Garbage Collector and IDisposable in .NET

1.
Garbage Collector

Garbage collection (GC) is a form of automatic memory management.

- Frees developers from having to manually release memory.
- Allocates objects on the managed heap efficiently.
- Reclaims objects that are no longer being used, clears their memory, and keeps the memory available for future allocations. Managed objects automatically get clean content to start with, so their constructors don't have to initialize every data field.
- Provides memory safety by making sure that an object cannot use for itself the memory allocated for another object.

Fundamentals of memory

- Each process has its own, separate virtual address space. All processes on the same computer share the same physical memory and the page file, if there is one.
- By default, on 32-bit computers, each process has a 2-GB user-mode virtual address space.
- As an application developer, you work only with virtual address space and never manipulate physical memory directly. The garbage collector allocates and frees virtual memory for you on the managed heap.
- If you're writing native code, you use Windows functions to work with the virtual address space. These functions allocate and free virtual memory for you on native heaps.

Fundamentals of memory

- Virtual memory can be in three states:

State	Description
Free	The block of memory has no references to it and is available for allocation.
Reserved	The block of memory is available for your use and cannot be used for any other allocation request. However, you cannot store data to this memory block until it is committed.
Committed	The block of memory is assigned to physical storage.

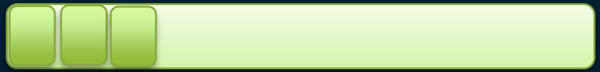
Fundamentals of memory

- Virtual address space can get fragmented. This means that there are free blocks, also known as holes, in the address space. When a virtual memory allocation is requested, the virtual memory manager has to find a single free block that is large enough to satisfy that allocation request. Even if you have 2 GB of free space, an allocation that requires 2 GB will be unsuccessful unless all of that free space is in a single address block.

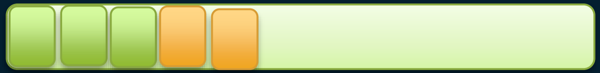
- You can run out of memory if there isn't enough virtual address space to reserve or physical space to commit.

Allocating Resources from the Managed Heap (new operator)

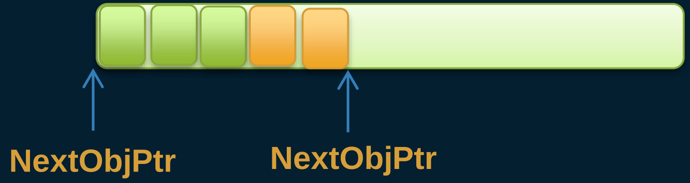
1. Calculate the number of bytes required for the type's fields



2. Add the bytes required for an object's overhead.

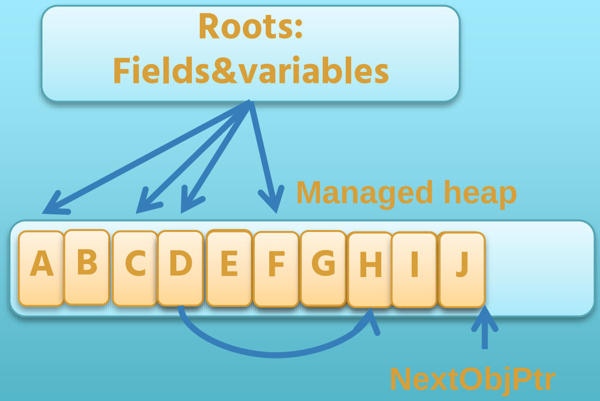


3. The CLR then checks that the bytes required to allocate the object are available in the region.



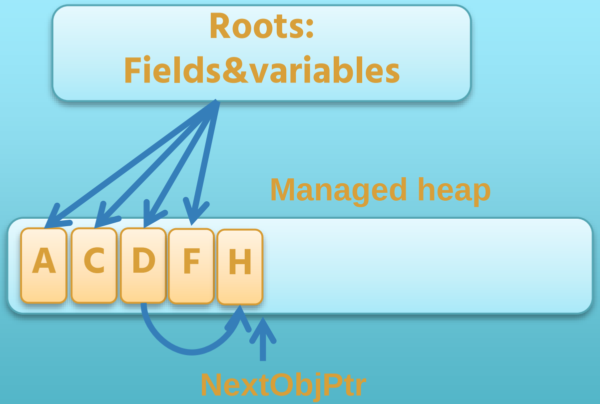
The Garbage Collection Algorithm

1. Marking



Managed heap before a collection

2. Compacting



Managed heap after a collection

Conditions for garbage collection:

- The system has low physical memory. This is detected by either the low memory notification from the OS or low memory as indicated by the host.

