

# Garbage Collector and IDisposable in .NET

The background is a dark navy blue. In the four corners, there are abstract, overlapping geometric shapes. On the left, there are shapes in shades of cyan, orange, and magenta. On the right, there are shapes in shades of cyan, magenta, and orange. These shapes appear to be 3D-like, with some having a slight shadow or gradient effect.

# 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

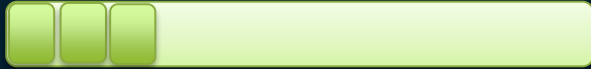
- Virtual address space can get fragmented. This means that there are free blocks, also known as holes, in the address space. When a virtual memory allocation is requested, the virtual memory manager has to find a single free block that is large enough to satisfy that allocation request. Even if you 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.

# Managed heap

- After the garbage collector is initialized by the CLR, it allocates a segment of memory to store and manage objects. This memory is called the managed heap.
- There is a managed heap for each managed process. All threads in the process allocate memory for objects on the same heap.
- To reserve memory, the garbage collector calls the Windows [VirtualAlloc](#) function and reserves one segment of memory at a time for managed applications. The size of segments allocated by the garbage collector is implementation-specific and is subject to change at any time, including in periodic updates. The garbage collector also reserves segments, as needed, and releases segments back to the operating system (after clearing them of any objects) by calling the Windows [VirtualFree](#) function.

# Allocating Resources from the Managed Heap (new operator)

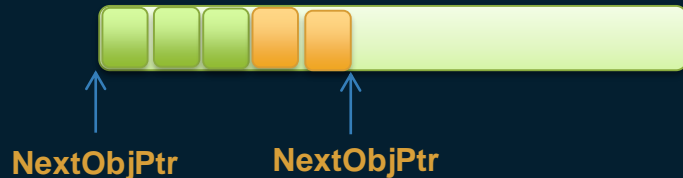
1. Calculate the number of bytes required for the type's fields



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.





# Conditions for garbage collection:

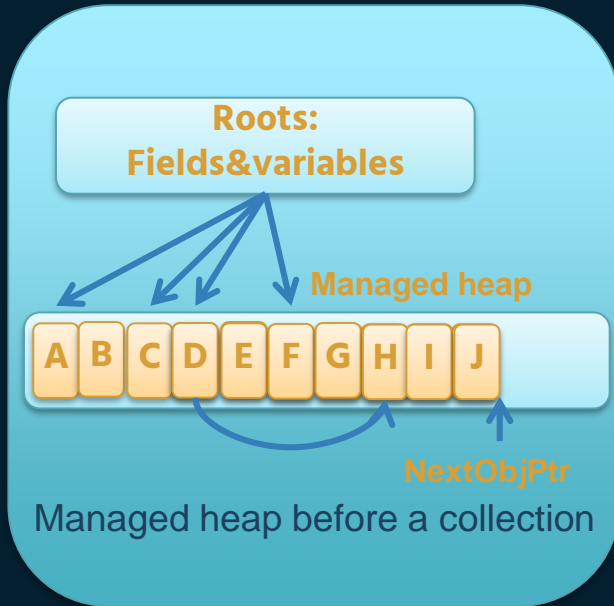
- The system has low physical memory. This is detected by either the low memory notification from the OS or low memory as indicated by the host.
- The memory that's used by allocated objects on the managed heap surpasses an acceptable threshold. This threshold is continuously adjusted as the process 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.

