Understanding and Preventing Memory Leaks in C#

Vildana Šuta · Follow
4 min read · Jun 11, 2024

Listen Share More

Memory leaks in C# can significantly degrade system performance, leading to application crashes and system instability. This article explores the causes of memory leaks, how to identify them, and best practices for preventing them, ensuring your applications run smoothly.

As the name suggests, a memory leak is a situation that occurs when a program or an application uses the system's primary memory over a long period.

Memory leaks can significantly impact system performance and are often very challenging to detect. There are various tools such as memory profilers and resource monitors that can be used to detect abnormal memory retention and developers often use these tools. But, there are also some general signs that there is a memory leak in your code that should be fixed to improve overall application performance.

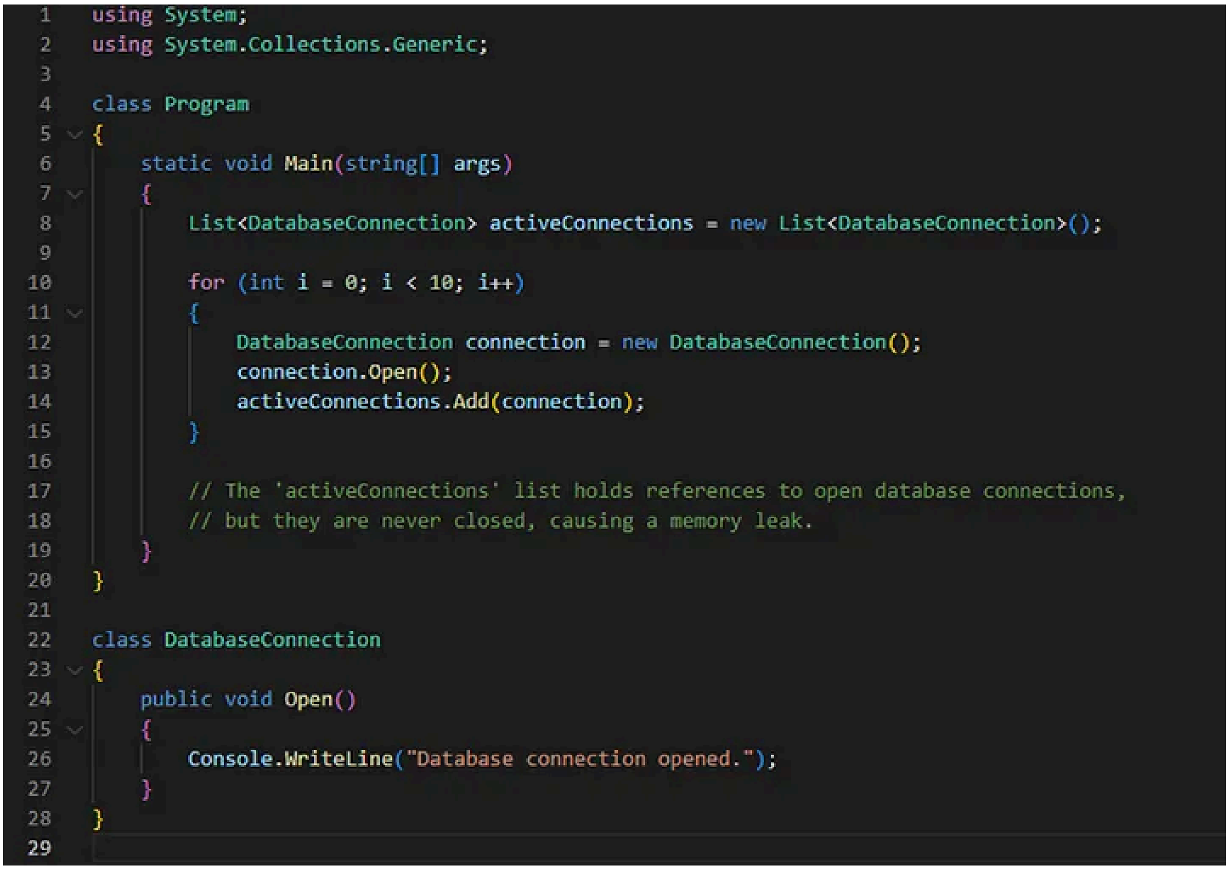
Let’s talk about signs that suggest there is a memory leak in your code first. The most common ones are:

- Gradual decrease in system performance over time
- Application crashes or freezes
- The application eventually throws “OutOfMemoryException” errors, particularly under sustained usage or heavy loads
- The overall system may become unstable, with other applications also slowing down or failing due to reduced available memory
- The garbage collector runs more frequently as it attempts to free up memory, often leading to noticeable pauses or delays in the application

We certainly don’t want any of those scenarios to happen. But why and how do they happen if we know there is a garbage collector that’s responsible for reclaiming the memory that is no longer needed by an application’s objects or processes?

A garbage collector doesn’t prevent *all* memory leaks because memory leaks typically occur when there are references to objects that are **unintentionally kept alive**, preventing the garbage collector from collecting them.

