Garbage collector

The Garbage Collector (GC) is a component of the .NET ecosystem that automatically manages memory allocation and deallocation in a .NET application. Its primary responsibility is to track and reclaim memory that is no longer in use by the application, allowing developers to focus on writing code without explicitly managing memory.

Memory Allocation: When objects are created in a .NET application, memory is allocated on the managed heap. The GC keeps track of the allocated memory and manages its lifetime.

Reachability: The GC determines which objects are still in use by the application. It considers an object as "reachable" if there is a reference to it from the application's root objects or from other reachable objects.

Garbage Collection Process: Periodically, the GC performs a garbage collection process to reclaim memory occupied by objects that are no longer reachable.

Generational GC: The .NET GC uses a generational garbage collection approach. It divides objects into different generations based on their age. Newly allocated objects are placed in the youngest generation (Generation 0), and objects that survive garbage collections are promoted to older generations (Generation 1 and Generation 2

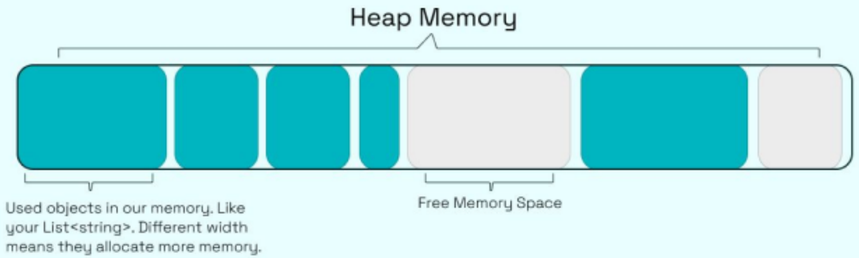
Automatic Memory Management: The GC automatically detects when the application needs more memory and triggers a garbage collection. Developers don't need to explicitly free memory or deallocate objects.

Finalization: The GC also provides a mechanism called finalization, which allows objects to perform cleanup tasks before they are reclaimed.

How the GC works

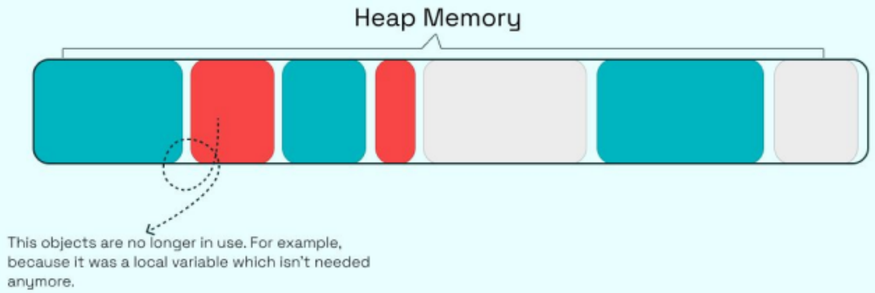
The GC in .NET works in several phases to reclaim memory and manage the heap efficiently. These phases include the marking phase, sweeping phase, and, in some cases, the compacting phase

The Garbage Collector is your personal housemaid that cleans after you(r memory). It takes care of the managed memory, kind of like a watch dog that runs through our objects and checks whether or not they can be deleted.



1. Marking Phase

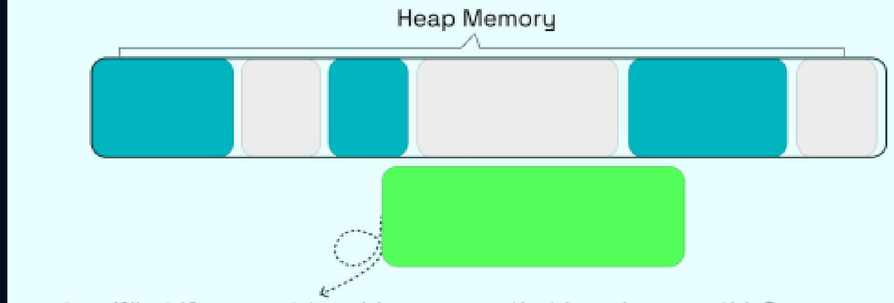
In the Marking Phase the Garbage Collector marks potential objects for deletion. They can be cleaned up in a later phase.



