Q4) Explain the difference between a runtime error and a logical error with examples.

Both **runtime errors** and **logical errors** are types of bugs that can occur during program execution, but they differ in **when** and **how** they occur.

Runtime Error:

A **runtime error** occurs **while the program is running**. These errors typically cause the program to crash or throw an exception.

```
int x = 10;

int y = 0;

int result = x / y; / / Division by zero → runtime error

Console. WriteLine(result);
```

Output:System.DivideByZeroException: Attempted to divide by zero.

A **logical error** happens when the program runs **without crashing**, but produces the **wrong output** due to incorrect logic.

```
int x = 10;

int y = 5;

int result = x - y;  //  Suppose the intention was to add

Console.WriteLine("Sum = " + result);
```

Output:Sum = 5 // Wrong! It should be 15 if we wanted to add

Q6) Why is it important to follow naming conventions such as PascalCase in C#?

1. Improves Readability

PascalCase helps distinguish different types of identifiers. In C#, PascalCase is used for:

- **Class names**: `Student`, `CustomerAccount`
- **Method names**: `CalculateSalary()`, `DisplayInfo()`

2. Promotes Consistency

- Using a consistent naming pattern across a project (or team) ensures that code looks familiar regardless of who wrote it. This reduces confusion and improves collaboration.

Q7) Explain the difference between value types and reference types in terms of memory allocation?

1. Value Types

🔹 **Stored In:**

- **Stack memory**

🔹 **Behavior:**

- A **copy of the value** is stored directly in the variable.
- When you assign one value type variable to another, a **new copy** is created.

```
int x = 10;

int y = x;  // y gets a copy of x's value

y = 20;

Console.WriteLine(x);  // Outputs 10 (x is unchanged)
```

Common Value Types:

int, float, bool, char, struct, enum

2. Reference Types

🔹 **Stored In:**

- **Reference (pointer) is on the Stack**,
  but the **actual object** is stored in the **Heap**.

🔹 **Behavior:**

- The variable holds a **reference (memory address)** to the object in the heap.

- Assigning one reference variable to another copies the **reference**, not the object.

```
class Person { public string Name; }

Person p1 = new Person();

p1.Name = "Alice";

Person p2 = p1;  // p2 and p1 point to the same object

p2.Name = "Bob";

Console.WriteLine(p1.Name);  // Outputs: Bob
```

Common Reference Types:

class, object, string, array, delegate

Question: What will be the output of the following code? Explain why:

The `%` operator in C# is the **modulus operator**, which returns the **remainder** after division.

In this case:

- `a = 2`
- `b = 7`
- Since 2 is **less than** 7, you cannot divide 2 into 7 even once.

`2 ÷ 7 = 0 remainder 2`

Question: How does the && (logical AND) operator differ from the & (bitwise AND) operator?

1. `&&` — Logical AND Operator

- **Used with boolean expressions**
- **Short-circuits**: If the first condition is false, the second is **not evaluated**.
- Used for **decision-making**, e.g., in `if` statements.

2. `&` — Bitwise AND Operator

- **Used with integers** to compare individual bits.
- Can also be used with `bool`, but **does not short-circuit** — both sides are **always evaluated**.

```
int a = 6; // 0110

int b = 3; // 0011

int result = a & b; // 0010 = 2


Console. WriteLine(result); // Output: 2
```

Question: Why is explicit casting required when converting a double to an int?

Explicit casting is required when converting a `double` to an `int` **because the conversion may result in data loss** — specifically, the fractional (decimal) part of the number is **truncated**, not rounded.

- A `double` (e.g., `5.9`) stores **decimal values** using floating-point representation.
- An `int` only stores **whole numbers**.
- So, converting from `double → int` might **lose precision**.

Because of this **potential loss of information**, C# forces you to use **explicit casting** with `(int)` to make sure you're aware of it.

Question: What exception might occur if the input is invalid and how can you handle it?

If the input is invalid (e.g., not a number like "abc" or "12.3"), calling int.Parse() will throw a:

System.FormatException

This happens because the input string does not represent a valid integer.

Question: Given the code below, what is the value of x after execution? Explain why
int x = 5;
int y = ++x + x++;

Step 1: `++x`

- Prefix increment → increment `x` first:
  `x` becomes `6`, and the value used in the expression is `6`.

Step 2: `x++`

- Postfix increment → use `x` first, then increment:
  - Value used is still `6`.
  - After this, `x` becomes `7`.