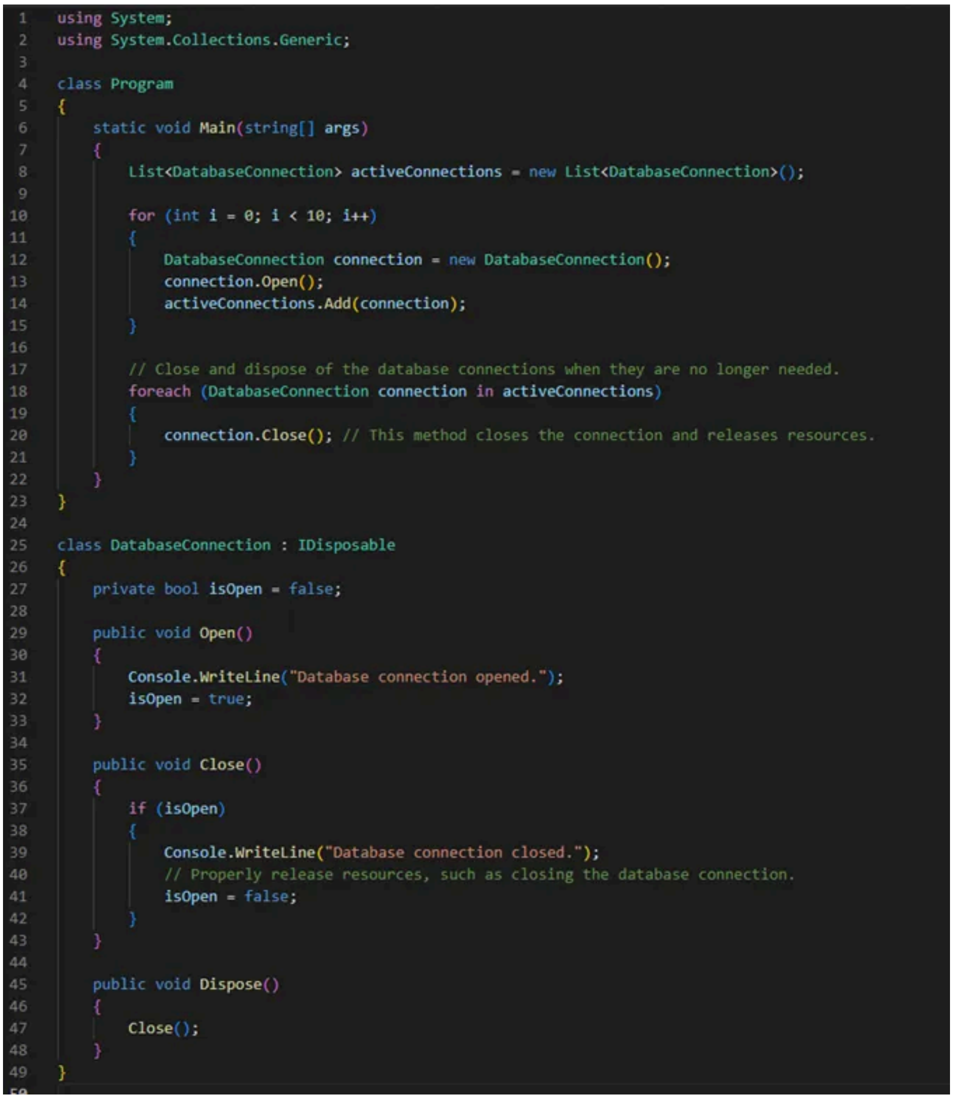
Here’s why the garbage collector can’t prevent all memory leaks:
➡️ **Rooted References**: If an object is referenced by a variable that remains in scope and is reachable from the root of the object graph, the garbage collector won’t collect that object even if it’s no longer needed. This can happen due to design flaws or long-lived references.



```
using System;
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        List<DatabaseConnection> activeConnections = new List<DatabaseConnection>();

        for (int i = 0; i < 10; i++)
        {
            DatabaseConnection connection = new DatabaseConnection();
            connection.Open();
            activeConnections.Add(connection);
        }

        // The 'activeConnections' list holds references to open database connections,
        // but they are never closed, causing a memory leak.
    }
}

class DatabaseConnection
{
    public void Open()
    {
        Console.WriteLine("Database connection opened.");
    }
}

```

Rooted References Memory Leak Example

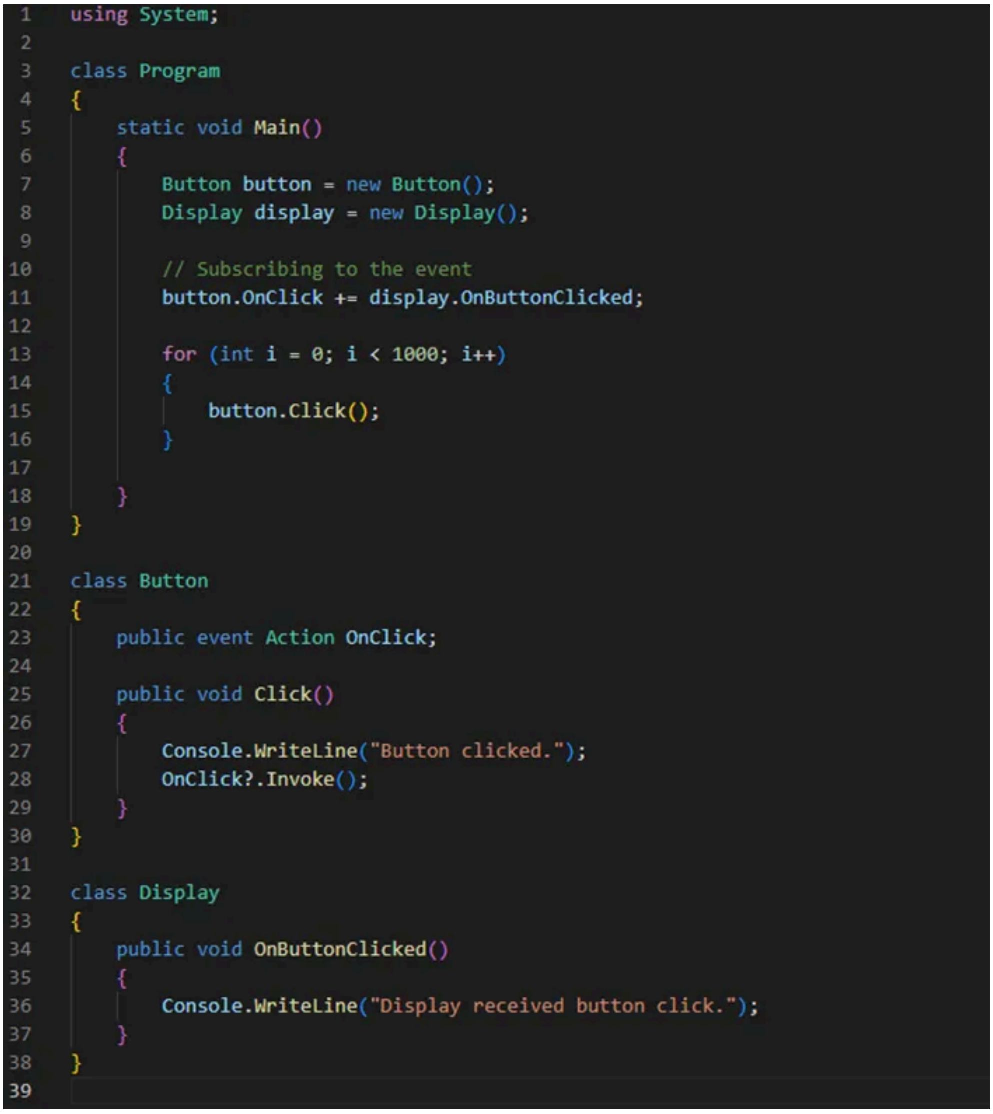
Instead, do this:



```
using System;
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        List<DatabaseConnection> activeConnections = new List<DatabaseConnection>();

        for (int i = 0; i < 10; i++)
        {
            DatabaseConnection connection = new DatabaseConnection();
            connection.Open();
            activeConnections.Add(connection);
        }

        // Close and dispose of the database connections when they are no longer needed.
        foreach (DatabaseConnection connection in activeConnections)
        {
            connection.Close(); // This method closes the connection and releases resources.
        }
    }
}

class DatabaseConnection : IDisposable
{
    private bool isOpen = false;

    public void Open()
    {
        Console.WriteLine("Database connection opened.");
        isOpen = true;
    }

    public void Close()
    {
        if (isOpen)
        {
            Console.WriteLine("Database connection closed.");
            // Properly release resources, such as closing the database connection.
            isOpen = false;
        }
    }

    public void Dispose()
    {
        Close();
    }
}
```

