have an app with a managed heap that’s larger than 100M, focus on reducing the garbage collection time only. You can help the .NET garbage collector achieve better performance by following the tips we look at next.

**Reduce memory consumption**

**Remove references**

A reference to an object in your app prevents that object, and all of the objects it references, from being collected. The .NET just-in-time (JIT) compiler does a good job of detecting when a variable is no longer in use so objects held onto by that variable will be eligible for collection. But in some cases it may not be obvious that some objects have a reference to other objects because part of the object graph might be owned by libraries your app uses. To learn about the tools and techniques to find out which objects survive a garbage collection, see [Garbage collection and performance](http://msdn.microsoft.com/en-us/library/windows/apps/ee851764.aspx).

**Induce a garbage collection if it’s useful**

Induce a garbage collection only after you have measured your app's performance and have determined that inducing a collection will improve its performance.

You can induce a garbage collection of a generation by calling **[GC.Collect(n)](http://msdn.microsoft.com/en-us/library/windows/apps/y46kxc5e.aspx)**, where n is the generation you want to collect (0, 1, or 2). We recommend that you don't force a garbage collection in your app because the garbage collector uses many heuristics to determine the best time to perform a collection, and forcing a collection is in many cases an unnecessary use of the CPU. But if you know that you have a large number of objects in your app that are suddenly no longer used and you want to return this memory to the system, then it may be appropriate to force a garbage collection. For example, you can induce a collection at the end of a loading sequence in a game to free up memory before gameplay starts.

To avoid accidently inducing too many garbage collections, you can set the **[GCCollectionMode](http://msdn.microsoft.com/en-us/library/windows/apps/bb495757.aspx)** to **Optimized**. This instructs the garbage collector to start a collection only if it determines that the collection would be productive enough to be justified.

**Reduce garbage collection time**

Follow the best practices here only after you've analyzed your app and observed large garbage collection times that you want to reduce. There are 2 aspects of garbage collection-related pause times: the time it takes to run a single garbage collection pass, and the total time your app spends doing garbage collections. The amount of time it takes to do a collection depends on how much live data the collector has to analyze. Generation 0 and generation 1 are bounded in size, but generation 2 continues to grow as more long-lived objects are active in your app. This means that the collection times for generation 0 and generation 1 are bounded, while generation 2 collections have the potential of taking a longer amount of time. How often garbage collections run depends mostly on how much memory you allocate, because a garbage collection frees up memory to satisfy allocation requests.

The garbage collector occasionally pauses your app to perform work, but doesn't necessarily pause your app the entire time it is doing a collection. Pause times are usually not user-perceivable in your app, especially for generation 0 and generation 1 collections. The [Background garbage collection](http://msdn.microsoft.com/en-us/library/windows/apps/ee787088.aspx#background_garbage_collection)feature of the .NET garbage collector allows Generation 2 collections to be performed concurrently while your app is running and will only pause your app for short periods of time. But it is not always possible to do a Generation 2 collection as a background collection. In that case, the pause can be user-perceivable if you have a large enough heap (more than 100MB).

Frequent garbage collections can contribute to increased CPU (and therefore power) consumption, longer loading times, or decreased frame rates in your application. Below are some techniques you can use to reduce garbage collection time and collection-related pauses in your managed Windows Store app.

**Reduce memory allocations**

If you don’t allocate any objects then the garbage collector doesn’t run unless there is a low memory condition in the system. Reducing the amount of memory you allocate directly translates to less frequent garbage collections.

If in some sections of your app pauses are completely undesirable, then you can pre-allocate the necessary objects beforehand during a less performance-critical time. For example, a game might allocate all of the objects needed for gameplay during the loading screen of a level and not make any allocations during gameplay. This avoids pauses while the user is playing the game and can result in a higher and more consistent frame rate.

**Reduce generation 2 collections by avoiding objects with a medium-length lifetime**

Generational garbage collections perform best when you have really short-lived and/or really long-lived objects in your app. Short lived objects are collected in the cheaper generation 0 and generation 1 collections, and objects that are long-lived get promoted to generation 2, which is collected infrequently. Long-lived objects are those that are in use for the entire duration of your app, or during a significant period of your app, such as during a specific page or game level.

If you frequently create objects that have a temporary lifetime but live long enough to be promoted to generation 2, then more of the expensive generation 2 collections happen. You may be able to reduce generation 2 collections by recycling existing objects or throwing away objects more quickly.

A common example of objects with medium-term lifetime is objects that are used for displaying items in a list that a user scrolls through. If objects are created when items in the list are scrolled into view, and are no longer referenced as items in the list are scrolled out of view, then your app typically has a large number of generation 2 collections. In situations like this you can pre-allocate and reuse a set of objects for the data that is actively shown to the user, and use short-lived objects to load info as items in the list come into view.

**Reduce generation 2 collections by avoiding large-sized objects with short lifetimes**

As mentioned previously, any object that is 85,000 bytes or larger is allocated on the large object heap (LOH) and gets collected as part of generation 2. If you have temporary variables, such as buffers, that are greater than 85,000 bytes, then a generation 2 collection cleans them up. Limiting temporary variables to less than 85,000 bytes reduces the number of generation 2 collections in your app. One common technique is to create a buffer pool and reuse objects from the pool to avoid large temporary allocations.

**Avoid reference-rich objects**

The garbage collector determines which objects are live by following references between objects, starting from roots in your app. For more info, see[What happens during a garbage collection](http://msdn.microsoft.com/en-us/library/windows/apps/ee787088.aspx#what_happens_during_a_garbage_collection). If an object contains many references, then there is more work for the garbage collector to do. A common technique (especially with large objects) is to convert reference rich objects into objects with no references (e.g., instead of storing a reference, store an index). Of course this technique works only when it is logically possible to do so.

Replacing object references with indexes can be a disruptive and complicated change to your app and is most effective for large objects with a large number of references. Do this only if you are noticing large garbage collection times in your app related to reference-heavy objects.