---

1)LinkedIn article about variables allocation in stack and heap for both value and ref types

Mahmoud Akl

Software Engineer | Backend Developer(ASP.NET)

10th of Ramadan, Sharkia

Software Engineer | Backend Developer(ASP.NET)

1m •

لو إنت مبرمج وبتتعلم C#، أكيد سمعت عن الـ Stack والـ Heap، وسمعت إن في فرق ما بين value types وreference types. بس إيه الكلام ده؟ ويعني إيه الواحد يهتم بيه؟ تعالى أقولك الحكاية بالراحة 👇

🧠 الأول: يعني إيه Stack و Heap؟

: Stack

ده زي شنطة ضهر صغيرة كده، بنحط فيها حاجات خفيفة وسريعة الاستخدام. بتشتغل بنظام "اللي يدخل آخر، يطلع أول" (LIFO). ف كل ما تستدعي Method، بتاخد مكان على Stack، ولما تخلص، تتشال.

: Heap

دي زي الدولاب الكبير في البيت، بنحط فيه الحاجات الكبيرة أو اللي بنستخدمها على فترات طويلة. مش بنفس السرعة بتاعة الستاك، بس بيسمحلك تخزن حاجات حجمها كبير أو معقدة شوية.

🧪 الفرق ما بين Value Type و Reference Type

✅ Value Types (زي int, double, bool...):

بيتخزنوا في Stack.

كل متغير بياخد نسخة مستقلة من القيمة.

```
int a = 10;
int b = a;
b = 20;
Console.WriteLine(a); // هتطبع 10
```

✅ ليه؟ عشان b أخد نسخة من a، وأي تعديل عليه مش هيأثر على a.

🔗 Reference Types (زي class, array, string, object...):

بيتخزنوا في Heap.

المتغير بيخزن (Reference) للحاجة، مش القيمة نفسها.

```
int[] arr1 = {1, 2, 3};
int[] arr2 = arr1;
arr2[0] = 99;
Console.WriteLine(arr1[0]); // هتطبع 99
```

❗ ليه؟ عشان arr1 و arr2 الاتنين بيشاوروا على نفس المكان في الهيب.

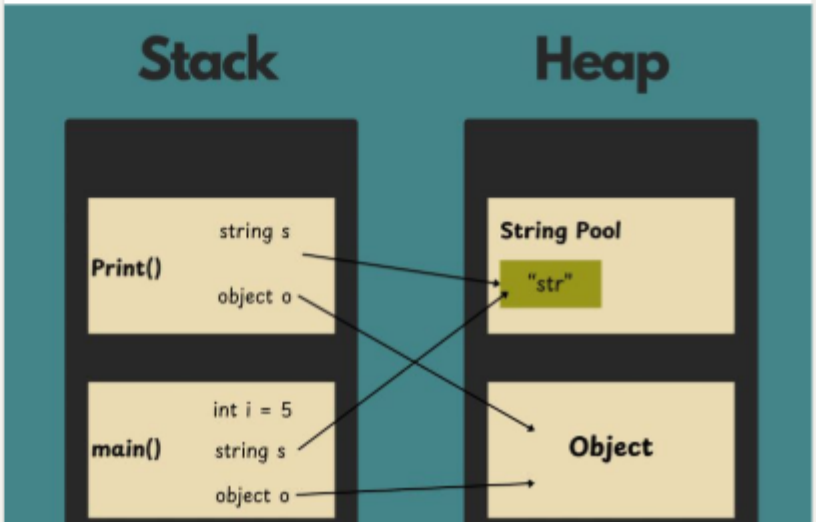
🤔 ليه الكلام ده مهم؟

الأداء: Stackأسرع بكتير، عشان بيتم الوصول ليه مباشرة.

إدارة الذاكرة: C# بتستخدم Garbage Collector عشان تنظف Heap، بس مش بتشيل Stack.

السلوك المتوقع: لازم تبقى فاهم إمتى الحاجة هتتعدل وإمتى هتفضل زي ما هي لما تبعتها Method أو تعمل منها نسخة.

#StackVsHeap# #CSharp #برمجة_بالمصري #DotNet #معلومة_بسيطة



2)What's the Difference Between Compiled and Interpreted Languages?

Compiled Languages:

- The **whole code** is **translated into machine code (binary) before** running the program.
- This happens using a **compiler**
- The output is an **executable file** (`.exe`, `.out`, etc.)

Pros:

- **Faster** execution.
- Better **performance optimization**.

Cons:

- Need to **compile again** after every change.
- Harder to debug during development.

Examples: C, C++, Rust, Go.

Interpreted Languages:

- Code is **executed line-by-line** by an **interpreter** (like Python or JavaScript engine).
- No pre-generated executable.

Pros:

- Easy to **test and debug**.
- Good for **scripting** and **dynamic behavior**.

Cons:

- Slower runtime.
- More **resource usage**.