# Memory Release

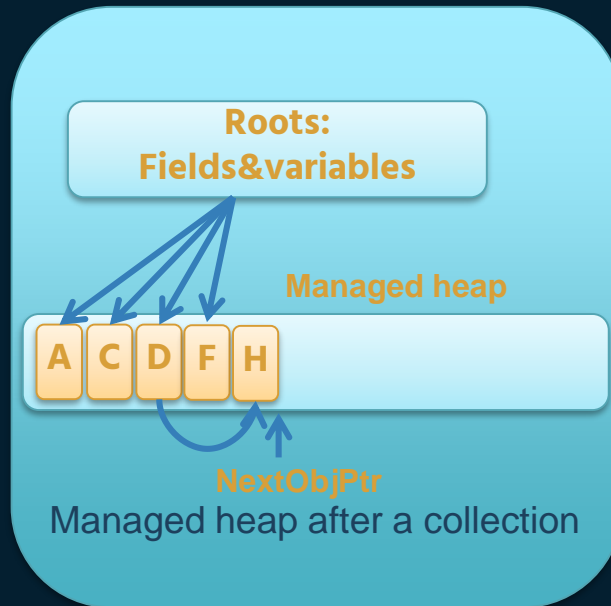
- When the garbage collector performs a collection, it releases the memory for objects that are no longer being used by the application. It determines which objects are no longer being used by examining the application's *roots*:
  - static fields
  - local variables on a thread's stack
  - CPU registers
  - GC handles
  - finalize queue
- Each root either refers to an object on the managed heap or is set to null.
- Using this list, the garbage collector creates a graph that contains all the objects that are reachable from the roots.

# The Garbage Collection Algorithm

## 1. Marking



## 2. Compacting



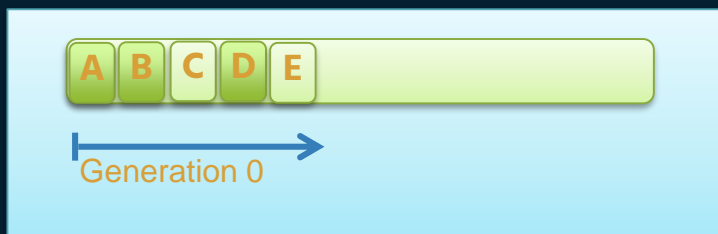
# Generational GC assumptions:

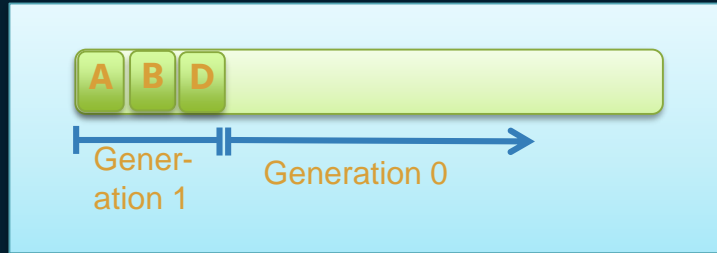
- **The newer an object is, the shorter its lifetime will be.**
- **The older an object is, the longer its lifetime will be.**
- **Collecting a portion of the heap is faster than collecting the whole heap.**

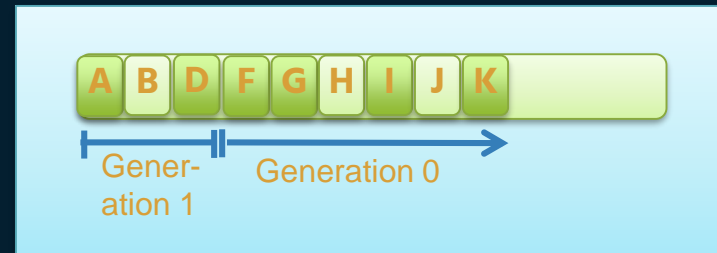
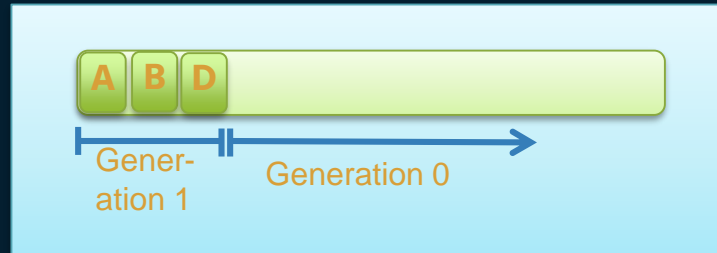
**So** garbage collection primarily occurs with the reclamation of short-lived objects. To optimize the performance of the garbage collector, the managed heap is divided into three generations, 0, 1, and 2, so it can handle long-lived and short-lived objects separately.

