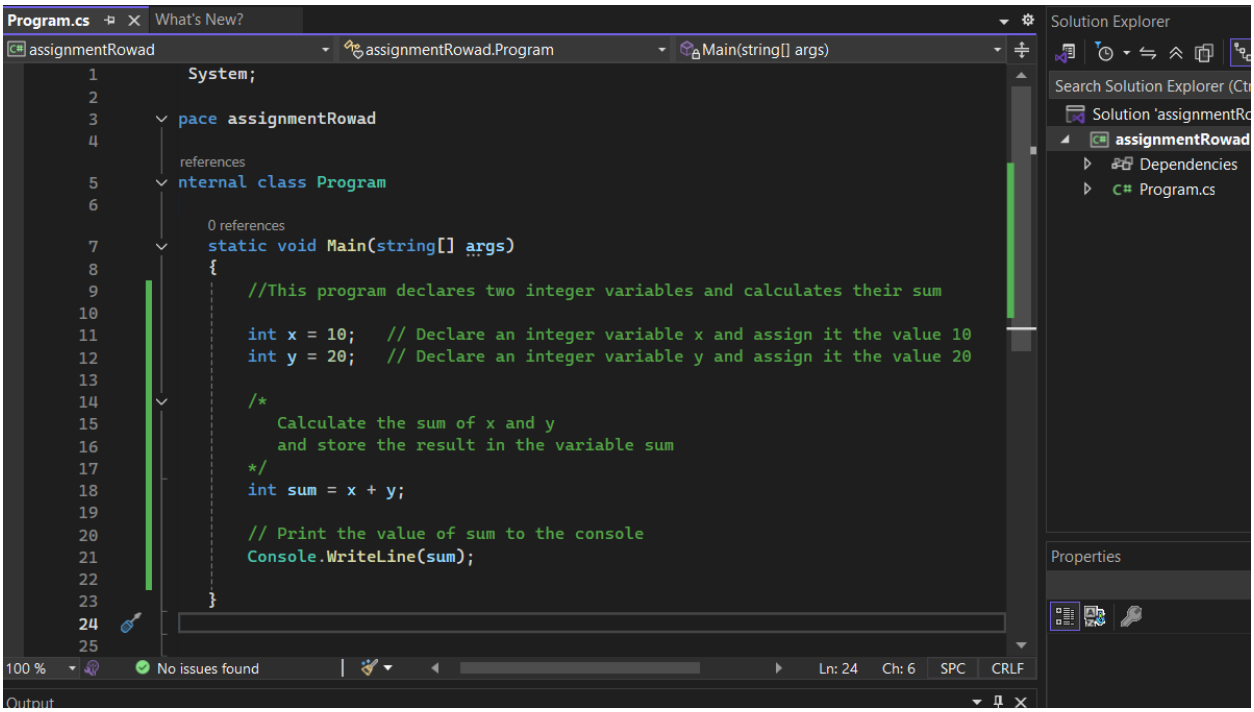
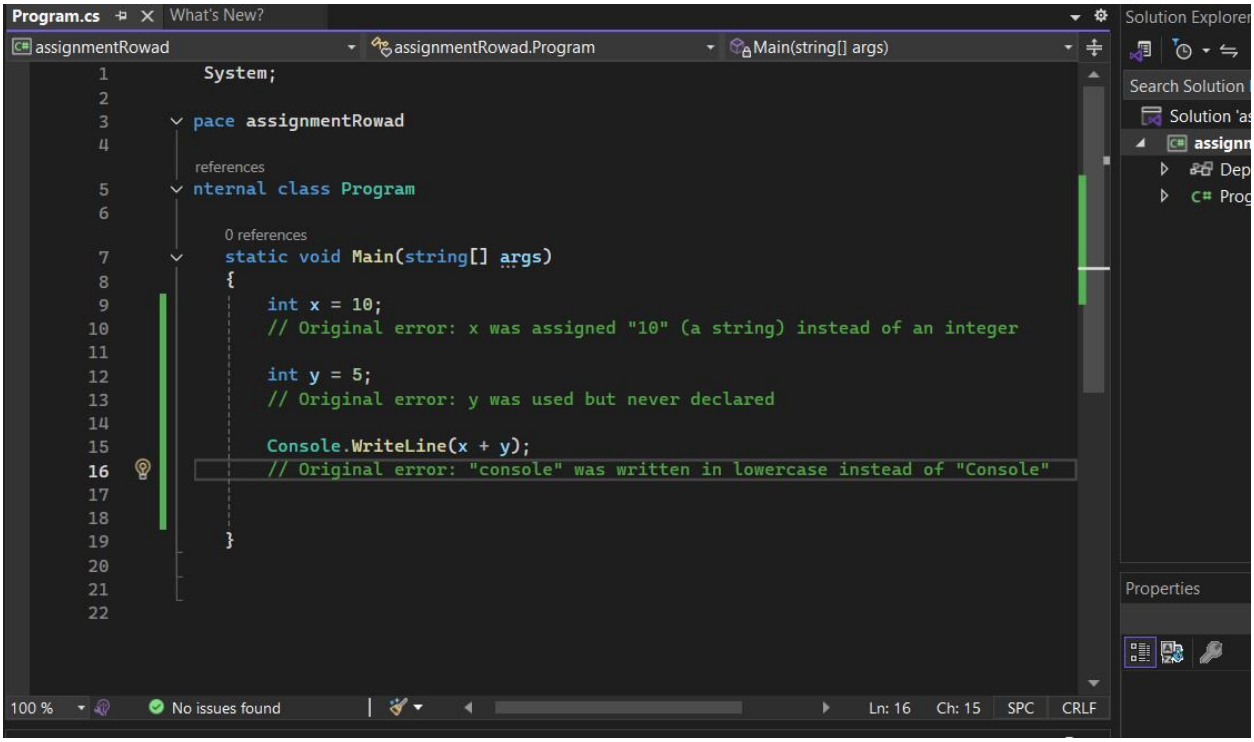
Part01:



