12214994SaurabhRana

C# Programming – Lecture - 10

Garbage Collection in C#

---

1. Garbage Collection (GC) – Overview

Garbage Collection (GC) in C# is an **automatic memory management mechanism** implemented by the **.NET Common Language Runtime (CLR)**.
It is responsible for **allocating memory for objects**, **monitoring their lifetime**, and **reclaiming memory** occupied by objects that are no longer in use (i.e., objects that are no longer referenced by the application).

In .NET, developers work with **managed memory**, meaning they do not explicitly allocate or free memory. The GC continuously runs in the background and ensures that memory is used efficiently and safely throughout the application's lifecycle.

In simple terms:
GC automatically removes unused objects from memory so developers don't have to.

Why Garbage Collection Is Needed

Before garbage collection systems existed, programmers had to manage memory manually (as in C or C++). This introduced several serious problems:

Problems Without GC

- **Manual memory allocation and deallocation:** Developers must remember exactly when to free memory.
- **Memory Leaks:** Memory is allocated but never released, causing applications to consume more memory over time.
- **Dangling Pointers:** Memory is freed but still referenced, leading to undefined behavior or crashes.

- **Double Free Errors:** Freeing the same memory twice can corrupt the application.
- **Difficult Debugging:** Memory-related bugs are hard to detect and fix.
- **Unstable Applications:** Long-running apps often crash or slow down due to poor memory handling.

How Garbage Collection Solves These Issues

GC automatically handles all memory-related responsibilities:

✓ Automatic Memory Cleanup

Unused objects are identified and removed automatically.

✓ Safe Memory Usage

Prevents access to invalid memory locations.

✓ Reduced Memory Leaks

Objects are freed once they are no longer reachable.

✓ Improved Application Stability

Applications can run for long periods without crashing.

✓ Performance Optimization

GC uses advanced algorithms to minimize overhead.

✓ Higher Developer Productivity

Developers focus on business logic instead of memory management.

When Garbage Collection Works

Garbage Collection is triggered:

- When available memory becomes low
- When many short-lived objects are created
- During application idle time
- Automatically by the CLR (developer intervention not required)

Calling `GC.Collect()` manually is generally discouraged unless absolutely necessary.

Where Garbage Collection Is Applied

Garbage Collection is used in **all managed .NET environments**, including:

Desktop Applications

- WinForms
- WPF
- Console Applications

Used for managing UI objects, event handlers, and background tasks.

Web Applications

- ASP.NET
- ASP.NET Core

Manages:

- HTTP request/response objects
- Middleware components
- Session data
- Dependency Injection services

Cloud & Microservices

- Azure Functions
- Kubernetes-hosted .NET services
- Background workers

GC ensures:

- Efficient scaling
- Stability under high load

- Reduced memory footprint

Games

- **Unity (C#)**

Handles:

- Game objects
- Physics components
- Animation objects

Efficient GC helps avoid frame drops and performance spikes.

Mobile Applications

- **Xamarin**
- **.NET MAUI**

Manages:

- UI components
- Navigation stacks
- Background services

Critical for battery efficiency and smooth user experience.

Real-World Example

```
class Customer

{

    public string Name;

}


static void Main()

{

    for (int i = 0; i < 1000; i++)

    {

        Customer c = new Customer();

    }

}
```

✓ Objects are created

✓ No manual deletion required

✓ GC automatically frees unused objects

---

3. Key Features of the Garbage Collector

Automatic Memory Deallocation

- GC automatically deallocates memory occupied by **unreachable objects**
- An object becomes unreachable when no active references point to it

---

Prevents Memory Leaks

- Objects no longer in use are cleaned up automatically
- Reduces risk of memory exhaustion in long-running apps

Heap Compaction

- After removing unused objects, GC **compacts memory**
- Moves live objects together
- Reduces memory fragmentation
- Improves allocation speed

Generational Memory Management

GC divides objects into **three generations** based on lifetime:

- Gen 0
- Gen 1
- Gen 2

This improves performance by focusing cleanup on short-lived objects.

Background Execution

- GC runs in the background
- Minimizes application pauses

- Uses optimized algorithms for low latency

---

4. How Does Garbage Collection Work?

The Garbage Collector follows a structured process:

Step 1: Object Allocation

- Objects are allocated on the **managed heap**
- New objects start in **Generation 0**

Step 2: Reachability Analysis

- GC checks which objects are **still referenced**
- Uses roots such as:
    - Local variables
    - Static variables
    - CPU registers
    - Active stack frames

Step 3: Object Collection

- Objects not reachable are marked for deletion
- Their memory is reclaimed

Step 4: Memory Compaction

- Remaining objects are moved together
- Heap is compacted
- Reduces fragmentation

Step 5: Promotion

- Surviving objects move to higher generations
- Longer lifetime = higher generation

---

5. Generations in Garbage Collection

The .NET GC uses **generational garbage collection** because **most objects die young**.

Generations Explained

Generation	Description	Example
Gen 0	Short-lived objects, collected frequently	Local variables, temporary objects
Gen 1	Medium-lived objects	Objects surviving Gen 0
Gen 2	Long-lived objects, collected rarely	Static data, caches, config data

---

Why Generations Improve Performance

- Faster collections

- Less CPU usage
- Shorter application pauses
- Efficient memory reuse

---

6. Example: Forcing Garbage Collection

Although GC runs automatically, it can be triggered manually.