- The memory that's used by allocated objects on the managed heap surpasses an acceptable threshold. This threshold is continuously adjusted as the process runs.

- The GC.Collect method is called. In almost all cases, you don't have to call this method, because the garbage collector runs continuously. This method is primarily used for unique situations and testing.

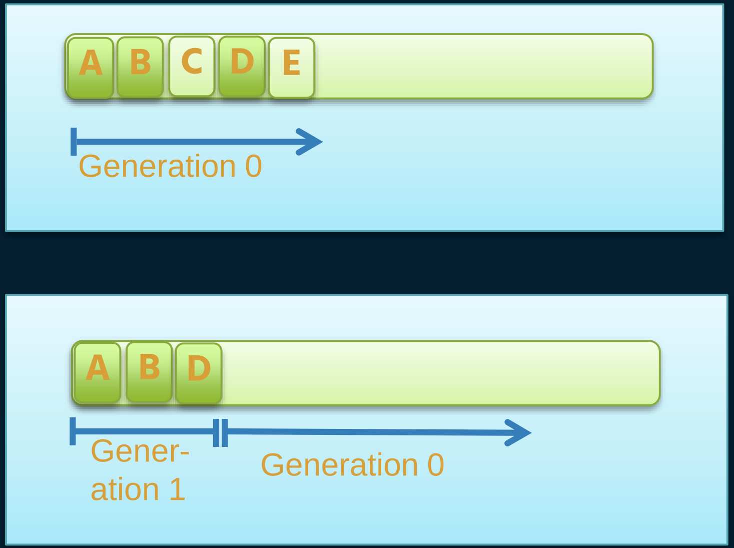
Generational GC assumptions:

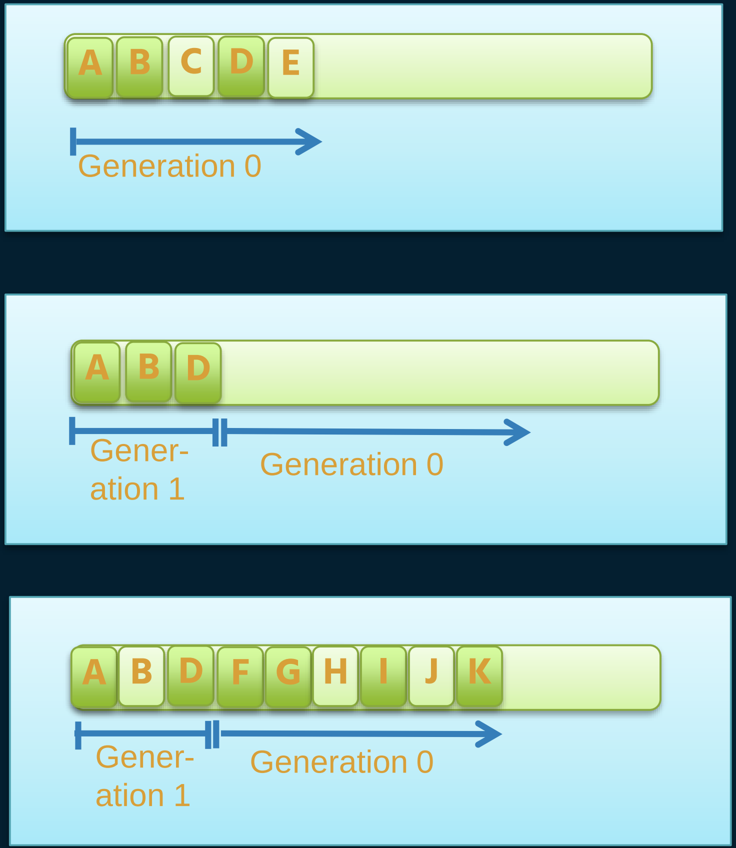
- The newer an object is, the shorter its lifetime will be.
- The older an object is, the longer its lifetime will be.
- Collecting a portion of the heap is faster than collecting the whole heap.