Marking phase

2. Sweeping Phase

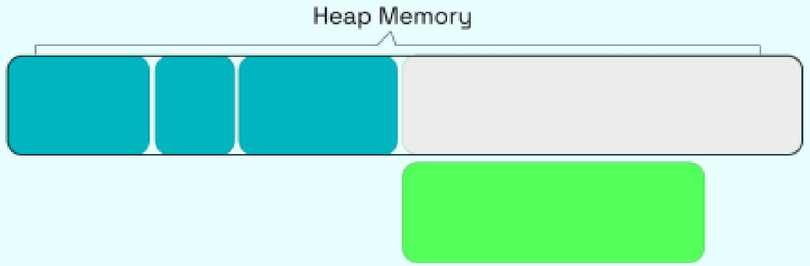
In the Sweeping Phase we really delete the marked object from the last phase so we have free space for later usage.



What if we want to add a memory that is as large as this?
It seems it doesn't fit in the free memory holes!

3. Compacting Phase

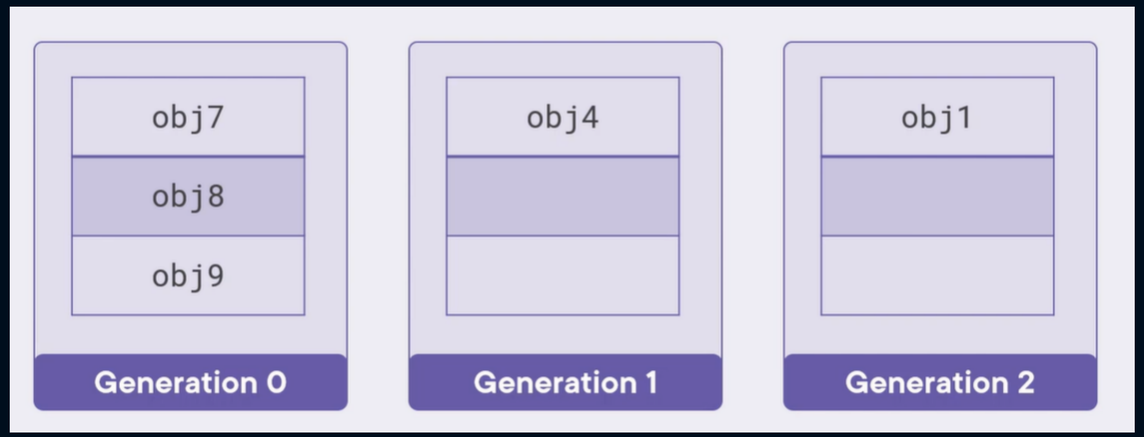
Like the old defragmentation tool on Windows we want to move the free space and used space together! Then also our new big memory block would fit in!