Code Example

```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Creating objects...");
        for (int i = 0; i < 5; i++)
        {
            MyClass obj = new MyClass();
        }

        Console.WriteLine("Forcing garbage collection...");
        GC.Collect();
        GC.WaitForPendingFinalizers();

        Console.WriteLine("Garbage collection completed.");
    }
}

class MyClass
{
    ~MyClass()
    {
        Console.WriteLine("Finalizer called, object collected.");
    }
```

```
}
```

Code Breakdown

Line	Explanation
`GC.Collect()`	Forces garbage collection
`GC.WaitForPendingFinalizers()`	Waits for finalizers
`~MyClass()`	Finalizer confirms collection

GC.Collect();

- Explicitly requests the CLR Garbage Collector to run
- It:
    - Scans memory
    - Identifies unreachable objects
    - Marks them for collection

Important rule:

- GC is non-deterministic
- Calling GC.Collect() does NOT guarantee immediate cleanup

- It is a request, not a command

GC.WaitForPendingFinalizers();

- Waits until all finalizers are executed
- Ensures:
    - ~MyClass() is called before proceeding
- Without this line:
    - Program might exit before finalizers run

---

6. Final Output Message

Console.WriteLine("Garbage collection completed.");

- Printed after all finalizers have executed
- Confirms GC cycle is done

---

7. MyClass and Finalizer

class MyClass

- Simple class with no fields
- Used only to demonstrate GC behavior

~MyClass()

This is a finalizer (destructor-like method).

Key points:

- Automatically called by GC
- Executes before memory is reclaimed
- Used for:
    - Cleaning unmanaged resources
    - Logging object destruction

Equivalent internal form:

protected override void Finalize()

Important Note

⚠ **Manual GC is discouraged** because:

- It affects performance
- It interrupts optimized GC behavior
- CLR knows best when to collect

What is a Finalizer in GC?

A finalizer is a special method in a class that is automatically called by the Garbage Collector just before an object's memory is removed from the heap.

In simple words:

A finalizer is the last chance an object gets to clean up resources before it is destroyed by GC.

Simple Definition (Exam-Friendly)

A finalizer is a method used to release unmanaged resources of an object when it is collected by the Garbage Collector.

How Do We Write a Finalizer?

In C#, a finalizer looks like a destructor:

```
class MyClass

{

   ~MyClass()

   {

      // cleanup code

   }

}
```

Important:

- The name is always ~ClassName
- No parameters
- No return type
- Cannot be called manually

When Is a Finalizer Called?

1. Object becomes unreachable
2. GC detects the object
3. Object is placed in the Finalization Queue
4. GC calls the finalizer
5. Memory is released later

Important:

- Finalizer is NOT called immediately
- Timing is non-deterministic.

Why Do We Need Finalizers?

Finalizers are mainly used to:

- Release unmanaged resources

- Act as a backup safety mechanism
- Prevent OS-level resource leaks

Examples of unmanaged resources:

- File handles
- Database connections
- Network sockets
- OS memory

What Finalizers Cannot Do

Finalizers:

- Cannot access other managed objects reliably
- Cannot throw exceptions
- Should not contain heavy logic
- Should not depend on execution order

Finalizer vs Dispose()

Feature	Finalizer	Dispose

Called by	GC	Developer
Timing	Unpredictable	Immediate
Deterministic	No	Yes
Performance	Slow	Fast
Recommended	Backup only	Primary

Example Demonstrating Finalizer

```
class Sample
{
   ~Sample()
   {
      Console.WriteLine("Finalizer executed");
   }
}
```

Execution flow:

- Object created
- Reference lost
- GC runs

- Finalizer executes
- Memory freed

Why Finalizers Are Discouraged

Finalizers:

- Delay garbage collection
- Increase memory pressure
- Run on a separate thread
- Can cause performance issues

That’s why:

- Microsoft recommends IDisposable over finalizers

Best Practice (Very Important)

If you use a finalizer:

- Always combine it with IDisposable
- Suppress finalization after cleanup

```
public void Dispose()
{
```

```
    // cleanup
    GC.SuppressFinalize(this);
}
```

Key Interview Points

- Finalizers run automatically by GC
- They clean unmanaged resources
- They are non-deterministic
- Use them only as a safety net
- Prefer Dispose() whenever possible

One-Line Summary

A finalizer is a GC-invoked method that cleans up unmanaged resources before an object is destroyed.

---

7. When Should You Use Garbage Collection Manually?

Manual GC should be used **rarely**, only in special cases:

✓ After releasing large temporary objects

✓ Before entering memory-critical operations

✓ During application shutdown

❌ Not inside loops

❌ Not in high-frequency code paths

---

8. Using IDisposable to Manage Resources

GC handles **managed memory**, but **unmanaged resources** must be cleaned explicitly.

What Are Unmanaged Resources?

- File handles
- Database connections
- Network sockets
- OS handles

IDisposable Interface

The `IDisposable` interface provides a standard way to release unmanaged resources.

Is IDisposable pre-built?

Yes.

IDisposable is a built-in interface provided by .NET.