Rooted References Memory Leak Solution

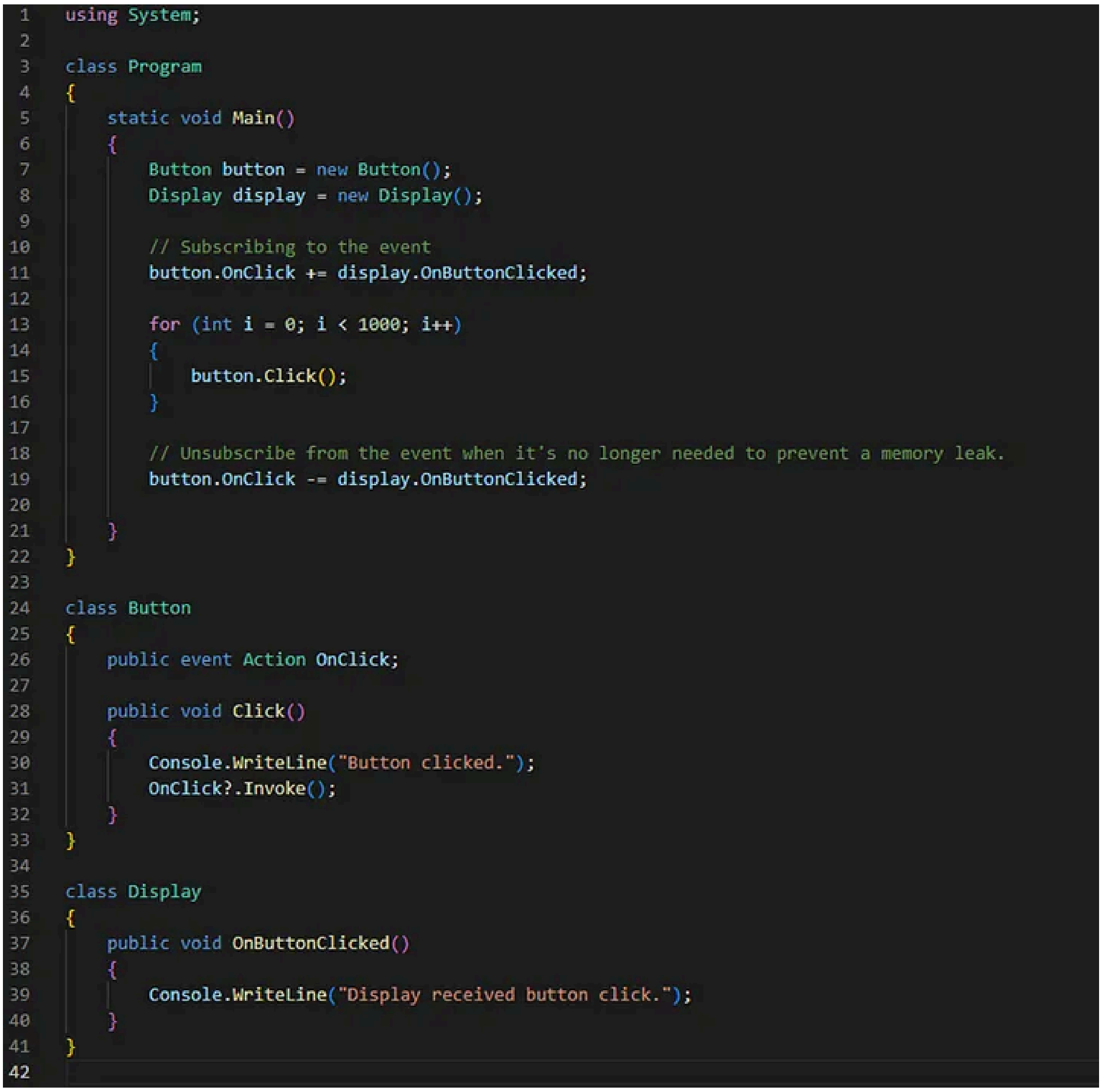
➡️ **Event Handlers:** Event handlers can create strong references, preventing objects from being collected. If you forget to unsubscribe from events when you’re done with them, objects can stay in memory.



```
using System;

class Program
{
    static void Main()
    {
        Button button = new Button();
        Display display = new Display();

        // Subscribing to the event
        button.OnClick += display.OnButtonClicked;

        for (int i = 0; i < 1000; i++)
        {
            button.Click();
        }

    }
}

class Button
{
    public event Action OnClick;

    public void Click()
    {
        Console.WriteLine("Button clicked.");
        OnClick?.Invoke();
    }
}

class Display
{
    public void OnButtonClicked()
    {
        Console.WriteLine("Display received button click.");
    }
}