 Examples: Python, JavaScript, Ruby.

What about C#?

C# is a **hybrid** — it's **compiled AND interpreted** in a way.
Here's how it works:

Step-by-Step:

1. Your C# code is first **compiled** into **Intermediate Language (IL)** using the **C# compiler** (`csc.exe`).
2. That IL code is stored in a **.NET assembly** (like `.exe` or `.dll`).
3. Then, when you run the program, the **Common Language Runtime (CLR)** takes over:
    - It uses a **Just-In-Time (JIT) compiler** to **convert IL into machine code at runtime**.

C# is:

- ✅ **Compiled** (to IL)
- ✅ **Then interpreted** (via **JIT compilation** at runtime)

3- Compare between implicit, explicit, Convert and parse casting?

Type	Definition	Example	Notes
Implicit	Automatic conversion from smaller to larger type (no data loss).	int a = 10; double b = a;	Safe and doesn't need extra syntax.
Explicit	Manual conversion from larger to smaller type (possible data loss).	double a = 10.5; int b = (int)a;	Requires casting — be cautious with precision loss.
Convert	Uses the Convert class to change between types.	string s = "123"; int a = Convert.ToInt32(s);	Handles null, throws exception if conversion fails.
Parse	Converts string to numeric type.	string s = "123"; int a = int.Parse(s);	Throws exception on null or invalid format — no null support.

```
// باستخدام Convert
string num1 = null;
int result1 = Convert.ToInt32(num1); // 0 يرجّع

// باستخدام Parse
string num2 = null;
int result2 = int.Parse(num2); // يرمي استثناء
```

Self Study

1)How to Customize the Garbage Collector in C#

In C#, you cannot directly control when the garbage collector (GC) runs, but you can influence it through the following methods:

- **Forcing a GC collection manually**:

```
GC.Collect(); // Forces an immediate garbage collection

GC.WaitForPendingFinalizers(); // Waits for finalizers to complete
```

- **Controlling object lifetime**:
  - Use `using` blocks or `Dispose()` for deterministic cleanup.
  - Call `GC.SuppressFinalize(this)` inside a `Dispose` method to skip finalization.

- **Changing GC mode**:

You can choose between *workstation* and *server* garbage collection in `app.config` or `runtimeconfig.json` for performance tuning:

```
<configuration>
 <runtime>
  <gcServer enabled="true"/>
 </runtime>
</configuration>
```

2) Bitwise Operators in C#

Operator	Name	Example (a = 5, b = 3)	Result (binary)
&	AND	a & b → 5 & 3	00000101 & 00000011 = 00000001 (1)
`	`	OR	`a
^	XOR	a ^ b → 5 ^ 3	00000101 ^ 00000011 = 00000110 (6)
~	NOT	~a → ~5	~00000101 = 11111010 (-6)
<<	Left Shift	a << 1 → 5 << 1	00000101 → 00001010 (10)
>>	Right Shift	a >> 1 → 5 >> 1	00000101 → 00000010 (2)

---

3)What "Managed Code" Means in C#

1. **Memory Management**
   - The CLR automatically handles memory allocation and garbage collection (GC).
   - You don't need to manually allocate (`malloc`) or free (`free`) memory like in C/C++.
2. **Type Safety**
   - The CLR ensures that variables are only used in ways that are consistent with their data types.
3. **Exception Handling**
   - Runtime exceptions are handled safely using try/catch blocks without crashing the system.
4. **Security**
   - Managed code runs in a secure environment called the *managed execution environment*, which enforces code access security and verification.
5. **Cross-Language Interoperability**
   - Managed code written in C#, VB.NET, or F# can interact seamlessly since they all compile to **Common Intermediate Language (CIL)**.

---

4)What It Means: "Struct is like a class before"

- Structs and classes both:
  - Can have fields, properties, methods, and constructors.
  - Use access modifiers (like `public`, `private`).

- Can implement interfaces.

That's why a `struct` *looks* like a class — but under the hood, they are **very different**.

Feature	`struct`	`class`
Type	Value type	Reference type
Stored in	Stack (usually)	Heap
Inheritance	Cannot inherit from another struct/class	Can inherit from other classes
Default constructor	Cannot define a parameterless constructor	Can define it freely
Nullability	Cannot be null (unless nullable struct)	Can be null
Performance	Faster for small data	Better for large/complex objects

```
struct Point
{
    public int X;
    public int Y;

    public void Print() => Console.WriteLine($"X={X}, Y={Y}");
}
```

```
Point p1 = new Point();
Point p2 = p1;  // COPY - value type behavior
p2.X = 100;

Console.WriteLine(p1.X); // Output: 0 (not affected by p2)
```