Garbage Collection

<https://www.linkedin.com/learning/dot-net-fundamentals-concepts-apis-and-libraries-in-the-dot-net-framework/auto-memory-management-with-garbage-collection?resume=false&u=42314660>

When the application process starts, the CLR allocates and reserves a contiguous memory.

This is the managed heap. All reference types are allocated on the managed heap. Each operating thread is allocated some memory, called the thread stack. The stack is where your local variables live. The managed heap is where the object instances live.

A screen shot of a computer

Description automatically generated

When you instantiate an object, two actions happen. The variables are added to the stack, and the objects are instantiated and placed on the managed heap.

In this example, the variables card1, player1, and card2 are on the thread stack, and each one is pointing to its object on the managed heap. You can think of the variable as a reference pointer to the existing object.

A screenshot of a computer

Description automatically generated

The managed heap maintains a pointer to the address where the next object in the heap will be allocated. Initially, this pointer is set to the managed heap's base address.

A screen shot of a computer

Description automatically generated

When an application creates the first reference type, memory is allocated for the type at the base address of the managed heap. The next object pointer is moved to the next available spot.

A screen shot of a computer

Description automatically generated

When the application creates the next object, the CLR allocates memory in the address space immediately following the first object. As long as address space is available, the CLR continues to allocate space for new objects in this manner.

A screen shot of a computer

Description automatically generated

Allocation on the heap is fast. This is because the CLR knows precisely where to put the new object. It doesn't have to look around the heap for a suitable size space. When the heap is full, the garbage collector removes unused objects and compacts the memory so all the items in the heap are moved to the top. While this happens, all application threads are halted, and the reserved garbage collection thread is activated to do the work. This means that your application is not working until the garbage collection is finished.

Here's what the heap looks like when full, and there is no room to allocate a new object. The garbage collector examines all the object references and determines which ones are no longer used in the application.

In this example, that is the Player and Deck objects. In garbage-collecting terms, these are called unreachable objects. Because there is no reachable path to the object, and there are no references to them.

The items currently in use are called reachable objects. The unreachable objects can be removed, or their memory can be reclaimed for use by newer objects. As the garbage collector discovers each unreachable object, it uses a memory copying function to compact the reachable objects into the newly freed memory location. Essentially, it is squeezing all the reachable ones into one contiguous space. Once the memory for the reachable objects has been compacted, the garbage collector makes the necessary reference pointer corrections so that the application's variables point to the objects in their new locations.

A screenshot of a computer screen

Description automatically generated

When the garbage collection is complete, the next object can finish being instantiated.

So, what are the implications of this process? Allocations are fast. However, triggering a collection takes some time to complete.

But the garbage collector is more sophisticated than we've discussed.

To optimize the garbage collector's performance, the managed heap is divided into three sections called generations: Gen 0, Gen 1, and Gen 2. Plus, there is a specialised heap for large objects, which I have yet to show on the screen.

To be clear, I've drawn them as separate heaps. However, the heap is one contiguous memory area. The generations are logical constructs managed by the CLR. Gen 0 is small, Gen 1 is bigger, and Gen 2 is the biggest of the three.

A screenshot of a video

Description automatically generated

Why have three? The runtime's garbage collection algorithm is based on well-known discoveries in computer science.

First, it is faster to compact the memory for a portion of the managed heap than for the entire managed heap.

Second, the lifetime of an object, in other words, how long it remains reachable, is strongly correlated to its age.

New objects tend to be short-lived. Think about an object instantiated in a button click handler. The user clicks the button. The object is created and used. And when the handler is done, the object is no longer needed. It was only in use for a few seconds.

So it is fruitful to isolate the newest objects together. Running the garbage collector on them would free up a lot of memory.

The opposite is true for older objects. The more collections they survive, the more likely they are to survive the next one. Examining old objects during a garbage collection is a waste of time.

That's why there are three generations.

Generation 0 is small in size, intentionally small enough to fit into the CPU's L2 cache. That means the Gen 0 portion of the heap fills up quickly, which means it triggers the garbage collection frequently.

Let's see what happens. Gen 0 is full. The garbage collection starts. The Player and Deck objects are no longer reachable.

A screenshot of a computer screen

Description automatically generated

All the reachable objects are moved from Gen 0 to Gen 1. Then the memory in Gen 0 is reset. The next object pointer is moved to the base address. When the application resumes, all the new objects are placed in Gen 0.

A screenshot of a game

Description automatically generated

During our garbage collection, Gen 1 is not checked until it gets full. Gen 0 collections are the most common. Because Gen 0 is small enough to fit into the CPU cache, the memory compression is super fast, usually less than one millisecond. So even if your app is triggering lots of collections, it's not noticeable as long as it's Gen 0 only.

As time goes on, more surviving objects get moved to Gen 1. Remember, it's bigger than Gen 0, so it might fill up slower than a smaller sibling. Eventually, it will get close to its max capacity.

Now, the next time that garbage collection runs, it knows that Gen 1 is nearly full. It includes both generations in the cleaning phase.

A screenshot of a computer

Description automatically generated

In this example, the Dice object in Gen 0 is reachable and moved to Gen 1. The Card and Score objects are reachable, and they are moved to Gen 2.

A screenshot of a computer

Description automatically generated

Gen 2 objects are likely the longest-living objects in the application. Perhaps they are objects used continuously and are needed as long as the application runs. Since it's always in use, there's no need to check it constantly. The garbage collector will compact Gen 2 when it deems it necessary. And when that happens, it will impact your application performance the most. You can be assured it won't happen unless it's necessary, though.

In the rare situation where all heaps are full, and there's no room to allocate the new object, the CLR throws a memory exception.