```

Event Handlers Memory Leak Example

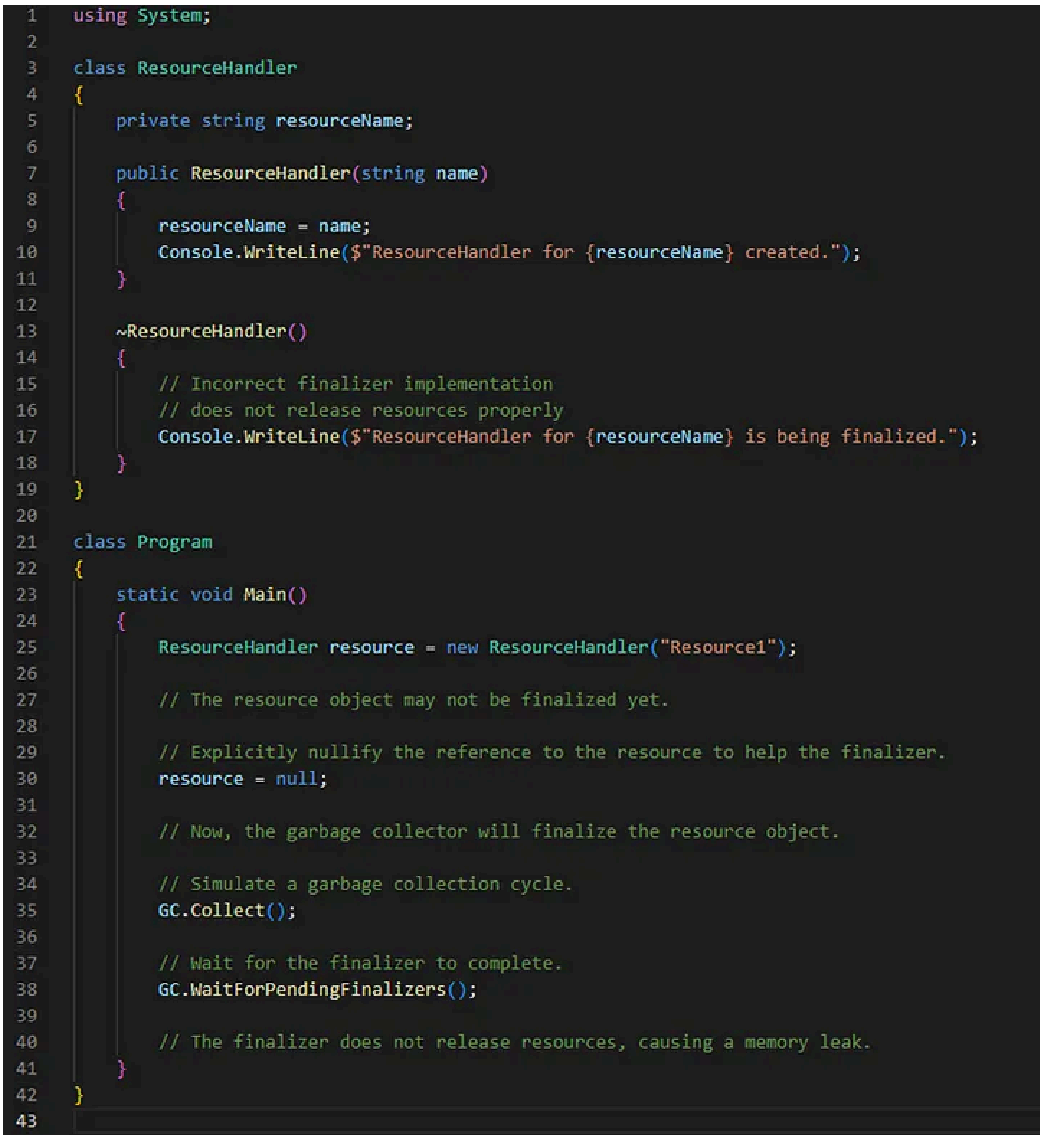
Instead, do this:



```
using System;

class Program
{
    static void Main()
    {
        Button button = new Button();
        Display display = new Display();

        // Subscribing to the event
        button.OnClick += display.OnButtonClicked;

        for (int i = 0; i < 1000; i++)
        {
            button.Click();
        }

        // Unsubscribe from the event when it's no longer needed to prevent a memory leak.
        button.OnClick -= display.OnButtonClicked;

    }
}

class Button
{
    public event Action OnClick;

    public void Click()
    {
        Console.WriteLine("Button clicked.");
        OnClick?.Invoke();
    }
}

class Display
{
    public void OnButtonClicked()
    {
        Console.WriteLine("Display received button click.");
    }
}
```