In Visual Studio, the shortcuts are:

- **Comment selected code:**
  Ctrl + K, Ctrl + C
- **Uncomment selected code:**
  Ctrl + K, Ctrl + U



Runtime Error:

An error that occurs while the program is running. The program compiles but crashes during execution.
Example: dividing by zero.

Example:

```
int x = 10;

int y = 0;

int result = x / y;   // Runtime error: division by zero
```

Logical Error

An error in the program logic. The program runs normally but gives incorrect results.
Example: using subtraction instead of addition.

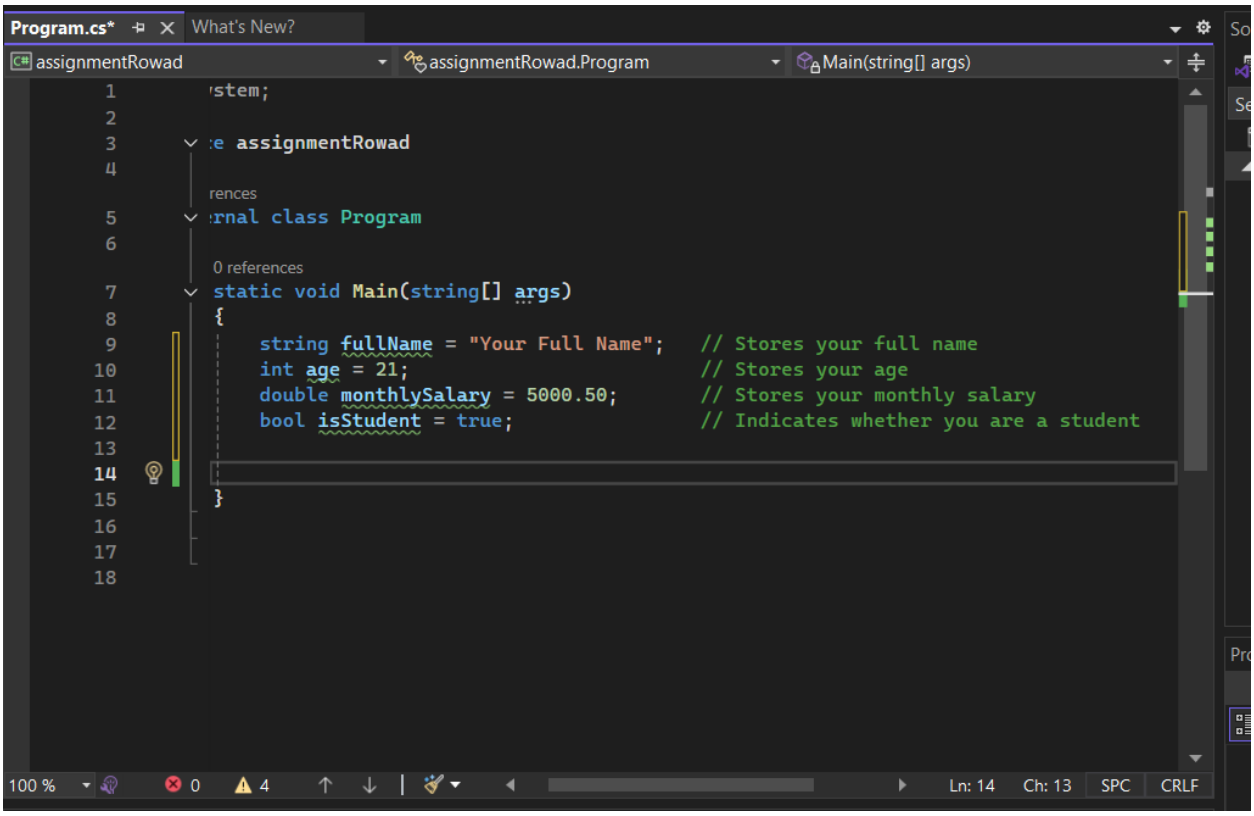
Example:

int x = 10;

int y = 5;

int result = x - y; // Logical error: should be x + y

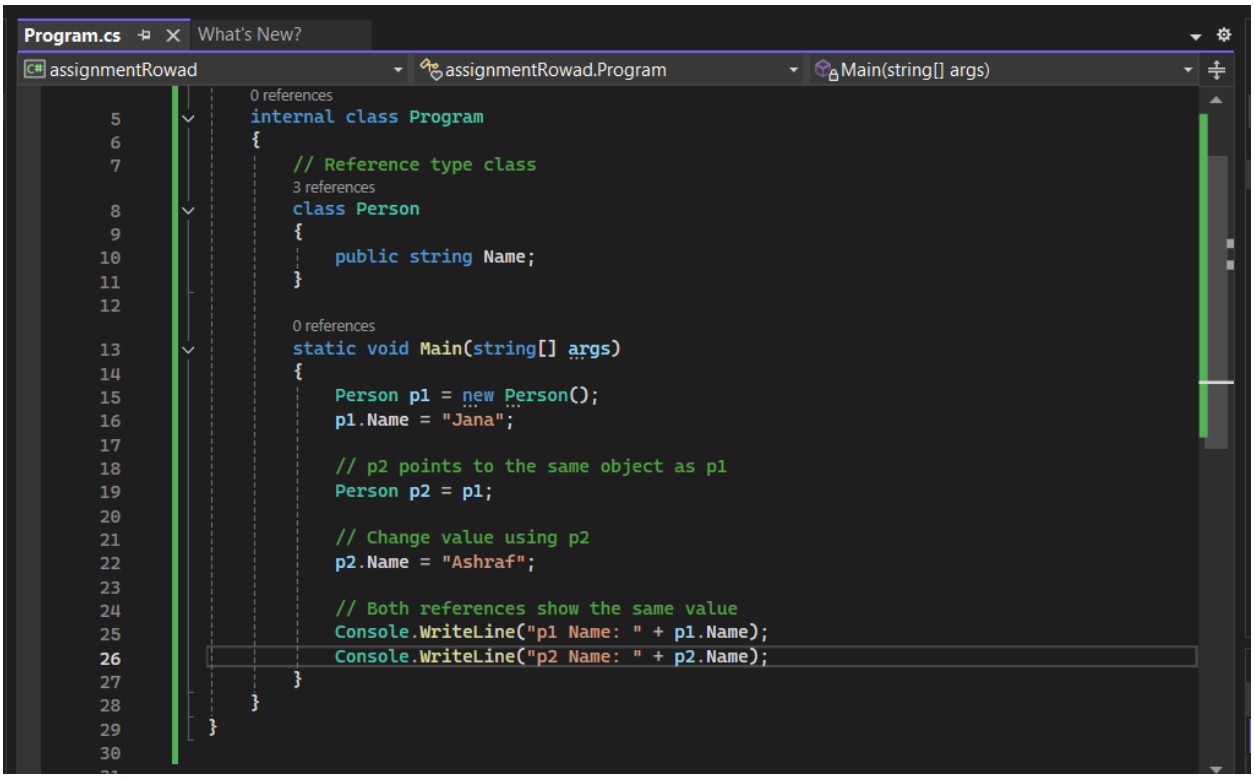
Console.WriteLine(result);

