Bonus

4) Self-Study Report

Creating Objects (Reference Types)
creating objects of reference types has a direct impact on memory behavior. When the new keyword is used, memory for the object itself is allocated in the heap, while the reference to that object is stored in the stack.
This design allows:

- Multiple references to point to the same object
- Shared access to the same memory location

As a result, any modification made through one reference is reflected across all references. While this makes reference types powerful and flexible, it can also introduce unintended side effects if references are not handled carefully.

Memory Leaks in C#
Although C# is a managed language, memory leaks can still occur.
A memory leak happens when:

- An object is no longer needed
- But it remains in memory because it is still referenced

Common causes include:

- Long-living references such as static objects
- Event handlers that are not unsubscribed
- Improper handling of unmanaged resources

Over time, these issues can lead to increased memory usage and degraded application performance.

Garbage Collector (GC)
The Garbage Collector is a core component of the Common Language Runtime (CLR) and is responsible for automatic memory management.
Its responsibilities include:

- Tracking objects allocated in the heap
- Identifying objects that are no longer reachable
- Reclaiming memory used by those objects

The GC uses a generational model:

- New objects start in Generation 0
- Objects that survive collections are promoted to higher generations

This approach improves performance by focusing on collecting short-lived objects more frequently.

Customizing Garbage Collection Behavior
Developers do not have direct control over when the Garbage Collector runs. However, its efficiency can be influenced through proper coding practices.
Best practices include:

- Reducing unnecessary object allocations
- Releasing unmanaged resources as early as possible
- Implementing the IDisposable interface
- Using the using statement where applicable

These practices help ensure that resources such as file streams and database connections are released promptly, without waiting for garbage collection.

Unmanaged Resources
Unmanaged resources are system-level resources that are not controlled by the Garbage Collector.
Examples include:

- Database connections
- File streams
- Network sockets

Because the GC does not manage these resources, developers must explicitly release them. Failure to do so can result in resource leaks, potentially exhausting system limits such as open file handles or active connections.

Floating-Point Numbers and the FPU
Floating-point types such as float and double rely on the Floating Point Unit (FPU) and use scientific notation to represent numbers.
This allows them to:

- Represent a wide range of values efficiently
- Perform fast mathematical operations

However, due to their binary representation:

- Some decimal values cannot be represented exactly
- Small precision errors may occur
- These errors can accumulate over time

When and Why to Use decimal
The decimal data type prioritizes precision over performance.
Key characteristics:

- Uses base-10 representation
- Stores decimal values exactly as written
- Requires more memory and processing power

Because of this, decimal is ideal for:

- Financial calculations
- Pricing systems
- Accounting applications

In precision-sensitive domains, accuracy outweighs the performance cost.

---

float vs double vs decimal
Choosing the correct numeric type depends on application requirements:

- float / double
  - Faster
  - Suitable for scientific calculations, simulations, and graphics
  - May introduce rounding errors
- decimal
  - Slower
  - Higher precision
  - Best for financial and monetary calculations

Understanding these differences helps avoid both rounding errors and performance bottlenecks.

Checked and Unchecked Blocks
C# provides checked and unchecked contexts to control integer overflow behavior.

- checked
  - Throws a runtime exception on overflow
  - Useful during development and debugging
- unchecked
  - Allows overflow to occur silently
  - Value wraps around without errors

This feature gives developers explicit control over how overflow should be handled based on application needs.

Parsing Null Values
Parsing values from strings requires careful validation.
For example:

- Calling int.Parse(null) throws an exception
- Parsing methods expect a valid string representation

This highlights the importance of input validation, especially when working with user input or external data sources.

Parse vs Convert (Performance Considerations)
Parse and Convert differ in both behavior and performance.

- Parse
  - Faster
  - Requires a valid, non-null string
  - Throws exceptions on invalid input
- Convert
  - Slightly slower
  - Handles null values safely
  - More robust for uncertain input

The choice depends on whether performance or safety is the primary concern.

Stack vs Heap
Memory in C# is primarily divided between the stack and the heap.

- Stack
  - Used for value types and method execution contexts
  - Fast allocation
  - Automatically cleared when scope ends
- Heap
  - Used for reference types
  - Managed by the Garbage Collector
  - Supports longer-lived objects

Understanding this distinction is essential for writing efficient and predictable code.

Deallocation
Memory deallocation in C# depends on where the memory is allocated:

- Stack memory is released automatically when a method completes
- Heap memory is reclaimed by the Garbage Collector
- Unmanaged resources must be released manually by the developer

Failing to release unmanaged resources can lead to serious leaks even in managed environments.

Bitwise Operators
Bitwise operators allow direct manipulation of individual bits within a value.
They are commonly used in:

- Performance-critical code
- Flag-based logic
- Low-level system programming

While not frequently used in everyday applications, understanding bitwise operations provides deeper insight into data representation and low-level processing.

---

5) What Does “C# Is Managed Code” Mean?

Managed code means that:

- Code execution is controlled by the CLR
- Memory is managed automatically
- No manual memory allocation or deallocation

And the CLR provides:

- Garbage Collection
- Type safety
- Exception handling
- Security checks

---

6) Why Is Struct Considered Like Class Before?

struct is considered like class because both of them can have fields, properties, methods, and constructors, and they support encapsulation

But the struct is for value type and it’s stored in stack, cannot inherit and its lightweight. While on the other hand the class is for reference type and it’s stored in heap, supports inheritance and its heavier than struct.

So the structs are described as class in syntax, but fundamentally different in behavior and memory model.