Event Handlers Memory Leak Solution

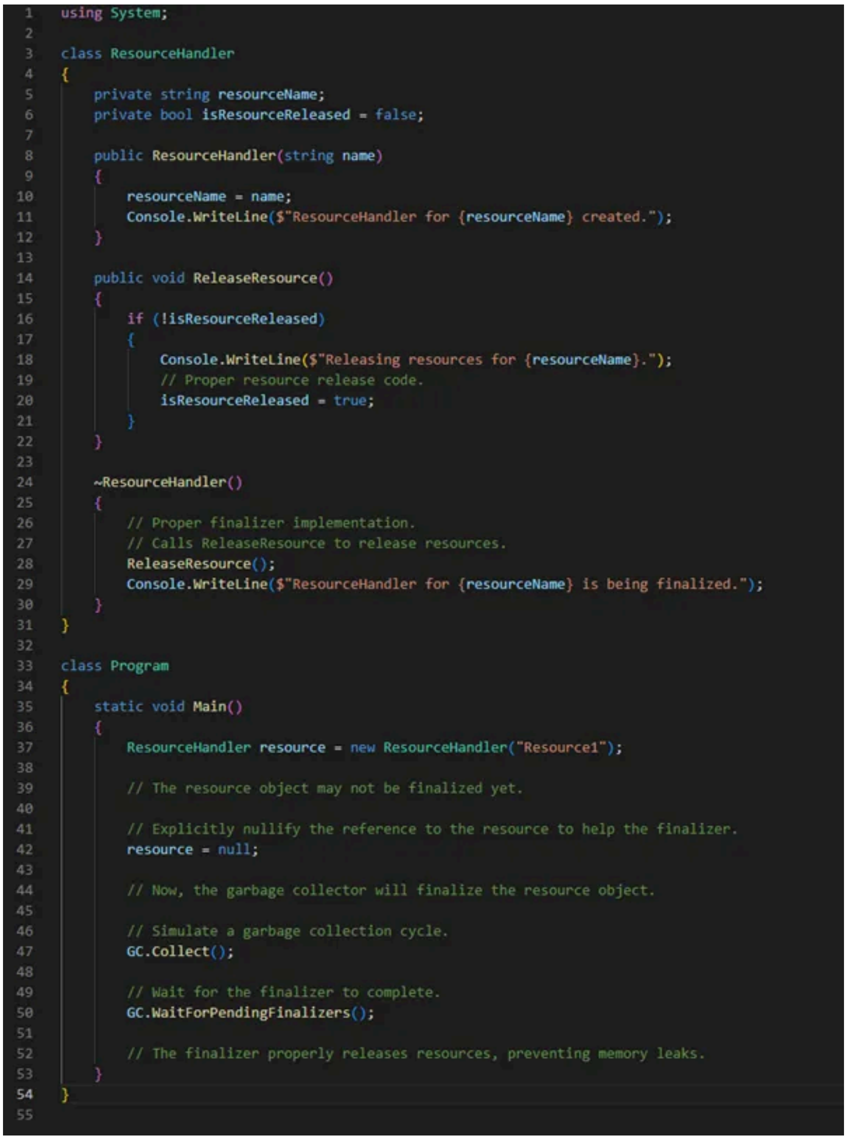
➡️ **Finalizers/Destructors**: Using finalizers can delay the reclamation of resources, potentially leading to memory leaks if finalizers are not implemented correctly.



```csharp
using System;

class ResourceHandler
{
    private string resourceName;

    public ResourceHandler(string name)
    {
        resourceName = name;
        Console.WriteLine($"ResourceHandler for {resourceName} created.");
    }

    ~ResourceHandler()
    {
        // Incorrect finalizer implementation
        // does not release resources properly
        Console.WriteLine($"ResourceHandler for {resourceName} is being finalized.");
    }
}

class Program
{
    static void Main()
    {
        ResourceHandler resource = new ResourceHandler("Resource1");

        // The resource object may not be finalized yet.

        // Explicitly nullify the reference to the resource to help the finalizer.
        resource = null;

        // Now, the garbage collector will finalize the resource object.

        // Simulate a garbage collection cycle.
        GC.Collect();

        // Wait for the finalizer to complete.
        GC.WaitForPendingFinalizers();

        // The finalizer does not release resources, causing a memory leak.
    }
}

```

Destructors Memory Leak Example

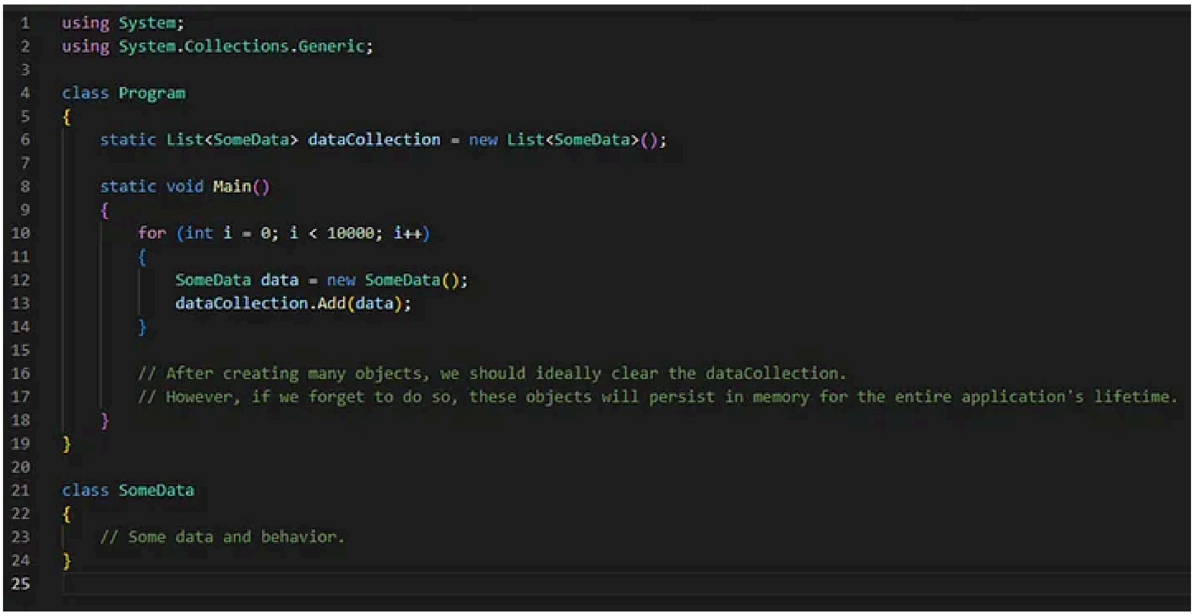
Instead, do this:



```csharp
using System;

class ResourceHandler
{
    private string resourceName;
    private bool isResourceReleased = false;

    public ResourceHandler(string name)
    {
        resourceName = name;
        Console.WriteLine($"ResourceHandler for {resourceName} created.");
    }

    public void ReleaseResource()
    {
        if (!isResourceReleased)
        {
            Console.WriteLine($"Releasing resources for {resourceName}.");
            // Proper resource release code.
            isResourceReleased = true;
        }
    }

    ~ResourceHandler()
    {
        // Proper finalizer implementation.
        // Calls ReleaseResource to release resources.
        ReleaseResource();
        Console.WriteLine($"ResourceHandler for {resourceName} is being finalized.");
    }
}

class Program
{
    static void Main()
    {
        ResourceHandler resource = new ResourceHandler("Resource1");

        // The resource object may not be finalized yet.

        // Explicitly nullify the reference to the resource to help the finalizer.
        resource = null;

        // Now, the garbage collector will finalize the resource object.

        // Simulate a garbage collection cycle.
        GC.Collect();

        // Wait for the finalizer to complete.
        GC.WaitForPendingFinalizers();

        // The finalizer properly releases resources, preventing memory leaks.
    }
}
```