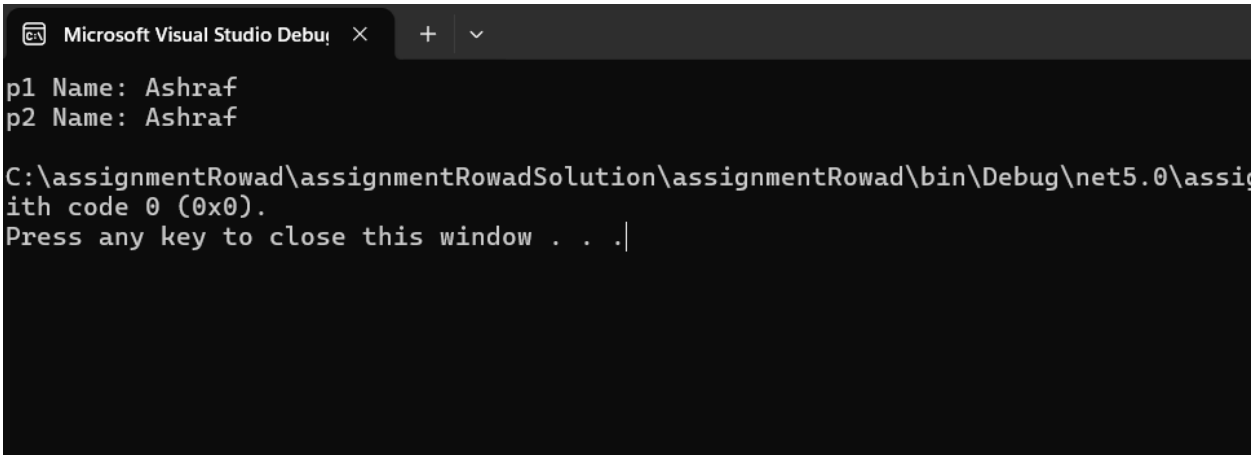


It's important to follow naming conventions like PascalCase in C# because:

- **Improves code readability – code is easier to read and understand.**

- **Makes code more maintainable – easier for you and others to modify later.**
- **Follows C# standards – matches .NET guidelines and professional practices.**
- **Reduces confusion – helps distinguish between variables, methods, and classes.**
- **Improves teamwork – everyone writes code in a consistent way.**





Value Types

- **Stored in the stack**
- **Contain actual value**
- **Assigning creates a copy**

int a = 5;