# Generations:

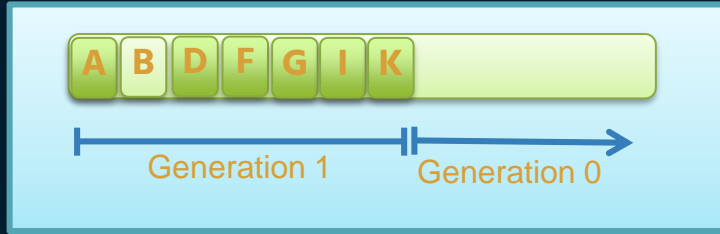
- **Generation 0.** This is the youngest generation and contains short-lived objects. An example of a short-lived object is a temporary variable.
- **Generation 1.** This generation contains short-lived objects and serves as a buffer between short-lived objects and long-lived objects.
- **Generation 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. Objects on the large object heap (which is sometimes referred to as generation 3) are also collected in generation 2

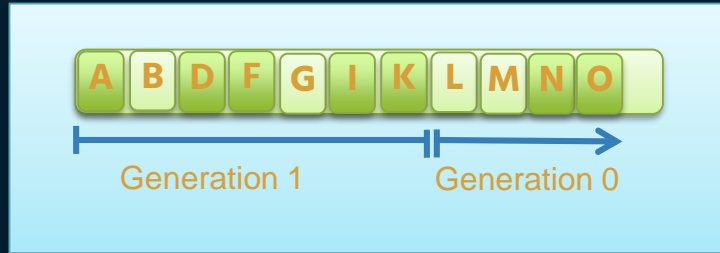
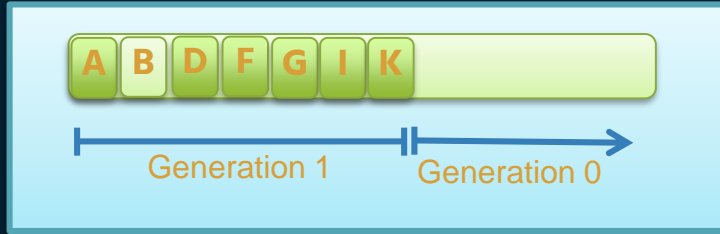


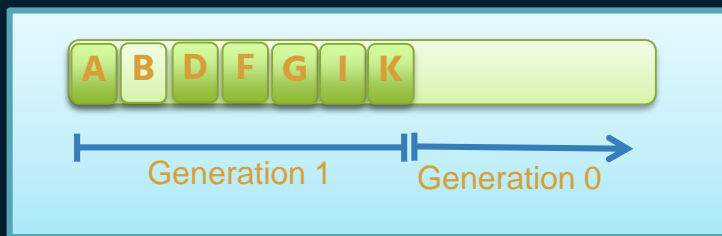




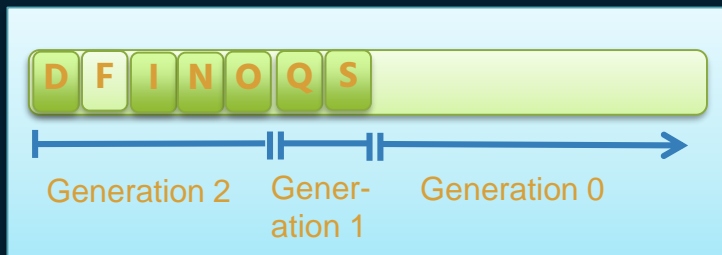
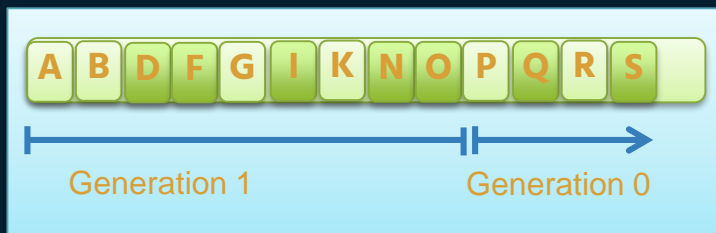












# Large Objects (> 85000b)

- 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. However, in .NET Core and in .NET Framework 4.5.1 and later, you can use the [GCSettings.LargeObjectHeapCompactionMode](#) property to compact the large object heap on demand.
- Large objects are immediately considered to be part of generation 2.

