PART01

- Question: What is the default value assigned to array elements in C#?

In C#, **when you create an array**, each element is automatically assigned a **default value** based on the type of the array:

Data Type	**Default Value**
int	0
float	0.0f
bool	false
char	'\0' (null char)
string	null
object	null
Custom Classes	null

🔍 **Example:**

```
int[] numbers = new int[3];

Console.WriteLine(numbers[0]); // Output: 0

string[] names = new string[2];

Console.WriteLine(names[1]); // Output: (nothing, because it's null)
```

So, **you don’t need to initialize all elements manually** unless you want custom values.

• Question: What is the difference between Array.Clone() and Array.Copy()?

Feature	Array.Clone()	Array.Copy()
Purpose	Creates a **shallow copy** of the array.	Copies elements from one array to another.
Return Type	Returns a **new object** (needs casting).	Returns void, works by passing destination array.
Copy Level	Shallow copy (only references copied).	Can be used for deep or shallow depending on type.
Usage	Simple duplication of an array.	More control over copying specific elements.

🧪 **Example:**

```
int[] original = { 1, 2, 3 };

// Clone (returns object, so needs cast)

int[] cloned = (int[])original.Clone();

// Copy (requires target array)

int[] copied = new int[3];

Array.Copy(original, copied, original.Length);
```

⚠️ **Note:**

Both methods **do not perform deep copies** of complex reference types (like arrays of objects). You must manually implement deep copy logic if needed.

• Question: What is the difference between GetLength() and Length for multi dimensional arrays?

Property/Method	GetLength(dimension)	Length
Purpose	Returns the **size of a specific dimension**.	Returns the **total number of elements**.
Usage	array.GetLength(0) → rows array.GetLength(1) → columns	Just array.Length
Type	Method	Property
Return Type	int	int

🧪 **Example:**

```
int[,] matrix = new int[3, 4];  // 3 rows, 4 columns

int rows = matrix.GetLength(0);     // 3

int cols = matrix.GetLength(1);     // 4

int total = matrix.Length;         // 12 (3 × 4)
```

📌 **Summary:**

- Use GetLength(d) to get the **size of a specific dimension**.
- Use Length to get the **total number of elements** across all dimensions.

• Question: What is the difference between Array.Copy() and Array.ConstrainedCopy()?

Feature	Array.Copy()	Array.ConstrainedCopy()
Exception Behavior	Can perform a **partial copy** if exception occurs	Guarantees **all-or-nothing** copy
Exception Handling	Doesn't rollback changes on error	Rolls back all changes if an exception occurs
Use Case	General-purpose copying	Situations requiring **strict safety & rollback**
Performance	Slightly faster (less overhead)	Slightly slower (adds extra checks)
Accessibility	public static	protected static – used mainly in base class libraries

---

🧪 **Example:**

```
int[] source = { 1, 2, 3 };

int[] target = new int[3];

// ✅ Array.Copy – may leave partial data if an exception occurs

Array.Copy(source, target, 3);

// ✅ Array.ConstrainedCopy – safer, rollback on failure (only works inside certain base code contexts)

// Array.ConstrainedCopy(source, 0, target, 0, 3);
```

📌 **Summary:**

- Use **Array.Copy()** for general copying needs.
- Use **Array.ConstrainedCopy()** when you need a **safe, atomic copy** (used more in system-level code or libraries, not common in user applications).

• Question: Why is foreach preferred for read-only operations on arrays?

foreach is preferred for **read-only operations** because:

Reason	Explanation
Simplicity	It provides a clean and concise syntax with **no need for index management**.
Safety	Prevents accidental **modification** of array elements (loop variable is **read-only**).
No Indexing Errors	Eliminates chances of IndexOutOfRangeException.
Clear Intent	Clearly expresses that you're **only reading**, not modifying.
Works with all collections	Unified way to iterate over arrays, lists, etc.

🧪 **Example:**

```
int[] numbers = { 1, 2, 3, 4 };

// ✅ Read-only iteration

foreach (int num in numbers)

{

   Console.WriteLine(num); // Safe read, no change to the array

}


// ❌ This will NOT compile because `num` is read-only inside foreach

// num = num + 1;
```

📌 **Summary:**

- Use foreach when you just need to **loop through** the array and **read values**.
- It's **cleaner**, **safer**, and **less error-prone** than for.

- Question: Why is input validation important when working with user inputs?

  Input validation is critical because it ensures that user input is:

Secure	Prevents **malicious inputs** like code injection or harmful data.
Error-Free	Avoids **runtime errors** such as FormatException, NullReferenceException, etc.
Accurate	Ensures the input meets **expected format, range, or type**.
Reliable	Helps the program behave **predictably and consistently**.
Maintainable	Makes code more **robust** and easier to debug or expand.

📌 **Example:**

```
Console.Write("Enter a positive number: ");

string input = Console.ReadLine();

if (int.TryParse(input, out int number) && number > 0)

{

   Console.WriteLine("Valid input: " + number);

}

else

{

   Console.WriteLine("Invalid input! Please enter a positive number.");

}
```

⚠️ **Without validation:**

- User types "abc" → app crashes.
- User types negative number → incorrect logic.
- User enters script → potential security risk.

✅ **Summary:** Input validation is essential for **security**, **stability**, and **user-friendly** software.
It protects your program **and** the user.

• Question: How can you format the output of a 2D array for better readability?

To **improve readability** when printing a 2D array, you can:

✅ 1. Use Nested Loops with Tabs or Alignment

Each **row** is printed on a **new line**, and **columns** are separated with tabs (\t) or spacing.