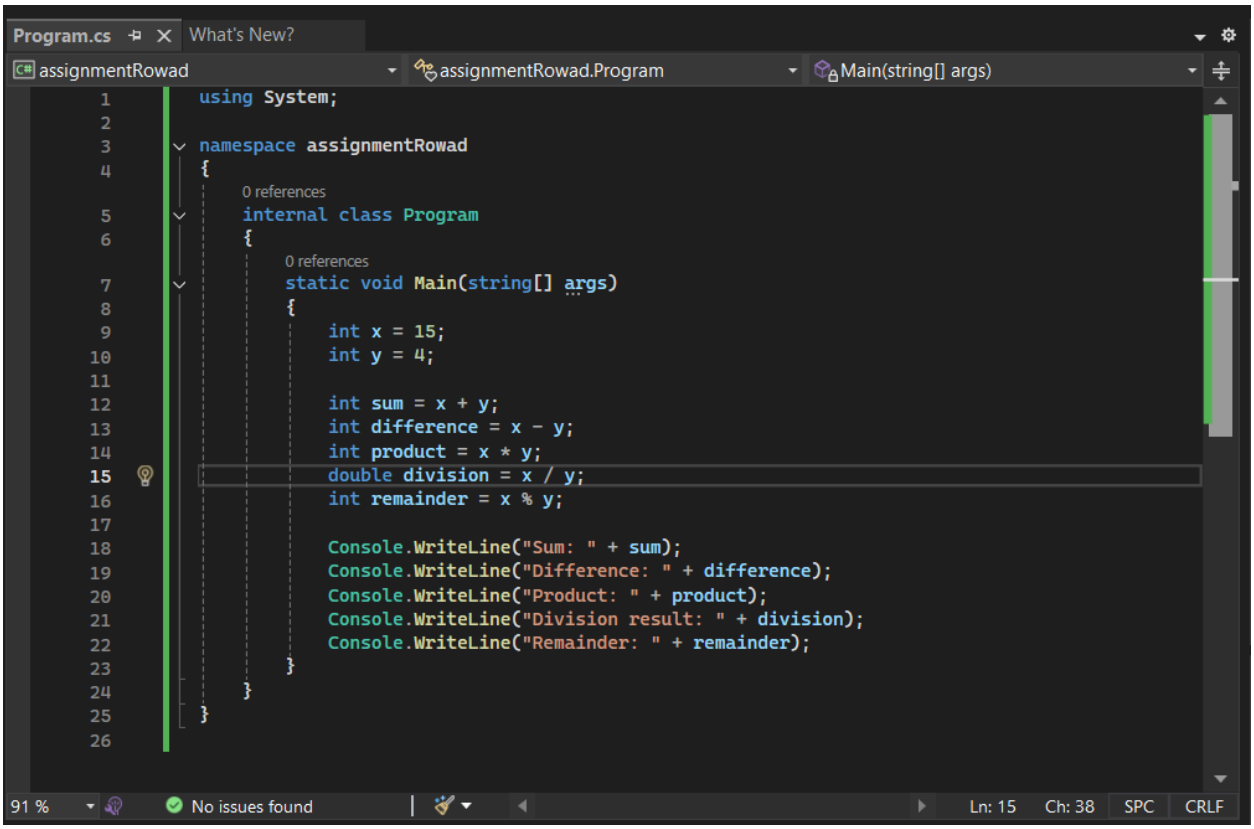
int b = a; // b gets a copy

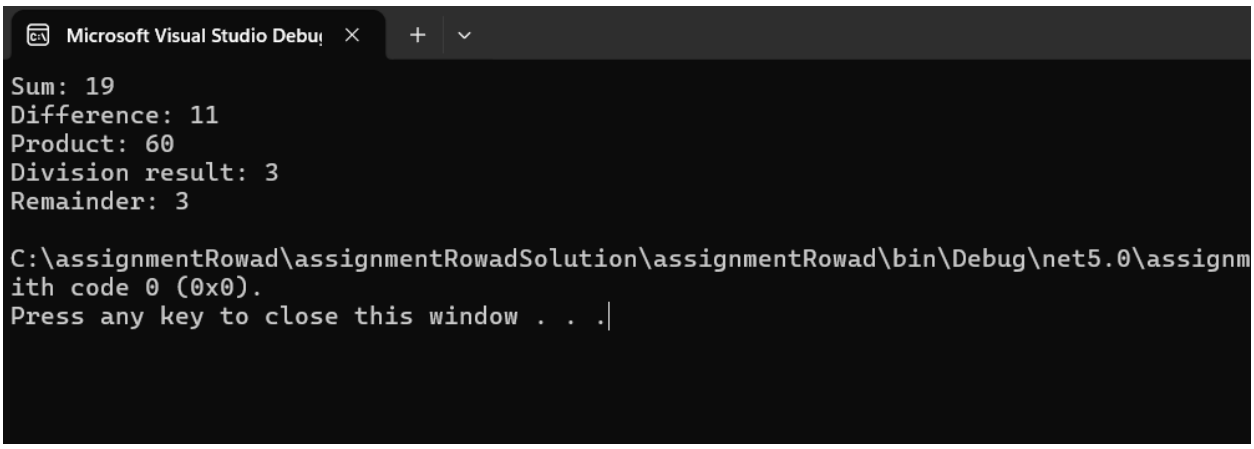
Reference Types