Destructors Memory Leak Solution

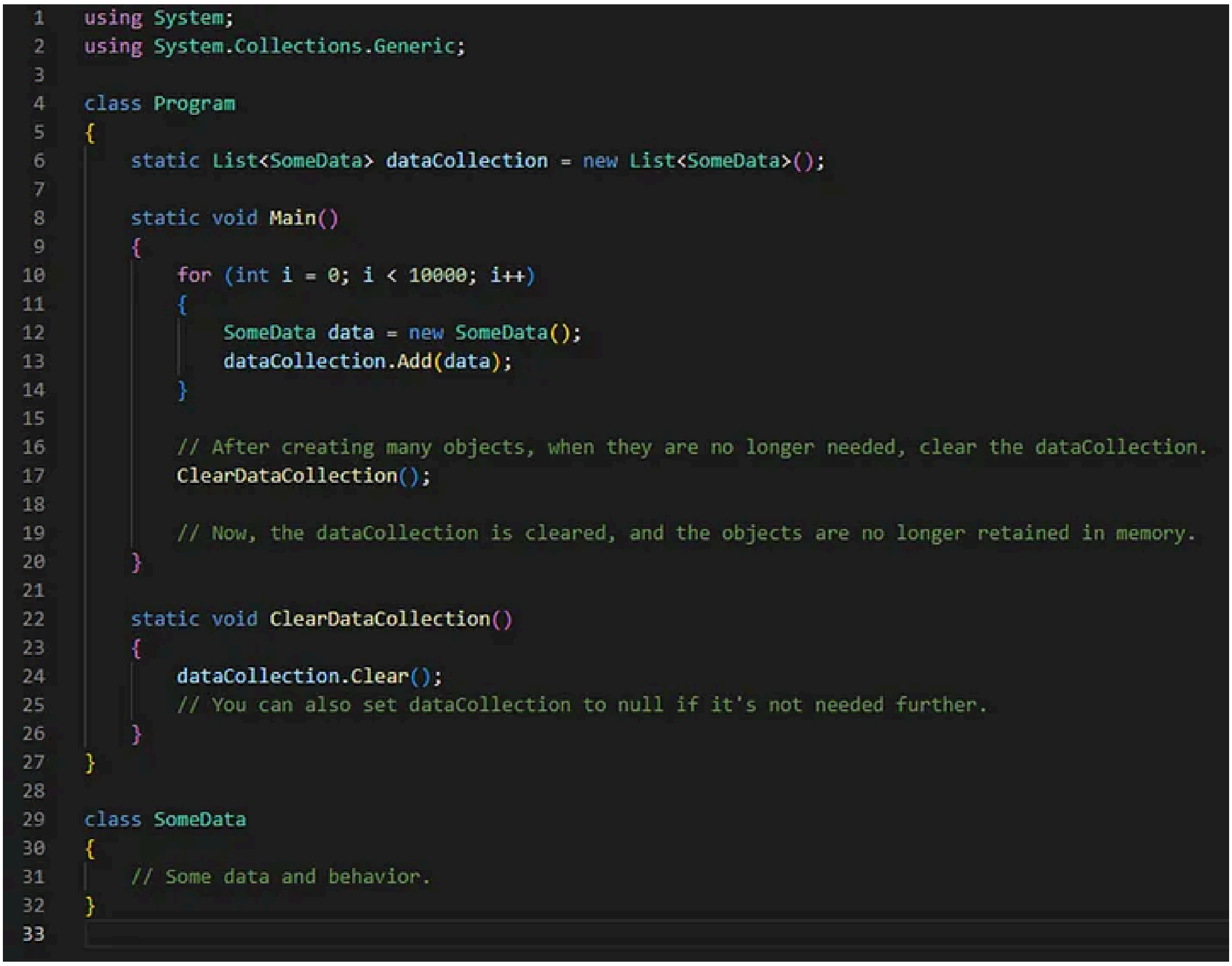
➡️ **Static Variables:** Objects referenced by static variables can persist for the entire application's lifetime, and if not managed carefully, can lead to memory leaks.



```csharp
using System;
using System.Collections.Generic;

class Program
{
    static List<SomeData> dataCollection = new List<SomeData>();

    static void Main()
    {
        for (int i = 0; i < 10000; i++)
        {
            SomeData data = new SomeData();
            dataCollection.Add(data);
        }

        // After creating many objects, we should ideally clear the dataCollection.
        // However, if we forget to do so, these objects will persist in memory for the entire application's lifetime.
    }
}

class SomeData
{
    // Some data and behavior.
}
```