# Forcing Garbage Collections

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



Default

Forced

Optimized

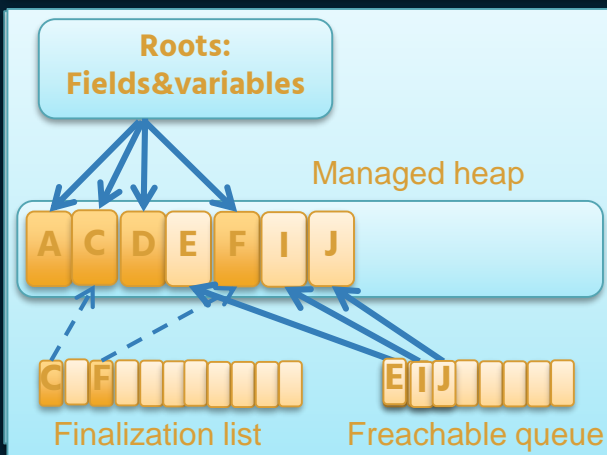
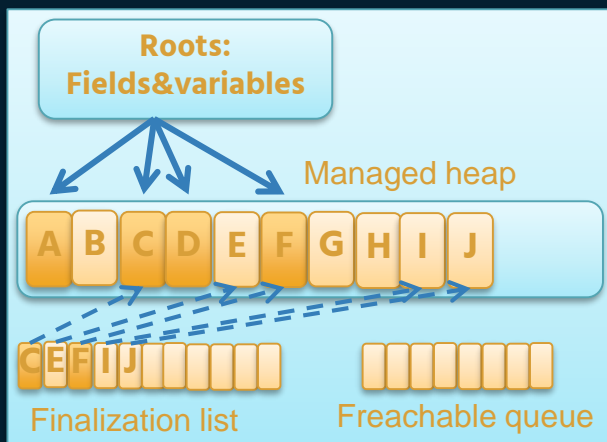
# Unmanaged resources

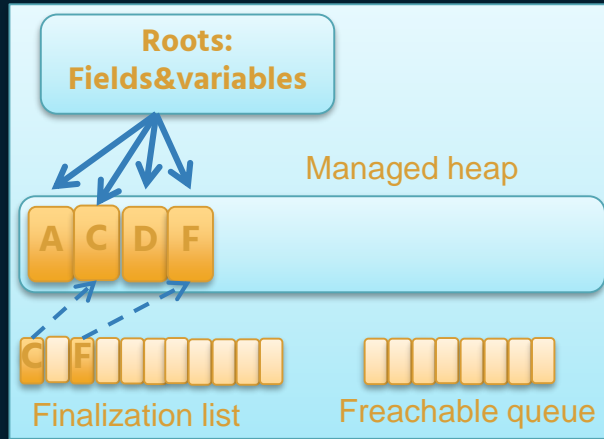
- › Unmanaged resources require explicit cleanup.
  - › The most common type of unmanaged resource is an object that wraps an operating system resource, such as a file handle, window handle, or network connection.
  - › Although the garbage collector is able to track the lifetime of a managed object that encapsulates an unmanaged resource, it doesn't have specific knowledge about how to clean up the resource.
- › Dispose method & [Object.Finalize\(\)](#)