- It is defined by Microsoft

- It comes with the .NET Base Class Library (BCL)
- You do not create it yourself

Where is IDisposable defined?

```
namespace System
{
    public interface IDisposable
    {
        void Dispose();
    }
}
```

Key points:

- It belongs to the System namespace
- Automatically available when you use:

```
using System;
```

Do we need to install anything?

No.

- Comes with:

    - .NET Framework
    - .NET Core
    - .NET 5+
- Available in all versions of C#

Why is it provided by .NET?

Because:

- Resource cleanup is a common requirement
- GC cannot manage unmanaged resources
- Microsoft standardized cleanup using IDisposable

So instead of everyone writing different cleanup methods:

- .NET provides one standard interface

Real Proof: Pre-built Classes Using IDisposable

All these built-in .NET classes implement IDisposable:

- FileStream

- StreamReader
- StreamWriter
- SqlConnection
- SqlCommand
- Bitmap
- MemoryStream

This proves it is core to .NET design.

What do WE do with IDisposable?

We implement it in our own classes when needed.

Example:

```
class MyResource : IDisposable
{
    public void Dispose()
    {
        Console.WriteLine("Resources released");
    }
}
```

We are not creating the interface — only using it.

Exam One-Line Answer

Yes, IDisposable is a pre-built interface defined in the System namespace of .NET and is used for deterministic resource cleanup.

Interview One-Line Answer

IDisposable is a built-in .NET interface that allows developers to explicitly release unmanaged resources using the Dispose method.

Very Important Clarification

Item	Created by
IDisposable interface	Microsoft (.NET)
Dispose() method implementation	Developer
Calling Dispose()	Developer / using
Garbage Collection	CLR

Example: Implementing IDisposable

```
using System;

class ResourceHandler : IDisposable
{
    public ResourceHandler()
    {
```

```
        Console.WriteLine("Resource acquired.");
    }

    public void Dispose()
    {
        Console.WriteLine("Resource released.");
    }
}
```

By using “Using” Statement (Best Practice)

```
class Program
{
    static void Main()
    {
        using (ResourceHandler handler = new ResourceHandler())
        {
            Console.WriteLine("Using resource...");
        } // Dispose() called automatically

        Console.WriteLine("End of program.");
    }
}
```

1. What is the purpose of this code?

This program demonstrates:

- The using statement
- Automatic resource cleanup

- The IDisposable pattern
- How Dispose() is called even if no exception occurs

In short:

---

2. Entry Point –

Main()

static void Main()

- Program execution starts here
- static → CLR can call it without creating an object
- void → no return value

---

3. The using Statement (MOST IMPORTANT PART)

using (ResourceHandler handler = new ResourceHandler())

What does using mean?

The using statement means:

Even if:

- An exception occurs
- The block exits early

---

4. What happens step-by-step at runtime

Step 1: Object Creation

ResourceHandler handler = new ResourceHandler();

- Memory allocated on the heap
- handler (reference) stored on the stack
- Constructor of ResourceHandler is executed
- Resource is acquired (file, DB, stream, etc.)

Step 2: Enter

using

block

```
{
   Console.WriteLine("Using resource...");
}
```

- Resource is actively being used
- Any logic that needs the resource goes here

Output so far:

Using resource...

Step 3: Exiting the using block

When execution reaches this line:

} // Dispose() called automatically

C# automatically calls:

handler.Dispose();

You do not write this call yourself.

---

5. What using actually compiles into (VERY IMPORTANT)

The compiler rewrites your code like this:

```
ResourceHandler handler = new ResourceHandler();
try
{
   Console.WriteLine("Using resource...");
}
finally
{
   if (handler != null)
      handler.Dispose();
}
```

Why finally?

- finally always executes
- Guarantees resource cleanup
- Works even if an exception occurs

This is why using is exception-safe.

---

6. Why ResourceHandler must implement IDisposable

Rule

Example implementation:

```
class ResourceHandler : IDisposable
{
    public void Dispose()
    {
        Console.WriteLine("Resource released");
    }
}
```

If IDisposable is missing → compile-time error

---

7. After using block completes

```
Console.WriteLine("End of program.");
```

- Resource is already released
- Safe to continue execution
- No memory/resource leak

Final output:

```
Using resource...
Resource released
End of program.
```

(assuming Dispose() prints a message)

---

8. Why this pattern is extremely important

Without

using

```
ResourceHandler handler = new ResourceHandler();
// forgot to call Dispose()
```

Problems:

- File remains locked
- DB connection stays open
- Memory leaks
- Application slowdown or crash

With using

- Cleanup is guaranteed
- Code is clean
- No forgotten Dispose() calls
- Industry best practice

9.

using vs Garbage Collection (Important Difference)

using / Dispose	Garbage Collector
Immediate cleanup	Non-deterministic
Manual control	Automatic
Releases unmanaged resources	Handles managed memory
Recommended	Not enough alone

---

10. Exam-Ready One-Line Answer

The using statement ensures that an object’s Dispose() method is automatically called to release resources when the block ends.

---

11. Interview-Ready Explanation (Short)

using is syntactic sugar for a try–finally block that guarantees deterministic cleanup of unmanaged resources through IDisposable.