Sweeping and Compacting phase

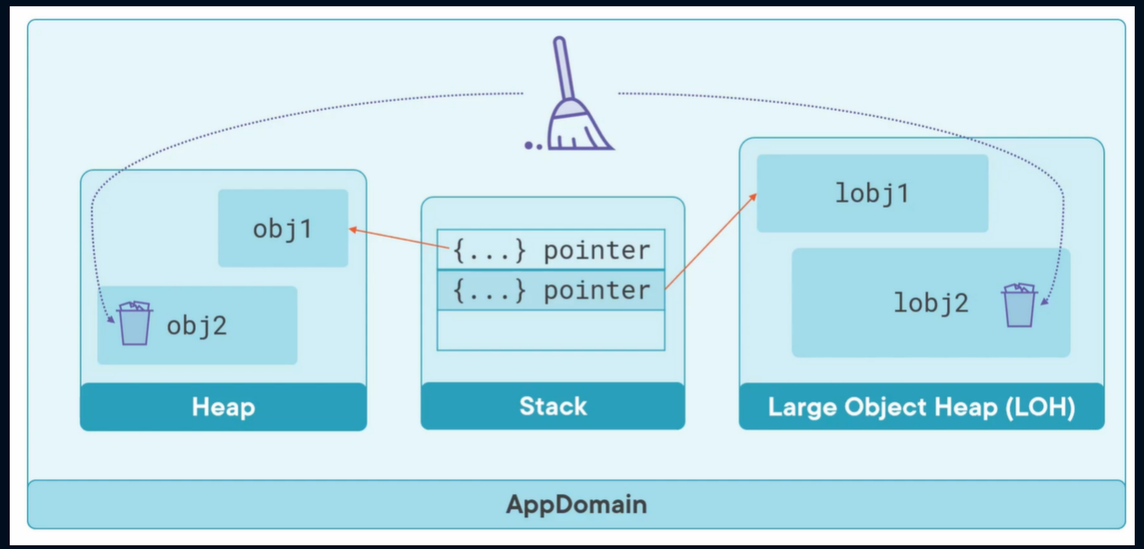
GC Generations

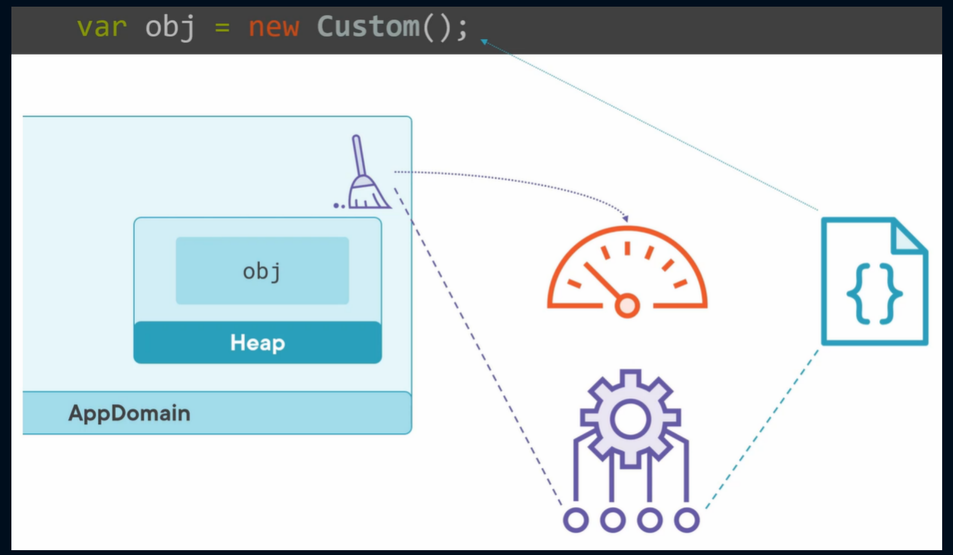
The generational GC takes advantage of the observation that most objects in a typical application have a short lifespan and become garbage quickly.
Generation 0 is the youngest generation and contains newly allocated objects. Gen 0 collections are generally fast because they only need to scan a small portion of the heap.
Objects that survive a Gen 0 collection are promoted to Generation 1. The GC performs less frequent Gen 1 collections. These collections involve scanning a larger portion of the heap but are still relatively fast.
Generation 2 contains long-lived objects that have been around for a while. Since Gen 2 collections involve scanning the entire heap, they are relatively slower compared to Gen 0 and Gen 1 collections.



Collection processes

Heap: The Heap is the memory region used for dynamic memory allocation. The Heap is divided into generations
The Stack is a region of memory used for storing method calls, local variables, and function call information. It does not directly participate in the GC collection process.
The Large Object Heap (LOH) is a separate memory area within the Heap specifically designed for storing large objects





Regarding the execution of the collection process, it typically occurs on a separate background thread, not the main thread of your application.

Managed and unmanaged code

The Garbage Collector (GC) in the managed runtime environment can only manage and collect memory used by managed objects in managed code.
Unmanaged objects may include resources like native memory, file handles, database connections, or COM objects. Since the GC cannot directly manage the memory used by unmanaged objects, it does not automatically free the associated resources when collecting managed objects, for that it is needed to implement disposal mechanism like finalizers, IDisposable pattern or APIs provided by the unmanaged code library.

The **extern** keyword in C# is used to indicate that a method is implemented externally; typically in unmanaged code or in another programming language.

The **DllImport** attribute is used to import functions from external libraries (DLLs) into your C# code.