# Finalization mechanism

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





# How finalization works

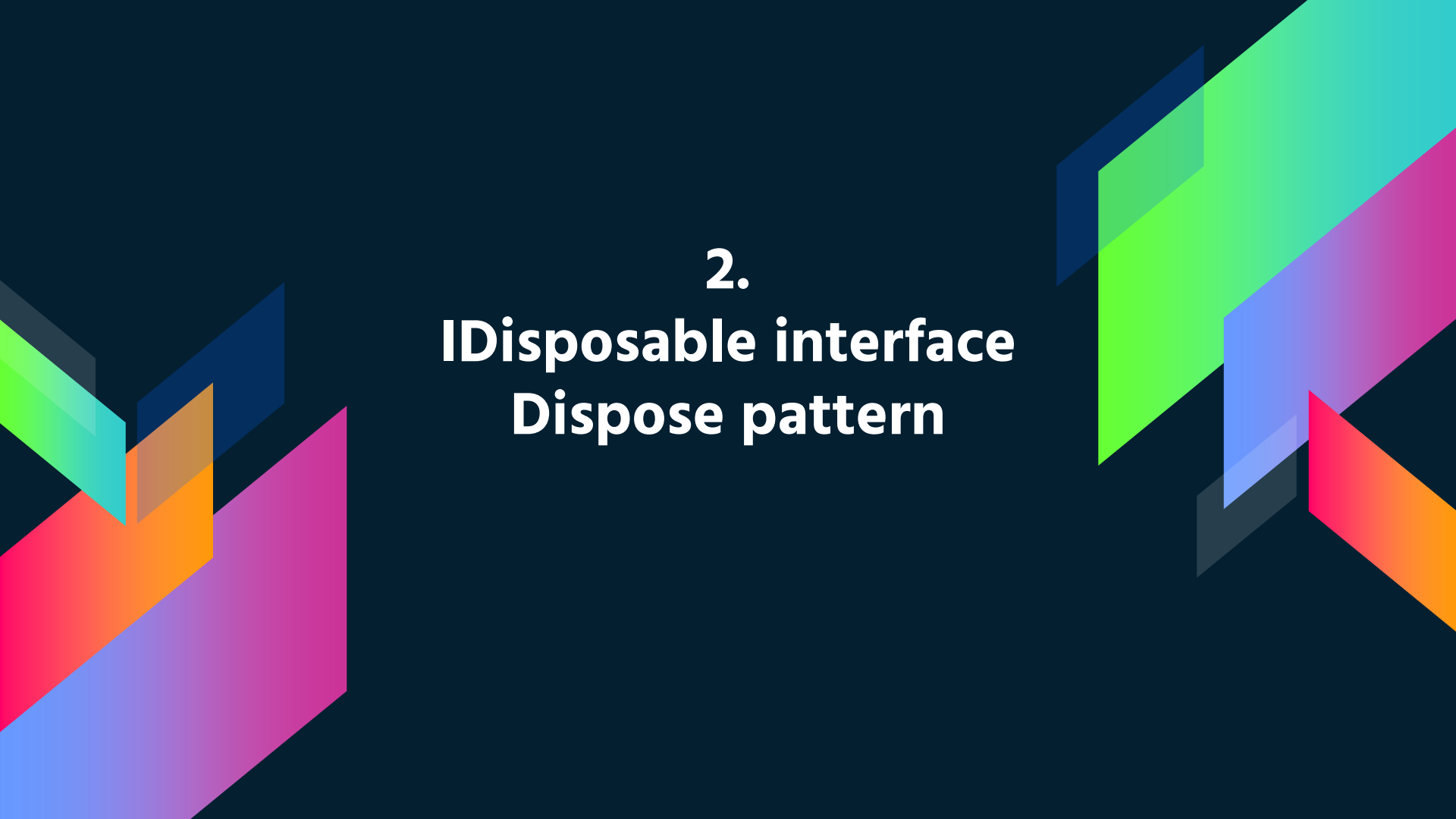
- › The garbage collector then calls the [Finalize](#) method automatically under the following conditions:
  - › After the garbage collector has discovered that an object is inaccessible, unless the object has been exempted from finalization by a call to the [GC.SuppressFinalize](#) method.
  - › **On .NET Framework only**, during shutdown of an application domain, unless the object is exempt from finalization. During shutdown, even objects that are still accessible are finalized.
- › [Finalize](#) is automatically called only once on a given instance, unless the object is re-registered by using a mechanism such as [GC.ReRegisterForFinalize](#) and the [GC.SuppressFinalize](#) method has not been subsequently called.

# How finalization works

- › Finalize operations have the following limitations
  - › The exact time when the finalizer executes is undefined. To ensure deterministic release of resources for instances of your class, implement a Close method or provide a IDisposable.Dispose implementation.
  - › The finalizers of two objects are not guaranteed to run in any specific order, even if one object refers to the other. That is, if Object A has a reference to Object B and both have finalizers, Object B might have already been finalized when the finalizer of Object A starts.
  - › The thread on which the finalizer runs is unspecified.

# 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** **Dispose pattern**

**IDisposable interface** provides a mechanism for releasing unmanaged resources.

#### ▲ Syntax

C#

C++


F#

VB

```
[ComVisibleAttribute(true)]  
public interface IDisposable
```

The IDisposable type exposes the following members.