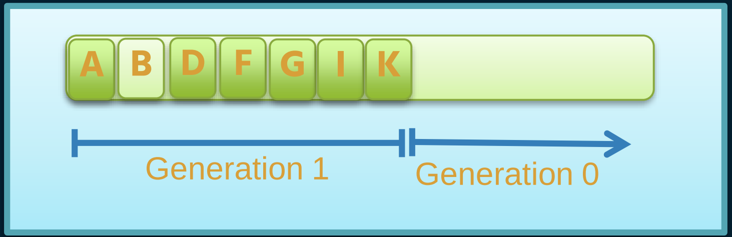
Generations:

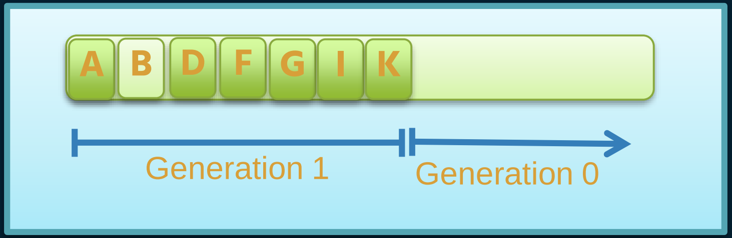
- **Generation 0**. This is the youngest generation and contains short-lived objects. An example of a short-lived object is a temporary variable.
- **Generation 1**. This generation contains short-lived objects and serves as a buffer between short-lived objects and long-lived objects.
- **Generation 2**. This generation contains long-lived objects. An example of a long-lived object is an object in a server application that contains static data that is live for the duration of the process.

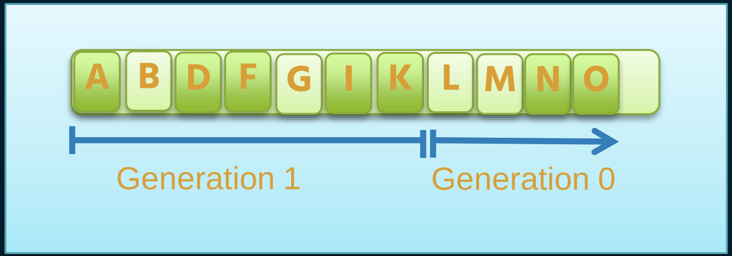

A
B
C
D
E
Generation 0

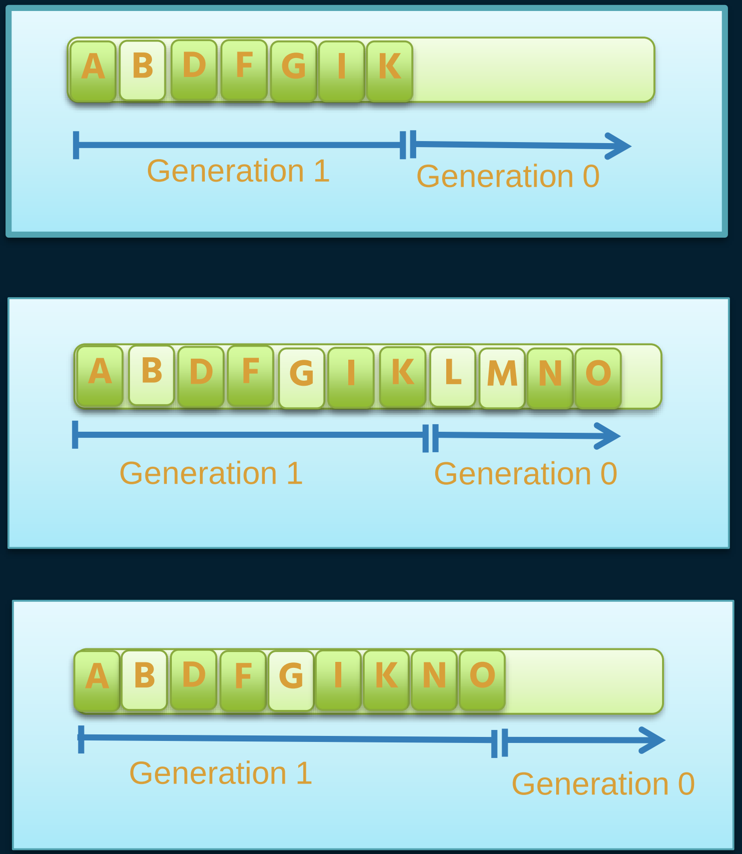

A B C D E
Generation 0
A B D
Gener-
ation 1
Generation 0


A
B
C
D
E
Generation 0
A
B
D
Gener-
ation 1
Generation 0
A
B
D
F
G
H
I
J
K
Gener-
ation 1
Generation 0