The **IntPtr** type in C# is a platform-specific type that is used to represent a pointer or a handle to an unmanaged resource.

```
// Declare the unmanaged function using the 'DllImport' attribute
[DllImport("user32.dll")]
public static extern int MessageBox(IntPtr hWnd, string text, string caption, int options);

public static void Main(string[] args)
{
    // Call the unmanaged function
    MessageBox(IntPtr.Zero, "Hello, World!", "Message", 0);
}
```

Use the GC manually

It is generally recommended to let the .NET garbage collector (GC) handle memory management automatically, as it is specifically designed to optimize memory usage and deallocation.

Here are some cases where manual GC invocation may be considered:

Performance Optimization: In certain scenarios, you might have insights into the application's memory usage patterns and can determine that a manual GC invocation at an appropriate time can improve performance

Memory Pressure Management: If your application is performing memory-intensive operations.

It's important to note that manually triggering the GC excessively or at inappropriate times can have adverse effects on performance

Consider the following: Memory should be available and contiguous and collection should be efficient and infrequent

```
// Create a large number of managed objects
for (int i = 0; i < 10000; i++)
{
    ManagedResource resource = new ManagedResource($"Resource {i}");
    // Perform some operations with the managed resource // ...
}

// Manually run the garbage collector for generation 2 objects
GC.Collect();
GC.WaitForPendingFinalizers();

// Check memory usage after garbage collection
long memoryUsed = GC.GetTotalMemory(false);
Console.WriteLine($"Memory used after garbage collection: {memoryUsed} bytes");
```

Unmanaged memory and IDisposable

Unmanaged memory refers to memory that is not managed by the .NET runtime and is typically allocated and deallocated manually. By implementing IDisposable and properly disposing of unmanaged resources, you can ensure that resources are released in a timely manner, the Dispose() method is typically used to free unmanaged memory, close file handles, release network connections, or perform any other necessary cleanup operations.

Unmanaged Memory

Unmamaged memory may come from the following sources:

Streams: representation of a sequence of bytes that can be read from or written to

Bitmaps: data structure that represents an image as a grid of pixels, where each pixel is typically represented by multiple bits.

UI

Memory leak

Allocating memory for longer than needed

Will eventually be collected but may cause issues before it is

Objects that are unable to be collected

The garbage collector is unable to collect resources that still have references, such as event handler leaks.

```
// Using statement - resource disposed after the statement
using(var stream = new MemoryStream(data))
{
    ... // Use the stream
}

// Using declaration - resource disposed when method completes
using var stream = new MemoryStream(data);
```