---

12. Key Takeaways (Remember These)

- using calls Dispose() automatically
- Works only with IDisposable
- Prevents resource leaks

- Exception-safe
- Preferred over finalizers
- Cleaner than manual disposal

---

13. Mental Model (Very Important)

Create Resource

↓

Use Resource

↓

Dispose Automatically

↓

Continue Program

Why `using` Is Important

✓ Ensures deterministic cleanup

✓ Prevents resource leaks

✓ Cleaner and safer code

---

9. Best Practices for Garbage Collection

- Let the GC manage memory automatically
- Avoid frequent `GC.Collect()` calls

- Use `IDisposable` for unmanaged resources
- Always use `using` blocks
- Keep object lifetimes short
- Use memory profilers for optimization

---

10. Real-World Application Scenario

Web Application (ASP.NET Core)

- GC cleans request objects after response
- Frees memory between requests
- Prevents server memory exhaustion
- Ensures scalability

Banking or Enterprise System

- Manages large datasets
- Keeps long-lived objects in Gen 2
- Uses Dispose for DB connections
- Ensures 24/7 uptime

Garbage Collection Algorithm

The **Garbage Collection (GC) Algorithm in C#** is an **automatic memory management mechanism** implemented by the **.NET Common Language Runtime (CLR)**.
Its primary responsibility is to **identify objects that are no longer reachable**, **reclaim the**

memory occupied by them, and **optimize memory usage** so that applications run efficiently and reliably.

The GC algorithm eliminates the need for manual memory deallocation, reduces memory-related bugs, and improves application performance.

---

2. Why the Garbage Collection Algorithm Is Needed

Without a GC algorithm:

- Developers must manually manage memory
- High chances of memory leaks
- Dangling pointers cause unpredictable crashes
- Heap fragmentation reduces performance
- Long-running applications become unstable

With the GC algorithm:

✓ Automatic memory reclamation
✓ Safe and predictable memory usage
✓ Improved application performance
✓ Reduced memory fragmentation
✓ Better scalability for enterprise applications

---

3. Key Features of the Garbage Collection Algorithm

Automatic Memory Cleanup

- Automatically frees memory occupied by **unreachable objects**
- Objects are reclaimed when no active references exist

Generational Optimization

- Works using **three generations (Gen 0, Gen 1, Gen 2)**
- Focuses cleanup on short-lived objects
- Reduces the cost of full heap scans

Heap Compaction

- Rearranges live objects after cleanup
- Eliminates memory gaps
- Improves allocation speed

Background Execution

- Runs in the background
- Minimizes application pauses
- Uses concurrent and server GC modes

High Performance

- Optimized for multi-core systems
- Efficient for both small and large applications

---

4. How the Garbage Collection Algorithm Works

The .NET Garbage Collector uses a **Mark–Sweep–Compact algorithm** combined with **generational collection**.

Step 1: Mark Phase

- GC identifies all **reachable (alive) objects**
- Starts from **GC Roots**, such as:
    - Local variables
    - Static fields
    - CPU registers
    - Thread stacks

✓ Reachable objects are marked as alive

❌ Unreachable objects are candidates for deletion

Step 2: Sweep Phase

- GC scans the heap
- Removes all **unmarked (dead) objects**
- Memory occupied by these objects is reclaimed

Step 3: Compact Phase

- Live objects are moved together
- Empty memory spaces are removed

- Heap becomes contiguous

✓ Reduces fragmentation

✓ Improves future allocation speed

---

5. Generational Garbage Collection

The GC algorithm divides objects into generations based on **object lifetime**.

Why Generations Exist

> Most objects die young.

Generations Explained

Generation	Description	Example
Gen 0	Newly allocated, short-lived objects	Local variables, temp objects
Gen 1	Objects surviving Gen 0	Short-lived cached objects
Gen 2	Long-lived objects	Static objects, application caches

Promotion Process

- Objects surviving Gen 0 → promoted to Gen 1
- Objects surviving Gen 1 → promoted to Gen 2
- Gen 2 is collected less frequently

Large Object Heap (LOH)

- Objects larger than **85 KB**
- Stored separately
- Collected less often
- Not compacted frequently (older .NET versions)

---

6. Example: Garbage Collection in Action

Demonstrating Generational GC

```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine($"Total Memory Before GC:
{GC.GetTotalMemory(false)} bytes");

        for (int i = 0; i < 10000; i++)
        {
            object obj = new object(); // Gen 0 allocation
        }

        Console.WriteLine($"Total Memory After Object Creation:
{GC.GetTotalMemory(false)} bytes");

        GC.Collect();
        GC.WaitForPendingFinalizers();

        Console.WriteLine($"Total Memory After GC:
{GC.GetTotalMemory(false)} bytes");
        Console.WriteLine($"Generation of a new object:
{GC.GetGeneration(new object())}");
```

```
    }
}
```

Code Breakdown

Line	Explanation
`new object()`	Allocated in Gen 0
`GC.Collect()`	Forces GC execution
`GC.GetGeneration()`	Shows object generation

Key Observation

- Temporary objects are cleaned quickly
- Memory usage drops after GC
- New objects always start in Gen 0

---

1. Purpose of This Program (Big Picture)

This program demonstrates:

- How memory increases when objects are created
- How Garbage Collection frees memory
- How GC generations (Gen 0, Gen 1, Gen 2) work

- That new objects start in Generation 0

---

2. Understanding

GC.GetTotalMemory(false)

GC.GetTotalMemory(false)

What it does

- Returns the approximate number of bytes currently allocated
- Includes managed heap memory only
- Does not include stack memory

Why false?