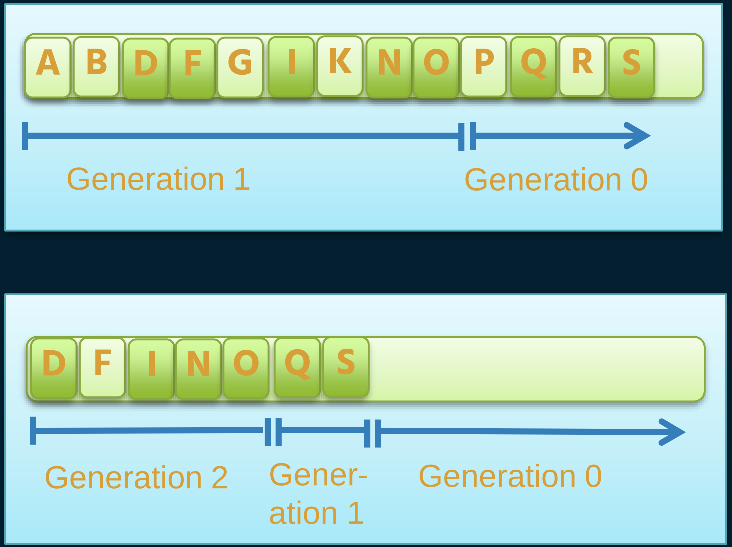

A B D F G I K
Generation 1
Generation 0


A B D F G I K
Generation 1
Generation 0


A B D F G I K L M N O
Generation 1
Generation 0


A B D F G I K
Generation 1
Generation 0
A B D F G I K L M N O
Generation 1
Generation 0
A B D F G I K N O
Generation 1
Generation 0


A B D F G I K N O P Q R S
Generation 1
Generation 0


A B D F G I K N O P Q R S
Generation 1
Generation 0
D F I N O Q S
Generation 2
Gener-
ation 1
Generation 0

Large Objects (> 85000b)

- Large Objects allocated elsewhere within the process' address space.
- GC doesn't compact large objects because of the time it would require to move them in memory.
- Large objects are immediately considered to be part of generation 2.

Running GC

- The system has low physical memory. This is detected by either the low memory notification from the OS or low memory as indicated by the host.
- The memory that's used by allocated objects on the managed heap surpasses an acceptable threshold. This threshold is continuously adjusted as the process runs.
- The GC.Collect method is called. In almost all cases, you don't have to call this method, because the garbage collector runs continuously. This method is primarily used for unique situations and testing.

Forcing Garbage Collections

void Collect(Int32 generation, GCCollectionMode mode, Boolean blocking)