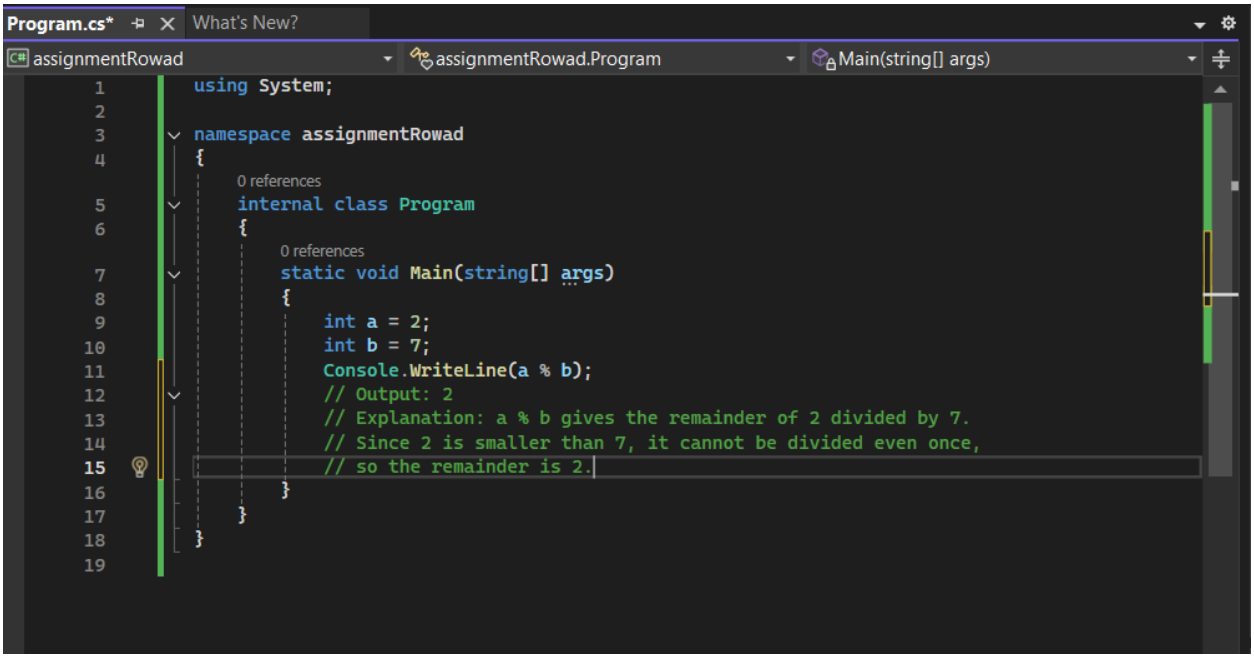
- **Stored in the heap**
- **Variables store a reference (address)**
- **Assigning shares the same object**