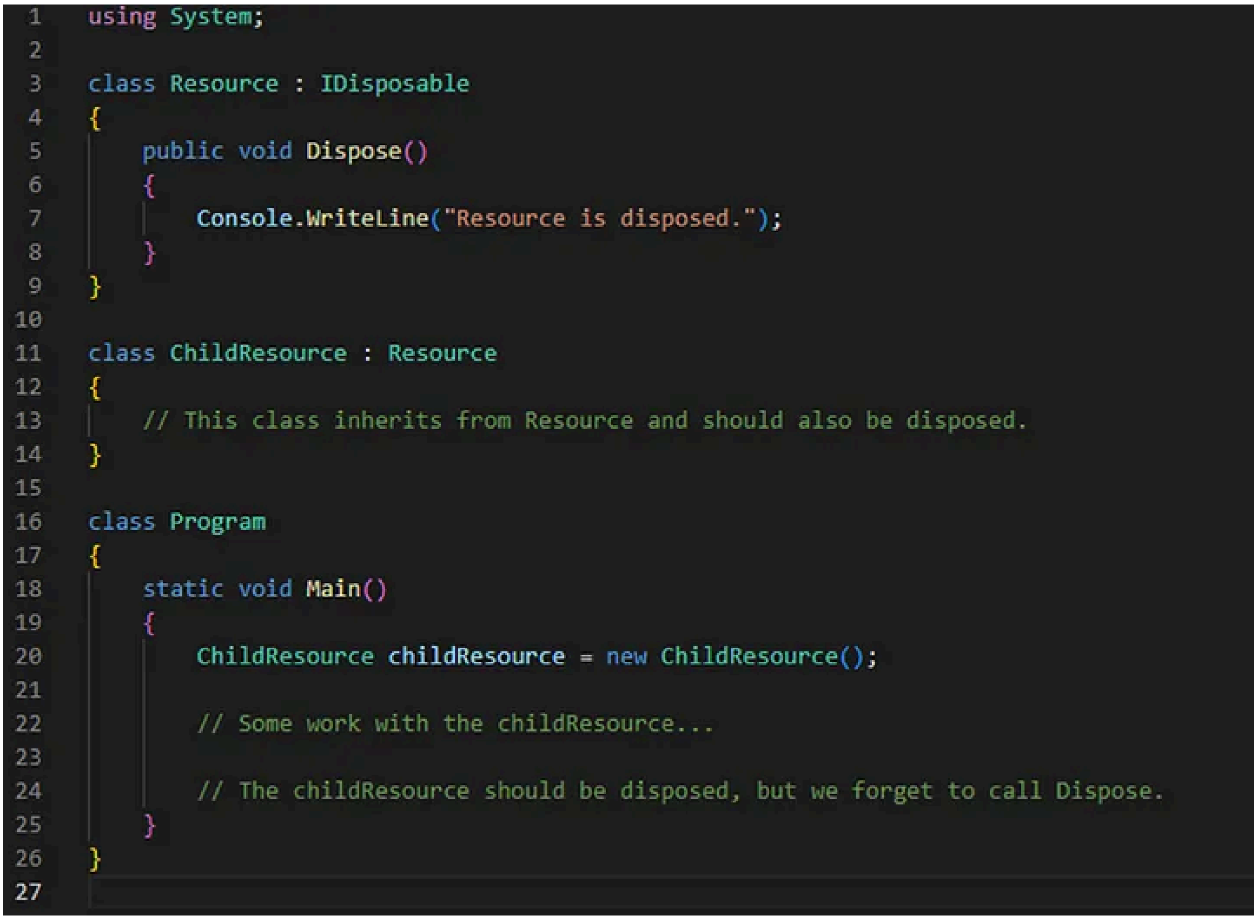
Static Variables Memory Leak Example

Instead, do this:



```csharp
using System;
using System.Collections.Generic;

class Program
{
    static List<SomeData> dataCollection = new List<SomeData>();

    static void Main()
    {
        for (int i = 0; i < 10000; i++)
        {
            SomeData data = new SomeData();
            dataCollection.Add(data);
        }

        // After creating many objects, when they are no longer needed, clear the dataCollection.
        ClearDataCollection();

        // Now, the dataCollection is cleared, and the objects are no longer retained in memory.
    }

    static void ClearDataCollection()
    {
        dataCollection.Clear();
        // You can also set dataCollection to null if it's not needed further.
    }
}

class SomeData
{
    // Some data and behavior.
}

```

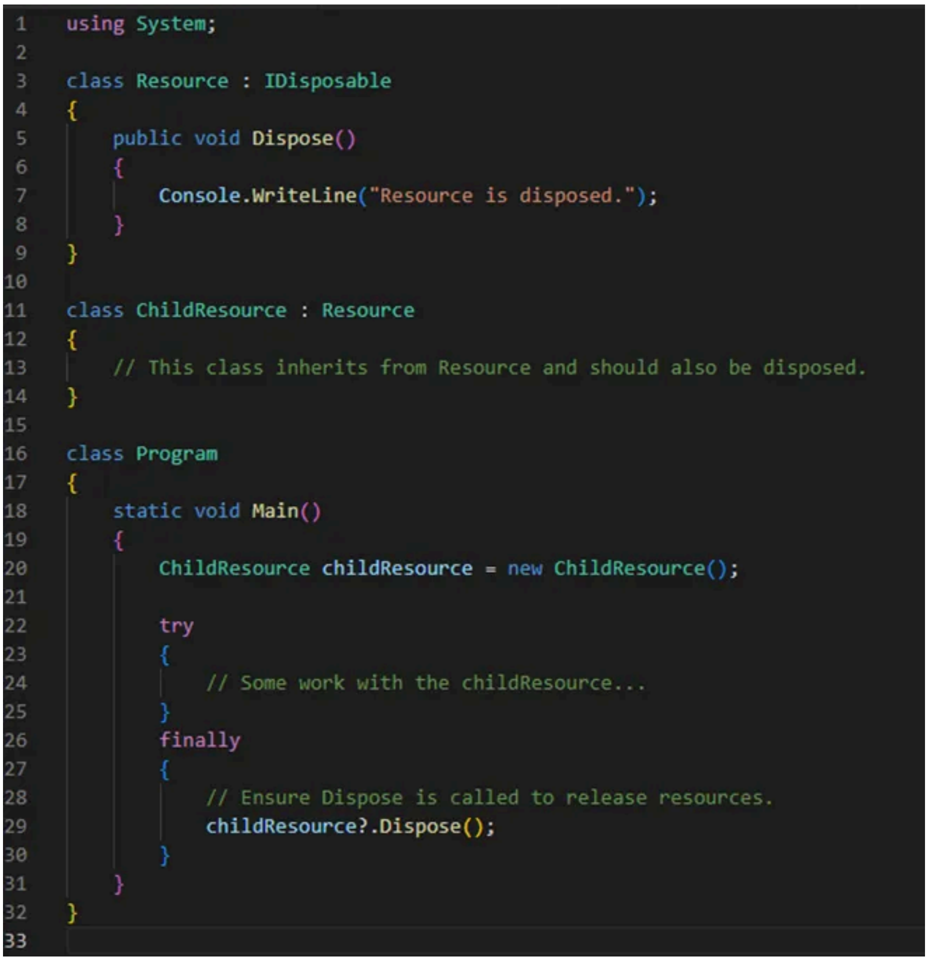
leaks.



```
using System;

class Resource : IDisposable
{
    public void Dispose()
    {
        Console.WriteLine("Resource is disposed.");
    }
}

class ChildResource : Resource
{
    // This class inherits from Resource and should also be disposed.
}

class Program
{
    static void Main()
    {
        ChildResource childResource = new ChildResource();

        // Some work with the childResource...

        // The childResource should be disposed, but we forget to call Dispose.
    }
}
```