#### ▲ Methods

	Name	Description
	<code>Dispose</code>	Performs application-defined tasks associated with freeing, releasing, or resetting 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.

# Method Dispose

- › Dispose method differs from destructor in that way that it not destroys the object but destroys the resource
- › Danger Consequences of dispose call: object is not destroyed but resource is not available and any further method call or access to property is potentially dangerous

# Managed and Unmanaged Resources

- › **Unmanaged resources** – IntPtr, socket descriptors, any OS objects obtained with WinAPI etc.
- › If **unmanaged** resource is wrapped into class with RAI (Resource Acquisition Is Initialization) it becomes **managed** resource
- › RAI – means that resource should be allocated in constructor and released in destructor
- › Any of two types of resources implies **different approaches** to work with them

# Sample Resource Wrapper

```
class NativeResourceWrapper : IDisposable
{
    // IntPtr - unmanaged resource descriptor
    private IntPtr nativeResourceHandle;
    public NativeResourceWrapper()
    {
        //Acquiring unmanaged resource
        nativeResourceHandle = AcquireNativeResource();
    }
    public void Dispose()
    {
        // Releasing unmanaged resource
        ReleaseNativeResource(nativeResourceHandle);
    }
    // Finalizer will be explained later
    ~NativeResourceWrapper() {...}
}
```

# Main Idea of Dispose Pattern

The main idea of Dispose Pattern is:

- › Place all logic of resource release into separate method;
- › Call it from Dispose method;
- › Also call it from finalizer;
- › Add special flag that helps to distinguish who exactly (Dispose or Finalizer) called the method.

# Dispose Pattern

1. Class that has both managed and unmanaged resources implements **IDisposable** interface

```
class Boo : IDisposable { ... }
```

2. Method Dispose(bool disposing)
  - › Class contains method **Dispose(bool disposing)** that does all job to release resources;
  - › disposing parameter tells if method is called from **Dispose** method or from **Finalize**. This method should be protected virtual for non-sealed classes and private for sealed classes

# Dispose Pattern

1. Dispose method implementation: first we call **Dispose(true)**, then we may call **GC.SuppressFinalize()** method that suppresses finalizer call:

```
public void Dispose()
{
    Dispose(true /*called by user directly*/);
    GC.SuppressFinalize(this);
}
```

- > Method **Dispose(bool disposing)** has two parts:
  1. If this method called from **Dispose (disposing** parameter is **true**) we should release both managed and unmanaged resources;
  2. If this method is called from finalizer (that is possible under normal circumstances only during garbage collection process when **disposing** parameter is **false**), we release only unmanaged resources.

# Finalizer

- › [OPTIONAL] Class may have finalizer and call **Dispose(bool disposing)** from it passing **false** as parameter.

```
~Boo()  
{  
    Dispose(false /*not called by user directly*/);  
}
```

- › Also we should take into account that finalizer may be called even for partially constructed classes, if constructor for such class raises an exception. That is why resource releasing code should handle situation when resources are not allocated yet



# Field “disposed”

- The good practice is to create special Boolean field **disposed** which indicates that object's resources are released.
- Disposable objects should allow **any** number of **Dispose()** method calls and generate an exception when any public member of the object is accessed after first call to the method (when **dispose flag** is set to true).

```
void Dispose(bool disposing)
{
    if (disposed)
        return; // Resources are already released
    // Releasing resources
    disposed = true;
}
public void SomeMethod()
{
    if (disposed)
        throw new ObjectDisposedException();
}
```

# 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;
    }
}
```

# Questions :Simplifying Dispose Pattern

- › Most difficulties with Dispose pattern implementation based on assumption that same class (or class hierarchy) may contain managed and unmanaged resources at the same time
- › But Single Responsibility Principle (SRP) suggest us that we do not mix resources of different kinds
- › RAII idiom suggests a solution: **if you have unmanaged resource, do not use it directly, wrap it into managed wrapper and work with it**

# 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);
    }
}
```

# Keyword using

```
// Opening file inside using block
using (FileStream file =
File.OpenRead("foo.txt"))
{
    // Leaving method on condition
    if (someCondition) return;
    // File closes automatically
}

// What if file opens outside using block?
FileStream file2 = File.OpenRead("foo.txt");
```

# 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?