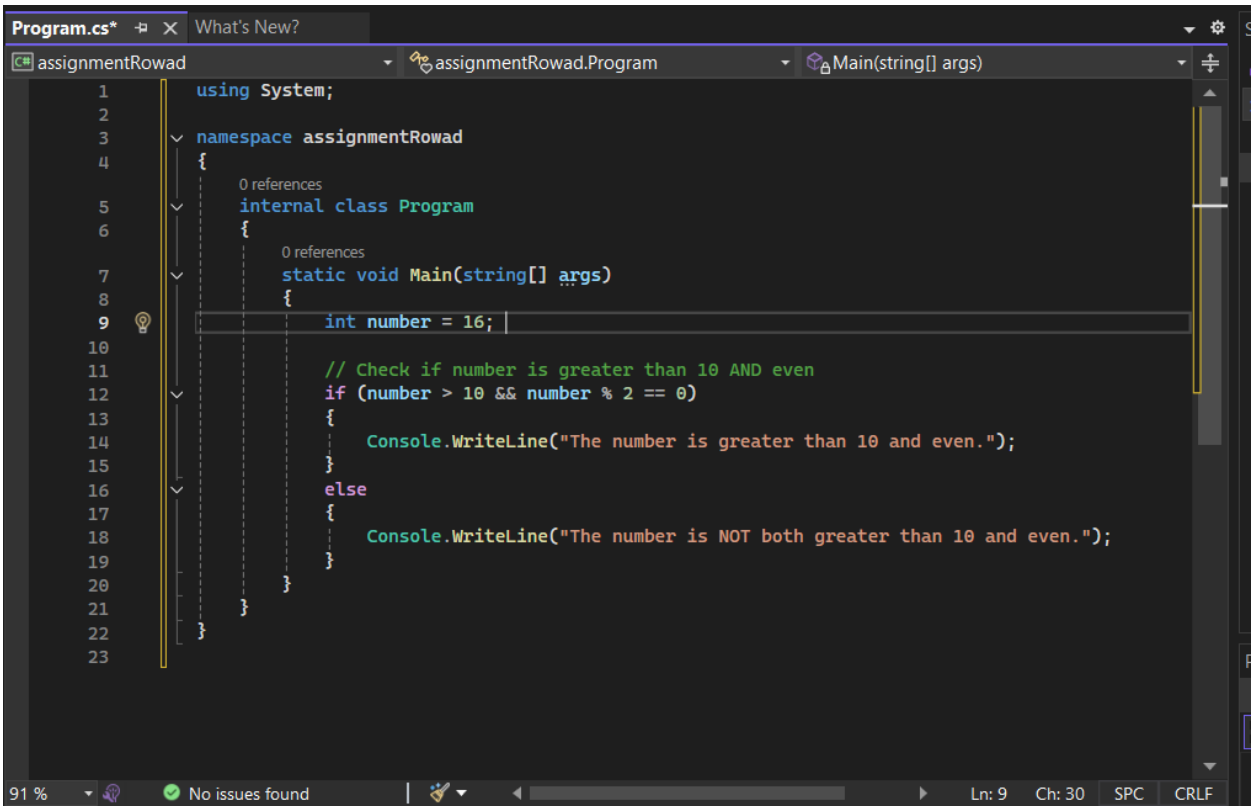
Person p1 = new Person();

Person p2 = p1; // both point to same object


Program.cs
What's New?
assignmentRowad
assignmentRowad.Program
Main(string[] args)
using System;
namespace assignmentRowad
{
0 references
internal class Program
{
0 references
static void Main(string[] args)
{
int x = 15;
int y = 4;
int sum = x + y;
int difference = x - y;
int product = x * y;
double division = x / y;
int remainder = x % y;
Console.WriteLine("Sum: " + sum);
Console.WriteLine("Difference: " + difference);
Console.WriteLine("Product: " + product);
Console.WriteLine("Division result: " + division);
Console.WriteLine("Remainder: " + remainder);
}
}
}
91 %
No issues found
Ln: 15
Ch: 38
SPC
CRLF


Microsoft Visual Studio Debu
Sum: 19
Difference: 11
Product: 60
Division result: 3
Remainder: 3
C:\assignmentRowad\assignmentRowadSolution\assignmentRowad\bin\Debug\net5.0\assignm
ith code 0 (0x0).
Press any key to close this window . . .