- false → do not force garbage collection
- If true → GC runs first (expensive)

Line Explanation

Console.WriteLine($"Total Memory Before GC: {GC.GetTotalMemory(false)} bytes");

- Prints memory usage before creating objects

- Represents baseline memory

---

3. Object Creation Loop (Gen 0 Allocation)

```
for (int i = 0; i < 10000; i++)
{
    object obj = new object(); // Gen 0 allocation
}
```

What happens here?

- Each new object():
    - Allocates memory on the managed heap
    - Goes into Generation 0

Important detail

object obj = new object();

- obj is a local variable
- Goes out of scope immediately after each iteration
- Objects become eligible for garbage collection

Why Generation 0?

Because:

- Most objects die quickly
- Gen 0 is optimized for fast allocation and cleanup

---

4. Memory After Object Creation

Console.WriteLine($"Total Memory After Object Creation: {GC.GetTotalMemory(false)} bytes");

What this shows

- Increased memory usage
- Objects are still in memory
- GC has not yet reclaimed them

Even though:

- Objects are unreachable
- GC hasn’t run yet

---

5. Forcing Garbage Collection

GC.Collect();

What this does

- Forces a full garbage collection
- Collects:
    - Gen 0
    - Gen 1
    - Gen 2

⚠️ Normally:

- You should not call this in production
- GC is designed to run automatically

Waiting for Finalizers

GC.WaitForPendingFinalizers();

Why this is needed

- Some objects may have finalizers
- GC puts them in the finalization queue
- This line:
    - Waits until all finalizers complete

- Ensures memory is fully reclaimed

---

6. Memory After GC

Console.WriteLine($"Total Memory After GC: {GC.GetTotalMemory(false)} bytes");

What you observe

- Memory usage drops significantly.
- Dead objects are reclaimed.
- Heap becomes compacted.

Important:

- Memory may not return to exact initial value
- CLR keeps some memory reserved for reuse

---

7. Checking Object Generation

Console.WriteLine($"Generation of a new object: {GC.GetGeneration(new object())}");

What happens here

- A new object is created

- GC checks its generation

Output

Generation of a new object: 0

Why?

---

8. Garbage Collection Generations Explained (Very Important)

Generation	Purpose	Characteristics
Gen 0	Short-lived objects	Fast collection
Gen 1	Medium-lived objects	Survivor buffer
Gen 2	Long-lived objects	Slow but rare

Promotion Rule

- Objects surviving Gen 0 GC → promoted to Gen 1
- Objects surviving Gen 1 GC → promoted to Gen 2

- Gen 2 objects:
    - Collected rarely
    - Used for long-living data

---

9. Why GC Uses Generations

Because:

- Most objects die young
- Checking all memory every time is slow
- Generations optimize performance

This is called the Generational Hypothesis.

---

10. Output (Typical Example)

Total Memory Before GC: 20000 bytes

Total Memory After Object Creation: 500000 bytes

Total Memory After GC: 25000 bytes

Generation of a new object: 0

(Exact values vary per machine/runtime)

---

11. Important Exam & Interview Points

Key Facts

- GC manages managed heap only
- Stack memory is auto-cleared
- New objects → Gen 0
- GC is non-deterministic
- GC.Collect() should be avoided in real apps

Common Interview Question

Q: Why does memory not drop to zero after GC?

Answer:

CLR keeps reserved memory for future allocations to improve performance.

---

12. One-Line Exam Answers

- GC.GetTotalMemory → Returns allocated managed heap memory
- Gen 0 → All new objects start here
- GC.Collect() → Forces garbage collection
- GC.WaitForPendingFinalizers() → Waits for finalizers to finish

---

13. Mental Model (Very Important)

Create Objects → Gen 0

Lose References

GC Runs

Dead Objects Removed

Memory Reclaimed

---

7. Using Weak References with Garbage Collection

Definition

A **WeakReference** allows the GC to collect an object even if it is still referenced weakly.

Why WeakReference Is Needed

- Prevents memory leaks in caching scenarios
- Allows GC to reclaim memory when needed
- Improves memory efficiency

Example: WeakReference

```
using System;

class Program
{
```

```
    static void Main()
    {
        WeakReference weakRef = new WeakReference(new object());

        Console.WriteLine($"Is Object Alive: {weakRef.IsAlive}");

        GC.Collect();

        Console.WriteLine($"Is Object Alive After GC:
{weakRef.IsAlive}");
    }
}
```

Real-World Use Case

- Image caching
- Large object caching
- Temporary data storage

---

8. When the GC Algorithm Runs

GC is triggered when:

- Heap memory is low
- Many Gen 0 allocations occur
- Application is idle
- Explicit `GC.Collect()` is called

---

9. Best Practices for Garbage Collection

✓ Avoid frequent `GC.Collect()`

✓ Keep object lifetimes short

✓ Use `IDisposable` for unmanaged resources

✓ Use weak references for caches

✓ Monitor memory using profilers

✓ Avoid unnecessary large objects

---

10. Real-World Application Example

Web Application (ASP.NET Core)