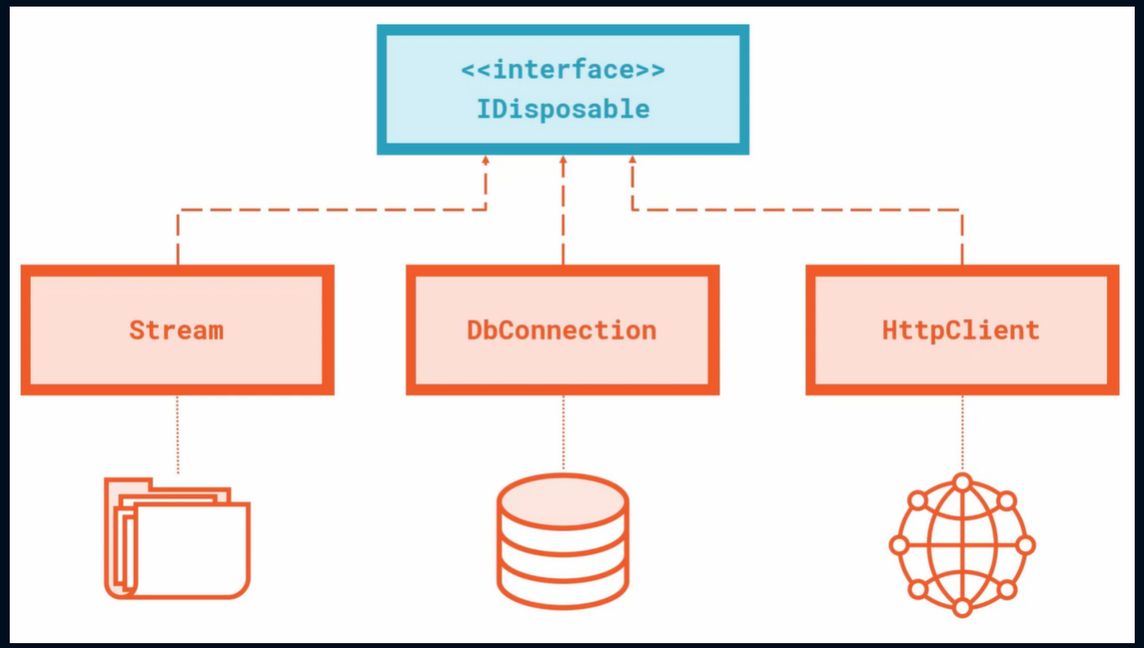
Link: Example

Dispose streams using IDisposable

This kind of resources should follow the disposable pattern

IDisposable

The IDisposable interface in C# is used to provide a mechanism for releasing unmanaged resources and performing cleanup operations.



Using IDisposable

Custom.cs

```
public class Custom : IDisposable
{
    public void Method() {}

    public void Dispose() {}
}
```

Program.cs

```
static void Main()
{
    using (var obj = new Custom())
    {
        obj.Method();
    }
}
```

Best practices

Typical.cs

```
using (var obj = new Custom())
{
    // work with obj
}
// obj.Dispose() is called here
```

Alternative.cs

```
var obj = new Custom();
try
{
    // work with obj
}
finally
{
    obj.Dispose();
}
```

Dispose of IDisposable objects as soon as you can

DatabaseState.cs

```
public class DatabaseState : IDisposable
{
  public void Dispose()
  {
    Dispose(true);
    GC.SuppressFinalize(this);
  }
```

Program.cs

```
using (var s = new DatabaseState())
{
  Console.WriteLine(s.GetDate());
}
```

If you use IDisposible objects as instance fields, implement IDisposable

Best practices

```
protected SqlConnection _connection;

protected void Dispose(bool disposing)
{
  if (_disposed)
    return;

  if (disposing)
  {
    if (_connection != null)
    {
      _connection.Dispose();
      _connection = null;
    }
    _disposed = true;
  }
}
```

◂ Local field is IDisposable

◂ Check disposable object is live

◂ Dispose and set to null

◂ Allow multiple Dispose() calls

Allow Dispose to be called multiple times and don't throw exceptions

BaseClass.cs

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

protected virtual void Dispose(bool disposing)
{
  // dispose only *this* class's resources
}
```

DerivedClass.cs

```
protected override void Dispose(bool disposing)
{
  // dispose only *this* class's resources
}
```

Implement IDisposible to support disposing resources in a class hierarchy

Best practices

Enable static analysis with rule CA200 (Dispose objects before losing scope)

```
protected override void Dispose(bool disposing)
{
    if (disposing)  {  // clean up managed resources  }

    // clean up unmanaged resources

    base.Dispose(disposing);
}

~UnmanagedDatabaseState()
{
    Dispose(false);
}
```

Cleaning Up Resources

Unmanaged: always. Managed: only if disposing.

If you use unmanaged, resources declare a finalizer which cleans them up

Typical.cs

```
using (var obj = new Custom())
{
    // work with obj
}
// obj.Dispose() is called here
```

Alternative.cs

```
var obj = new Custom();
try
{
    // work with obj
}
finally
{
    obj.Dispose();
}
```

Dispose of IDisposable objects as soon as you can

Best practices

IAsyncDisposable

```
public class FileProcessor : IDisposable, IAsyncDisposable
{
    public void Dispose() { }

    public ValueTask DisposeAsync()
    {
        Dispose();

        return ValueTask.CompletedTask;
    }
}

await using var fileProcessor = new FileProcessor();
```

```
public class WithAsyncCleanup : IAsyncDisposable
{
    public async ValueTask DisposeAsync()
    {
        await DisposeAsyncCore();
        GC.SuppressFinalize(this);
    }

    protected virtual async ValueTask DisposeAsyncCore()
    {
        // clean up managed resources
    }
}
```

Implement IAsyncDisposable if your class uses an async disposable field

Demo

We will check the following:

Managed and unmanaged code

Manual GC collections

Dispose pattern

You can check the following Repository(ies) for some examples:

C# fundamentals