1. **Static Class vs Single Ton.**

**Static Classes vs Singleton Classes**

Both **static classes** and **singleton classes** are used to enforce a single instance in some way, but they serve different purposes and have distinct behaviors.

**1. Static Class**

A **static class** is a class that cannot be instantiated and contains only static members. It is typically used for utility/helper functions that do not require object state.

**Key Characteristics of Static Classes:**

* **Cannot be instantiated**: You cannot create an instance of a static class.
* **Only contains static members**: All methods, properties, and fields must be static.
* **Memory Usage**: Exists for the entire application lifecycle and is stored in memory until the application shuts down.
* **Thread Safety**: Since static classes don't hold instance-specific data, they are inherently thread-safe (unless they modify shared static variables).
* **Performance**: Slightly better performance than singleton classes because there is no object creation.

**Example of a Static Class (C#)**

public static class MathHelper

{

public static double Square(double number)

{

return number \* number;

}

}

// Usage

double result = MathHelper.Square(5);

**2. Singleton Class**

A **singleton class** ensures that only **one instance** of the class exists and provides a global access point to that instance.

**Key Characteristics of Singleton Classes:**

* **Single instance**: Only one object of the class is created throughout the application lifecycle.
* **Encapsulation**: Unlike static classes, singleton classes can implement interfaces and support inheritance.
* **Memory Management**: The instance is created only when needed (lazy initialization) or at the start of the application (eager initialization).
* **Thread Safety**: Needs to be handled properly in multi-threaded environments to ensure only one instance is created.
* **Stateful**: Singleton classes can maintain state across different method calls.

**Example of a Singleton Class (C#)**

public class Singleton

{

private static Singleton instance;

private static readonly object lockObject = new object();

private Singleton() { } // Private constructor

public static Singleton GetInstance()

{

if (instance == null)

{

lock (lockObject)

{

if (instance == null)

{

instance = new Singleton();

}

}

}

return instance;

}

}

// Usage

Singleton obj1 = Singleton.GetInstance();

Singleton obj2 = Singleton.GetInstance();

Console.WriteLine(obj1 == obj2); // True (Same instance)

**Key Differences**

| **Feature** | **Static Class** | **Singleton Class** |
| --- | --- | --- |
| **Instance** | No instance allowed | Only one instance allowed |
| **State Management** | No state | Can maintain state |
| **Memory Usage** | Exists throughout application lifecycle | Created when needed (lazy loading possible) |
| **Inheritance** | Cannot implement interfaces or inherit from other classes | Can implement interfaces and inherit from classes |
| **Thread Safety** | Inherently thread-safe | Needs explicit thread-safety implementation |
| **Usage** | Utility/helper methods (e.g., Math, Logger) | Shared resources (e.g., database connections, config managers) |

**When to Use Which?**

* **Use a Static Class** when:
  + The class only contains utility or helper methods.
  + No instance-specific data is required.
  + You do not need to implement interfaces or extend other classes.
* **Use a Singleton Class** when:
  + You need to maintain state across multiple method calls.
  + You need to implement an interface or extend another class.
  + You want lazy loading or controlled access to resources.

**Conclusion**

* **Static classes** are useful for storing reusable methods without maintaining state.
* **Singleton classes** allow a single instance with state persistence and controlled instantiation.

Would you like a real-world scenario comparison (e.g., logging, database connections) to see their differences in action?

**Why Do Static Classes Not Support Inheritance in C#?**

1. **Static Classes Are Sealed by Default: -**

**A sealed class in C# cannot be inherited. When you declare a class as static, the compiler implicitly marks it as sealed to prevent inheritance.**