- Each HTTP request creates temporary objects
- GC cleans them after request completion
- Keeps server memory stable under load

Gaming Application

- Short-lived game objects → Gen 0
- Scene objects → Gen 2
- Weak references for assets

---

11. Comparison with Manual Memory Management (C++)

Feature	C# GC Algorithm	C++ Manual Memory
Memory cleanup	Automatic	Manual

Safety	High	Error-prone
Memory leaks	Rare	Common
Performance	Optimized	Depends on developer
Ease of use	Easy	Complex

Comparison with Reference Counting

Feature	Generational GC	Reference Counting
Cyclic references	Supported	❌
Performance	High	Lower
Manual cleanup	❌	Required
Used by .NET	✓	❌

---

4. Finalize vs Dispose

In C#, **memory management** is handled automatically by the **Garbage Collector (GC)**.
However, **resource management**—especially for **unmanaged resources**—requires special handling.
This is where **Finalize** and **Dispose** come into play.

Although both are used for cleanup, they serve **different purposes**, follow **different lifecycles**, and have **different performance implications**.

---

2. What Is Finalize?

Definition

`Finalize()` (also called a **finalizer**) is a special method that is **automatically invoked by the Garbage Collector** before an object is removed from memory.

It is used as a **last-resort cleanup mechanism** for unmanaged resources when the developer fails to release them explicitly.

> In C#, a finalizer is written using a destructor syntax: `~ClassName()`.

Syntax

```
class ResourceHandler
{
    ~ResourceHandler()
    {
        // Cleanup unmanaged resources
    }
}
```

Why Finalize Is Needed

- Acts as a **safety net**
- Ensures unmanaged resources are eventually released
- Protects against programmer mistakes (forgotten Dispose)

When Finalize Is Applied

- When an object holds unmanaged resources
- When Dispose was not called
- When GC decides to collect the object

Where Finalize Is Used

- Wrappers around native APIs
- File handles
- OS resources
- Interop scenarios (P/Invoke)

Limitations of Finalize

❌ Execution time is **non-deterministic**
❌ Runs on GC finalizer thread
❌ Slower performance
❌ Increases GC overhead
❌ Delays memory reclamation

Example: Using Finalize

```
using System;

class ResourceHandler
{
    public ResourceHandler()
    {
        Console.WriteLine("Resource acquired.");
    }

    ~ResourceHandler()
    {
        Console.WriteLine("Finalizer called, resource released.");
    }
}
```

```
class Program
{
    static void Main()
    {
        ResourceHandler handler = new ResourceHandler();
        handler = null;

        GC.Collect();
        GC.WaitForPendingFinalizers();

        Console.WriteLine("End of program.");
    }
}
```

Code Breakdown

Line	Explanation
`~ResourceHandler()`	Finalizer method
`GC.Collect()`	Forces GC
`GC.WaitForPendingFinalizers()`	Waits for finalizers

---

3. What Is Dispose?

Definition

`Dispose()` is a **developer-controlled cleanup method** defined by the `IDisposable` interface.
It provides a **deterministic way** to release resources **immediately**.

Syntax

```
class ResourceHandler : IDisposable
{
    public void Dispose()
    {
        // Release resources
    }
}
```

Why Dispose Is Needed

- Immediate release of critical resources
- Predictable cleanup timing
- Better performance
- Essential for unmanaged resources

When Dispose Is Applied

- File operations
- Database connections
- Network sockets
- Streams
- Locks

Where Dispose Is Used

- Enterprise applications
- Web APIs

- Cloud services
- Desktop and mobile apps

Example: Using Dispose

```
using System;

class ResourceHandler : IDisposable
{
    public ResourceHandler()
    {
        Console.WriteLine("Resource acquired.");
    }

    public void Dispose()
    {
        Console.WriteLine("Dispose method called, resource
released.");
    }
}

class Program
{
    static void Main()
    {
        using (ResourceHandler handler = new ResourceHandler())
        {
            Console.WriteLine("Using resource...");
        }

        Console.WriteLine("End of program.");
    }
}
```

Code Breakdown

Line	Explanation
`IDisposable`	Enables Dispose
`using`	Ensures Dispose is called
`Dispose()`	Releases resources

---

4. Key Differences Between Finalize and Dispose

Feature	Finalize	Dispose
Called by	Garbage Collector	Developer
Explicit call	❌	✓
Deterministic	❌	✓
Performance	Slow	Fast
Execution time	Unpredictable	Immediate
using support	❌	✓
Best practice	❌	✓

---

5. Combining Finalize and Dispose (Dispose Pattern)

Why Combine Them?