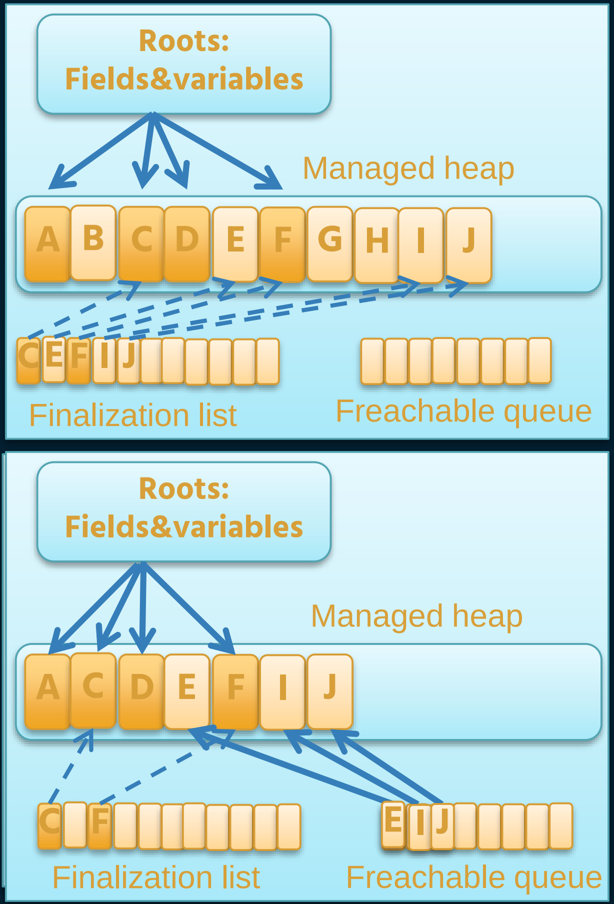
Default

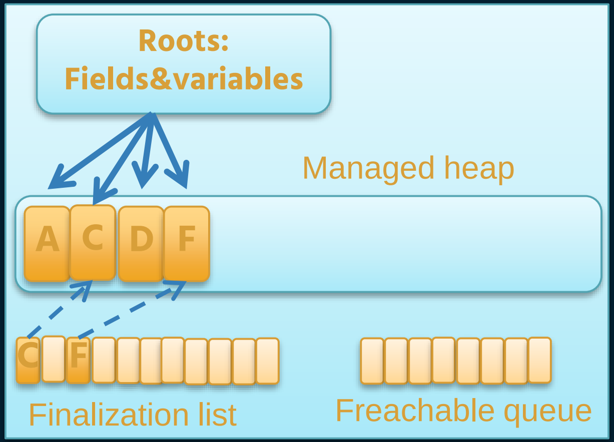
Forced

Optimized

Finalization mechanism

```
internal sealed class SomeType
{
    // Finalization method
     ~SomeType()
     {
       // Finalization code
     }
 }
```


Roots:
Fields&variables
Managed heap
A B C D E F G H I J
C E F I J
Finalization list
Freachable queue
Roots:
Fields&variables
Managed heap
A C D E F I J
C F
E I J
Finalization list
Freachable queue


Roots:
Fields&variables
Managed heap
A
C
D
F
C
F
Finalization list
Freachable queue

Questions :

1. What is a Garbage Collector?
2. When Garbage Collector starts working in .Net?
3. Explain how garbage collection deals with circular references.
4. What are the different generations of Garbage Collection and how do they work ?

2.
IDisposable interface

IDisposable interface provides a mechanism for releasing unmanaged resources.

What is the difference between Dispose and Finalize methods?

Dispose	Finalize
It is used to free unmanaged resources at any time.	It can be used to free unmanaged resources held by an object before that object is destroyed.
It is called by user code and the class which is implementing dispose method, must has to implement IDisposable interface.	It is called by Garbage Collector and cannot be called by user code.
It is implemented by implementing IDisposable interface Dispose() method.	It is implemented with the help of Destructors
There is no performance costs associated with Dispose method.	There is performance costs associated with Finalize method since it doesn't clean the memory immediately and called by GC automatically.

Dispose Pattern

```
using Microsoft.Win32.SafeHandles;
using System;
using System.Runtime.InteropServices;

class BaseClass : IDisposable
{
   // Flag: Has Dispose already been called?
   bool disposed = false;
   // Instantiate a SafeHandle instance.
   SafeHandle handle = new SafeFileHandle(IntPtr.Zero, true);

   // Public implementation of Dispose pattern callable by consumers.
   public void Dispose()
   {
      Dispose(true);
      GC.SuppressFinalize(this);
   }

   // Protected implementation of Dispose pattern.
   protected virtual void Dispose(bool disposing)
   {
      if (disposed)
         return;

      if (disposing) {
         handle.Dispose();
         // Free any other managed objects here.
         //
      }

      // Free any unmanaged objects here.
      //
      disposed = true;
   }
}
```

Dispose Pattern for derived class

```
using Microsoft.Win32.SafeHandles;
using System;
using System.Runtime.InteropServices;

class DerivedClass : BaseClass
{
   // Flag: Has Dispose already been called?
   bool disposed = false;
   // Instantiate a SafeHandle instance.
   SafeHandle handle = new SafeFileHandle(IntPtr.Zero, true);

   // Protected implementation of Dispose pattern.
   protected override void Dispose(bool disposing)
   {
      if (disposed)
         return;

      if (disposing) {
         handle.Dispose();
         // Free any other managed objects here.
         //
      }

      // Free any unmanaged objects here.
      //

      disposed = true;
      // Call base class implementation.
      base.Dispose(disposing);
   }
}
```

Questions :

1. When it is necessary to use Dispose method?
2. After calling Dispose() method for some object, is it immediately removed from memory?
3. What is the difference between Dispose and Finalize methods?

THANKS!

Any questions?