Objects that implement IDisposable Memory Leak Example (using Dispose 1)

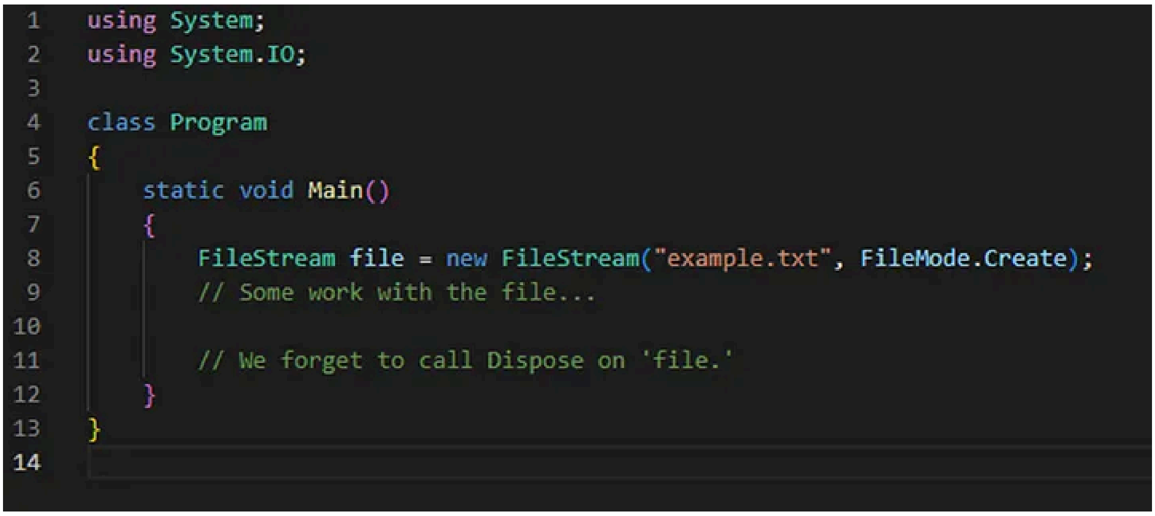
Instead, do this:



```
using System;

class Resource : IDisposable
{
    public void Dispose()
    {
        Console.WriteLine("Resource is disposed.");
    }
}

class ChildResource : Resource
{
    // This class inherits from Resource and should also be disposed.
}

class Program
{
    static void Main()
    {
        ChildResource childResource = new ChildResource();

        try
        {
            // Some work with the childResource...
        }
        finally
        {
            // Ensure Dispose is called to release resources.
            childResource?.Dispose();
        }
    }
}
```

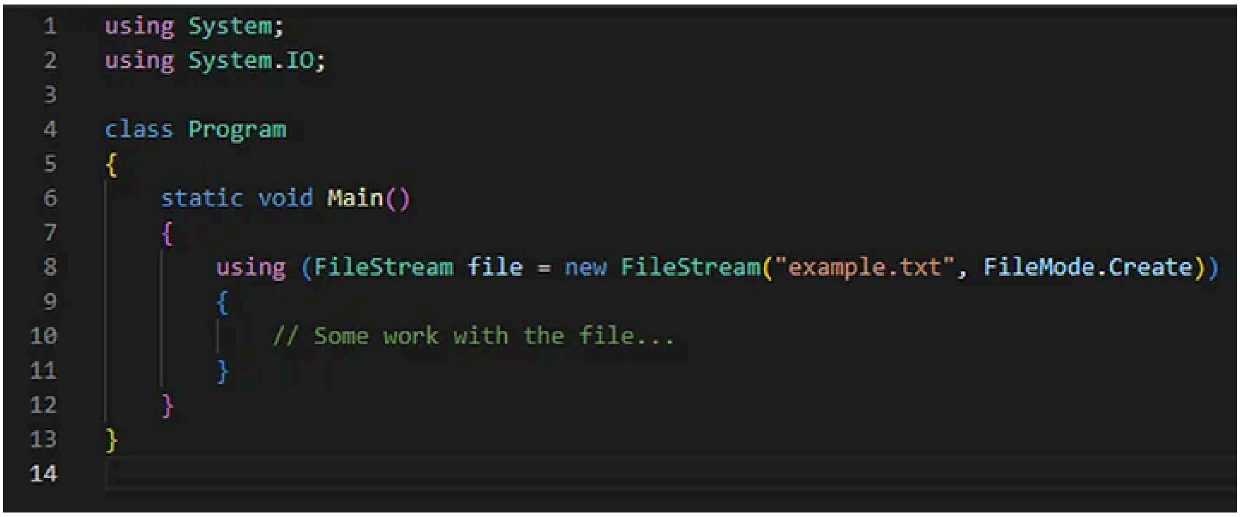
Objects that implement IDisposable Memory Leak Solution (using Dispose 1)

Another example with *using*:



```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        FileStream file = new FileStream("example.txt", FileMode.Create);
        // Some work with the file...

        // We forget to call Dispose on 'file.'
    }
}

```

Objects that implement IDisposable Memory Leak Example (with using 2️⃣)



```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        using (FileStream file = new FileStream("example.txt", FileMode.Create))
        {
            // Some work with the file...
        }
    }
}

```

Objects that implement IDisposable Memory Leak Solution (with using 2️⃣)

To conclude, some of the most efficient practices to follow to avoid memory leaks are:
1️⃣ Use IDisposable and the „using“ statement
2️⃣ Nullify References
3️⃣ Dispose of Event Handlers
4️⃣ Avoid Circular References
5️⃣ Profile and Analyze

Thank you for reading! Hope it was helpful!

Memory Leak · Programming · C Sharp Programming · Dotnet · Dotnet Core

Follow

Written by Vildana Šuta

97 Followers · 43 Following

👩‍💻 Hi, I'm Vildana! Software engineer working with .NET (C#), exploring full-stack with Angular, Flutter, React, and Vue. Coding adventures await! 🚀