- Dispose → Primary cleanup

- Finalize → Backup safety
- Ensures no resource leaks

Standard Dispose Pattern

```
using System;

class ResourceHandler : IDisposable
{
    private bool disposed = false;

    public ResourceHandler()
    {
        Console.WriteLine("Resource acquired.");
    }

    ~ResourceHandler()
    {
        Dispose(false);
    }

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

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                Console.WriteLine("Releasing managed resources.");
            }
```

```
            Console.WriteLine("Releasing unmanaged resources.");
            disposed = true;
        }
    }
}
```

Why GC.SuppressFinalize() Is Important

- Prevents finalizer from running
- Improves performance
- Avoids duplicate cleanup

---

6. Real-World Code-Based Application

Database Connection Example

```
class DbManager : IDisposable
{
    private SqlConnection connection;

    public DbManager(string conn)
    {
        connection = new SqlConnection(conn);
        connection.Open();
    }

    public void Dispose()
    {
        connection?.Close();
        connection?.Dispose();
    }
}
```

✓ Connection closed immediately

✓ Prevents connection leaks

✓ Scales well in web apps

---

7. Best Practices for Finalize and Dispose

✓ Prefer `Dispose()` over `Finalize()`

✓ Implement finalizer only when necessary

✓ Always call `GC.SuppressFinalize()` in Dispose

✓ Use `using` blocks

✓ Avoid empty finalizers

✓ Never rely solely on Finalize

---

8. Comparison with C++ Destructors

Feature	C# Finalize	C++ Destructor
Execution	GC-controlled	Deterministic
Timing	Non-deterministic	Predictable
Resource cleanup	Limited	Complete
Performance	Slower	Faster

1-hr Task

PROBLEM STATEMENT

Enterprise Log Processing System – Memory Management & Garbage Collection in C#

Objective

Design and implement an **Enterprise Log Processing System** in C# that demonstrates **Garbage Collection**, **Object Lifetime**, **Weak References**, and the **Dispose–Finalize pattern** used in real-world .NET applications such as banking systems, cloud services, and logging frameworks.

SYSTEM REQUIREMENTS OVERVIEW

The system must:

- Create and manage short-lived, medium-lived, and long-lived objects
- Demonstrate **Generational Garbage Collection**
- Handle unmanaged resources safely
- Use `WeakReference` to build a GC-friendly cache
- Show deterministic and non-deterministic cleanup

TASK 1 – Create a Short-Lived Log Entry Object (Gen 0)

Namespace

EnterpriseLogSystem

Class Name

LogEntry

Data Members

Name	Type	Access Modifier	Purpose
`Message`	`string`	`public`	Stores log message
`CreatedAt`	`DateTime`	`public`	Stores creation timestamp

Constructor

LogEntry(string message)

Input Parameters

- `message` – text of the log entry

What It Does

- Initializes log message
- Stores current date and time
- Creates a **short-lived object (Gen 0)**

Student Task

- Create a `LogEntry` class
- Allocate thousands of objects in a loop

Expected Outcome

- Objects are created in **Generation 0**
- Eligible for garbage collection quickly

Test Case

Input

LogEntry entry = new LogEntry("System started");

Output

Message = System started
CreatedAt = Current DateTime

TASK 2 – Implement a Medium-Lived Cache (Gen 1)

Class Name

LogCache

Data Members

Name	Type	Access Modifier
`_cache`	`List<LogEntry>`	`private`

Methods

1. Add Log Entry

void Add(LogEntry entry)

Purpose

- Stores log entries in memory
- Promotes objects to **Gen 1**

---

2. Clear Cache
void Clear()

Purpose

- Removes references
- Makes objects eligible for GC

Student Task

- Store multiple `LogEntry` objects
- Clear cache to release references

Expected Outcome

- Objects become **unreachable**
- Garbage Collector reclaims memory

Test Case

Input

```
cache.Add(new LogEntry("Login"));
cache.Clear();
```

Output

Log entries eligible for garbage collection

TASK 3 – Handle Unmanaged Resources (Finalize + Dispose)

Class Name

FileLogger

Interfaces Implemented

IDisposable

Data Members

Name	Type	Access Modifier
`_writer`	`StreamWriter`	`private`
`disposed`	`bool`	`private`

Constructor

FileLogger(string filePath)

Purpose

- Opens a file
- Acquires unmanaged file handle

Methods

1. Write Log
void WriteLog(string message)

Writes message to log file.

---

2. Dispose (Deterministic Cleanup)
public void Dispose()

- Releases managed resources
- Suppresses finalizer

---

3. Finalizer (Safety Net)
~FileLogger()

- Executes if `Dispose()` not called

---

4. Protected Dispose Logic
protected virtual void Dispose(bool disposing)

Handles:

- Managed resources
- Unmanaged resources

Student Task

- Implement full **Dispose pattern**
- Use `using` block

Expected Outcome

- File handle is always released
- No resource leaks

Test Case

Input

```
using (FileLogger logger = new FileLogger("app.log"))
{
```

```
    logger.WriteLog("Started");
}
```

Output

File resource acquired.
Managed resources released.
Unmanaged resources released.

TASK 4 – Demonstrate WeakReference (GC-Friendly Cache)

Class Name

LogProcessor

Data Members

Name	Type	Access Modifier
`_cache`	`LogCache`	`private`
`_weakCacheRef`	`WeakReference`	`private`

Method

void ProcessLogs()

What This Method Does

1. Creates thousands of `LogEntry` objects
2. Stores them in cache

3. Clears cache and removes strong references
4. Uses `WeakReference` to check object survival
5. Forces garbage collection

Student Task

- Observe memory before and after GC
- Check weak reference status

Expected Outcome

- Cache is collected
- Weak reference becomes invalid

Test Case

Input

processor.ProcessLogs();

Output

Initial Memory: XXXX bytes
Memory After Log Creation: YYYY bytes
Memory After GC: ZZZZ bytes
Is Cache Alive (WeakReference): False