🔶 Example:

```
int[,] grades = {
   { 85, 90, 78 },
   { 88, 92, 80 },
   { 75, 85, 89 }
};


for (int i = 0; i < grades.GetLength(0); i++)  // Rows
{
   for (int j = 0; j < grades.GetLength(1); j++)  // Columns
   {
      Console.Write(grades[i, j] + "\t");
   }
   Console.WriteLine();  // New line for next row
}
```

✅ 2. Format Using String.Format or Interpolation

You can format numbers with fixed width:

```
Console.Write($"{grades[i, j],-5}");
```

This aligns columns neatly regardless of the number of digits.

✅ 3. Add Row/Column Headers (Optional)

Adding headers can make the output clearer:

Console.WriteLine("Stu1\tStu2\tStu3");

✅ Summary Table:

Technique	Benefit
\t or spacing	Aligns columns cleanly
Console.WriteLine()	Starts new row
string.Format or $""	Control spacing and alignment
Headers	Improves clarity for context

- Question: What is the time complexity of Array.Sort()?

The **time complexity** of Array.Sort() depends on the **data type** and **array size**, but generally:

✅ 1. For Primitives (e.g., int, double, etc.):

- **Average Time Complexity:** O(n log n)
- **Worst-Case Time Complexity:** O(n log n)

🔧 Internally:

- C# uses a **hybrid sorting algorithm** called **Introspective Sort (Introsort)** for primitive types.
- Introsort = QuickSort + HeapSort + InsertionSort

✅ **2. For Reference Types / Custom Objects:**

- Also typically **O(n log n)**, but may vary slightly depending on:
    - The IComparer or IComparable logic
    - Internal object layout

✅ **Summary Table:**

Type	Algorithm Used	Average Case	Worst Case
int[]	Introsort	O(n log n)	O(n log n)
string[]	Introsort	O(n log n)	O(n log n)
Custom types	Introsort	O(n log n)	O(n log n)

💡 **Note:**

- Array.Sort() is **in-place**, so **space complexity is O(log n)** for the recursive stack.
- Fast and stable for most use cases.

• Question: Which loop (for or foreach) is more efficient for calculating the sum of an array, and why?

Both for and foreach loops can be used to calculate the sum of an array, but in terms of **efficiency**, the **for loop is slightly more efficient**.

✅ **Why for is more efficient:**

Factor	Explanation
🔄 Indexing	for directly accesses elements via index (arr[i]), which is fast.
🔄 foreach	Internally uses an **enumerator**, which adds **a small overhead** in performance.
⚙️ Control	for gives **more control** over iteration, including ability to skip elements, etc.

🧪 **Example:**

```
int[] arr = { 1, 2, 3, 4, 5 };

int sum = 0;

// Using for loop

for (int i = 0; i < arr.Length; i++)

{

   sum += arr[i];

}

// Using foreach loop

foreach (int num in arr)

{

   sum += num;

}
```

Performance difference:

- The difference is **very small** and only noticeable for **very large arrays**.
- For **readability and simplicity**, foreach is usually preferred.
- For **maximum performance**, especially in **performance-critical code**, for is better.

✅ **Conclusion:**

- ✓ **For performance-critical scenarios**: Use for.
- ✓ **For cleaner and safer code**: Use foreach.

PART03

✅ 5. What's the default size of stack and heap in C#? What are the considerations?

📌 **Stack:**

- **Default size (for 64-bit Windows):**
  Usually **1 MB per thread**, but it can vary depending on:
  - OS
  - .NET version
  - Whether you're in a console app, ASP.NET, etc.

📌 **Heap:**

- **Heap size:**
  There is **no fixed limit**—it's only constrained by:
  - Available system memory
  - Process architecture (32-bit vs 64-bit)

🧠 **Considerations:**

Feature	Stack	Heap
Allocation	Fast (LIFO: Last In, First Out)	Slower (managed by Garbage Collector)
Storage	Value types, method calls, local vars	Reference types, objects, arrays
Size	Limited (e.g., ~1MB)	Large (dynamic, grows as needed)
Lifetime	Automatic (when method ends)	Manual (collected by GC)
Error risk	Stack Overflow	OutOfMemoryException

⚠️ **Why care?**

- Large arrays or objects should go on the **heap**.
- Avoid **deep recursion** or large structs on the **stack** to prevent **StackOverflowException**.
- Use **value types** for performance-critical, short-lived data.

✅ **6. What is Time Complexity?**

📌 **Definition:**

Time complexity is a measure that describes **how the runtime of an algorithm grows** relative to the input size n.

⏱️ **Examples:**

Time Complexity	Description	Example
O(1)	Constant time	Accessing array element: arr[0]
O(n)	Linear time	Looping through array
$O(n^2)$	Quadratic time	Nested loops
O(log n)	Logarithmic	Binary search
O(n log n)	Linearithmic	Merge Sort, Quick Sort (avg)
$O(2^n)$, O(n!)	Exponential / Factorial	Brute-force problems

🔍 **Why is it important?**

- Helps predict **performance** and **scalability**.
- Helps choose the **best algorithm** for large input sizes.
- Especially important in **interviews**, **optimization**, and **system design**.

🧠 **Example:**

```
// O(n)
void PrintAll(int[] arr)
{
   for (int i = 0; i < arr.Length; i++)
      Console.WriteLine(arr[i]);
}

// O(1)
void PrintFirst(int[] arr)
{
   Console.WriteLine(arr[0]);
}
```