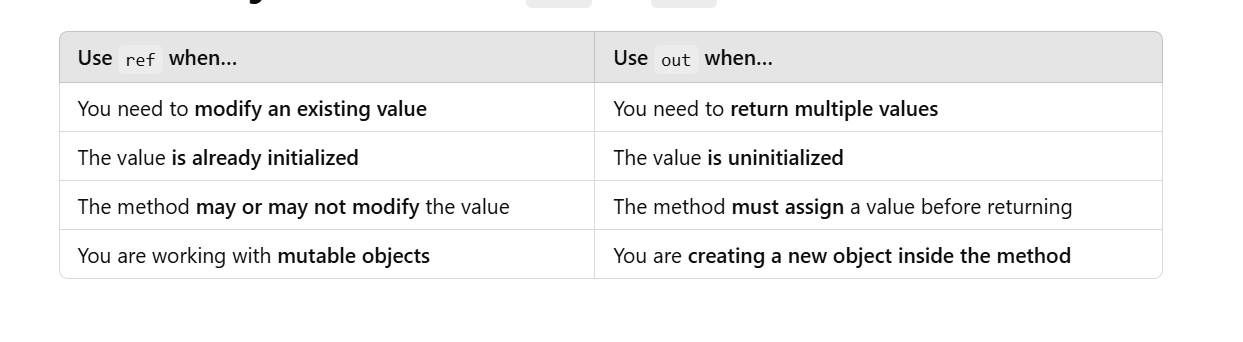
1. **No Instance, No Polymorphism**
   1. **Inheritance is mainly useful for creating object hierarchies and allowing polymorphic behaviour.**
   2. **Since static classes cannot be instantiated, there is no concept of polymorphism, making inheritance meaningless.**
2. **Static Methods Belong to the Type, Not an Instance: -**
   1. **Static members belong to the class itself, not an instance.**
   2. **Inheritance is used to override instance methods, which static classes do not support**
3. **Alternative: Use Extension Methods Instead: -**
   1. **Since inheritance is not possible for static classes, you can extend their functionality using extension**
4. **Prevents Misuse of OOP Principles**

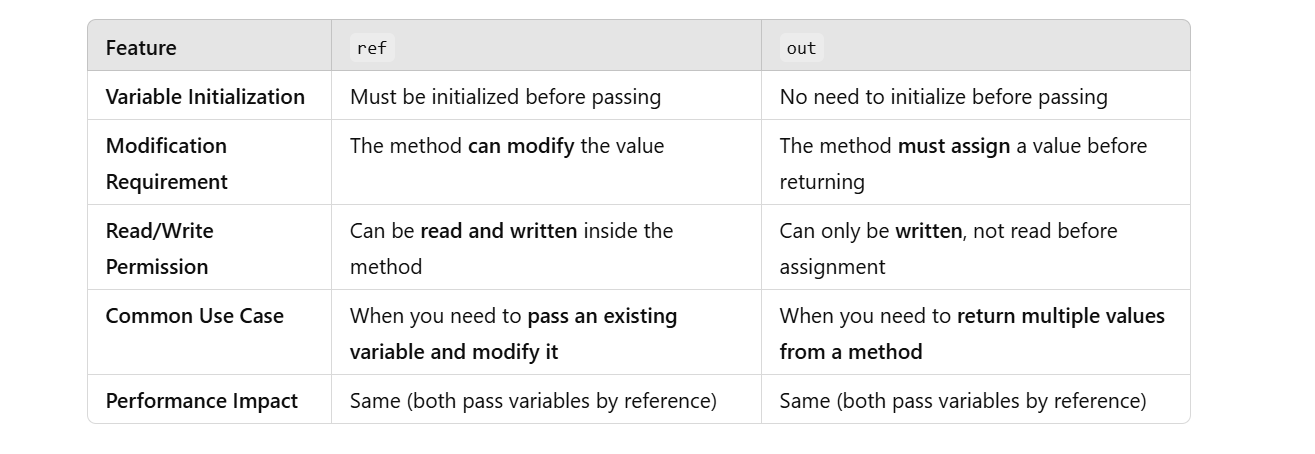
**Why Static Classes Cannot Implement Interfaces in C#?**

1. Read-Only vs Const.
2. **Key Differences Between readonly and const**

| **Feature** | **const** | **readonly** |
| --- | --- | --- |
| **Value Assignment** | Must be assigned at declaration | Can be assigned in a constructor or declaration |
| **Modification** | **Cannot change** after compilation | Can be **set at runtime** but not modified after initialization |
| **Supports Complex Types** | **No** (Only primitive types, strings) | **Yes** (Objects, arrays, lists) |
| **Memory Storage** | **Replaced at compile time in IL (metadata)** | **Stored in memory (heap/stack)** |
| **Scope** | Implicitly static (cannot be changed per instance) | Can be **instance-based or static** |
| **Runtime Value Assignment** | ❌ No (Compile-time only) | ✅ Yes (Can use dynamic values) |
| **Performance** | Faster (Replaced in IL at compile time) | Slightly slower (Resolved at runtime) |

1. Ref vs Out.





1. Throw vs Throw ex.

In C#, both throw and throw ex are used to raise exceptions, but they have **different effects on the stack trace**, which is crucial for debugging.

1. throw (Preserves Original Stack Trace)

✅ **Definition:**

* throw **rethrows the original exception** without modifying the stack trace.
* Best practice for **maintaining debugging information**.

1. throw ex (Resets Stack Trace)
2.  throw ex **creates a new exception**, causing the **original stack trace to be lost**.
3.  This makes debugging **more difficult**.