TASK 5 – Demonstrate Generational GC

Code Snippet

```
object obj = new object();
Console.WriteLine(GC.GetGeneration(obj));
```

Student Task

- Create objects
- Check their generation

Expected Outcome

Object	Generation
New object	Gen 0
Long-lived cache	Gen 1
Application-wide objects	Gen 2

TASK 6 – Application Entry Point

Class Name

Program

Method

static void Main()

Responsibilities

- Use `FileLogger`
- Trigger log processing
- Display GC generations
- End program cleanly

Expected Final Output

```
=== Enterprise Log Processing System ===
File resource acquired.
Managed resources released.
Unmanaged resources released.
FileLogger disposed.
Initial Memory: ...
Memory After GC: ...
Generation of new object: 0
Application execution completed.
```

FINAL LEARNING OUTCOMES

After completing this problem, students will understand:

- ✓ Garbage Collector internals
- ✓ Generational GC
- ✓ WeakReference usage
- ✓ Dispose vs Finalize
- ✓ Real enterprise memory-safe design

How This Code Covers ALL TOPICS

---

1. Role of Garbage Collector

✓ Automatically deallocates unused objects

✓ Reclaims memory after `_cache.Clear()`

✓ Manages heap memory

✓ Optimizes performance

✓ Prevents memory leaks

📌 Demonstrated via:

```
GC.GetTotalMemory()
GC.Collect()
```

---

2. Garbage Collection Algorithm

Mark

- GC finds reachable objects (`GC Roots`)

Sweep

- `LogEntry` objects removed after cache cleared

Compact

- Heap memory reorganized

📌 Generational behavior:

- `LogEntry` → Gen 0
- `LogCache` → Gen 1
- Long-running objects → Gen 2

---

3. WeakReference (GC-Friendly Cache)

```
WeakReference _weakCacheRef;
```

✓ Allows GC to reclaim cache

✓ Used in real systems for caching

---

4. Finalize vs Dispose

Dispose

✓ Deterministic

✓ Fast

✓ Preferred

Finalize

✓ Backup safety

✓ GC-controlled

✓ Non-deterministic

📌 Implemented using **Dispose Pattern**:

```
Dispose(bool disposing)
GC.SuppressFinalize(this)
```

---

5. Real-Time Business Justification

Used In

- Banking systems
- Payment gateways
- Logging frameworks
- Cloud services
- Microservices

Why This Design Works

✓ No memory leaks

✓ No file-handle leaks

✓ Scalable

✓ Production-ready

```
public class LogProcessor
{
    private LogCache _cache;
    private WeakReference _weakCacheRef;

    public void ProcessLogs()
    {
        Console.WriteLine("\n--- Log Processing Started ---");

        long memoryBefore = GC.GetTotalMemory(false);
        Console.WriteLine($"Initial Memory: {memoryBefore} bytes");

        _cache = new LogCache();
        _weakCacheRef = new WeakReference(_cache);

        for (int i = 0; i < 10000; i++)
        {
```

```
            _cache.Add(new LogEntry("Log entry " + i));
        }

        long memoryAfterCreation = GC.GetTotalMemory(false);
        Console.WriteLine($"Memory After Log Creation: {memoryAfterCreation} bytes");

        _cache.Clear();
        _cache = null;

        GC.Collect();
        GC.WaitForPendingFinalizers();
        GC.Collect();

        long memoryAfterGC = GC.GetTotalMemory(true);
        Console.WriteLine($"Memory After GC: {memoryAfterGC} bytes");

        Console.WriteLine("Is Cache Alive (WeakReference): " +
            _weakCacheRef.IsAlive);
    }
}

using System;
using System.Collections.Generic;
using System.IO;

public class LogEntry
{
    public string Message { get; set; }
    public DateTime CreatedAt { get; set; }

    public LogEntry(string message)
    {
        Message = message;
        CreatedAt = DateTime.Now;
    }
}

public class LogCache
{
    private List<LogEntry> _cache = new List<LogEntry>();

    public void Add(LogEntry entry)
    {
        _cache.Add(entry);
```

```
    }

    public void Clear()
    {
        _cache.Clear();
    }
}

public class FileLogger : IDisposable
{
    private StreamWriter _writer;
    private bool disposed = false;

    public FileLogger(string filePath)
    {
        _writer = new StreamWriter(filePath, true);
        Console.WriteLine("File resource acquired.");
    }

    public void WriteLog(string message)
    {
        _writer.WriteLine($"{DateTime.Now}: {message}");
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                _writer?.Dispose();
                Console.WriteLine("Managed resources released.");
            }

            Console.WriteLine("Unmanaged resources released.");
            disposed = true;
        }
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
        Console.WriteLine("FileLogger disposed.");
```

```
    }

    ~FileLogger()
    {
        Dispose(false);
    }
}

public class Program
{
    static void Main()
    {
        Console.WriteLine("=== Enterprise Log Processing System ===\n");

        using (FileLogger logger = new FileLogger("app.log"))
        {
            logger.WriteLog("Application started");
        }

        LogProcessor processor = new LogProcessor();
        processor.ProcessLogs();

        object obj = new object();
        Console.WriteLine("\nGeneration of new object: " +
            GC.GetGeneration(obj));

        Console.WriteLine("\nApplication execution completed.");
    }
}
```