Program.cs*
What's New?
assignmentRowad
assignmentRowad.Program
Main(string[] args)
using System;
namespace assignmentRowad
{
0 references
internal class Program
{
0 references
static void Main(string[] args)
{
int a = 2;
int b = 7;
Console.WriteLine(a % b);
// Output: 2
// Explanation: a % b gives the remainder of 2 divided by 7.
// Since 2 is smaller than 7, it cannot be divided even once,
// so the remainder is 2.
}
}
}


Program.cs*
What's New?
assignmentRowad
assignmentRowad.Program
Main(string[] args)
using System;
namespace assignmentRowad
{
0 references
internal class Program
{
0 references
static void Main(string[] args)
{
int number = 16;
// Check if number is greater than 10 AND even
if (number > 10 && number % 2 == 0)
{
Console.WriteLine("The number is greater than 10 and even.");
}
else
{
Console.WriteLine("The number is NOT both greater than 10 and even.");
}
}
}
}
91 %
No issues found
Ln: 9
Ch: 30
SPC
CRLF

&& (Logical AND)

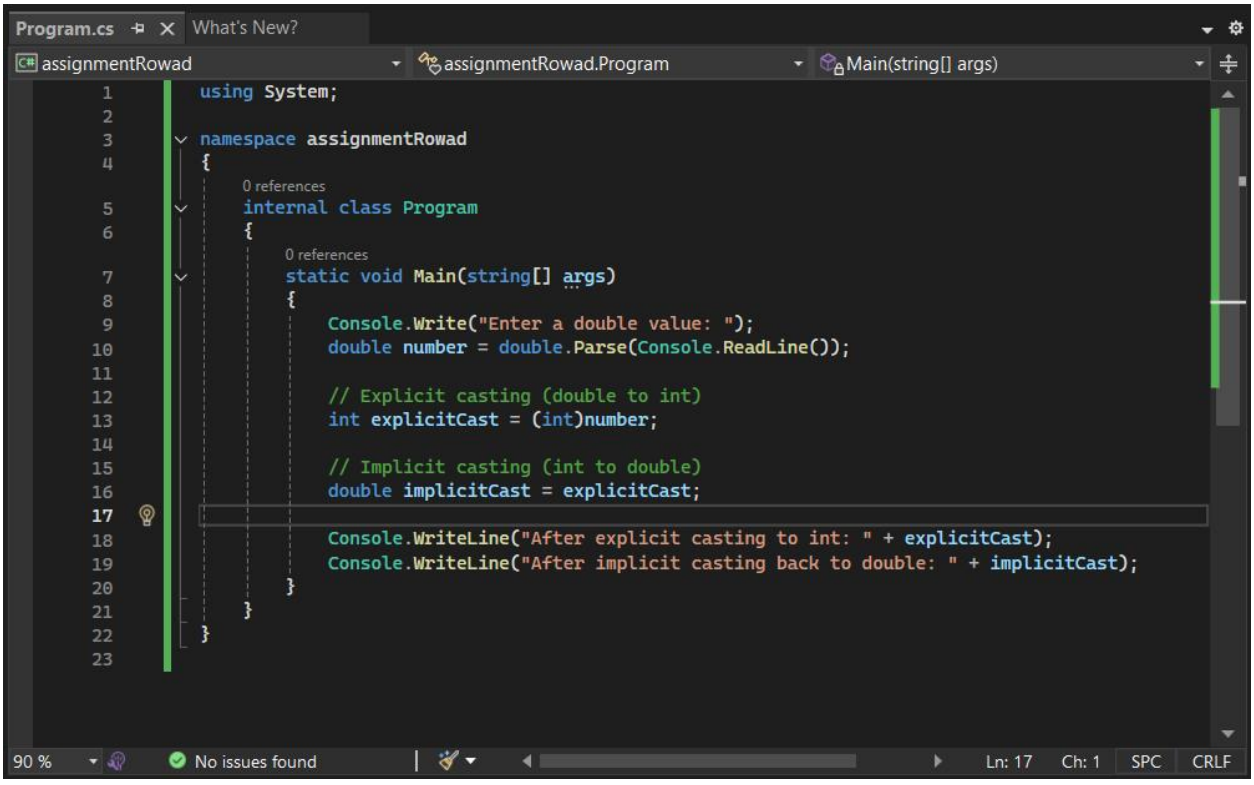
- **Used with boolean conditions**
- **Short-circuit: if the first condition is false, the second is not evaluated**
- **Commonly used in if statements**

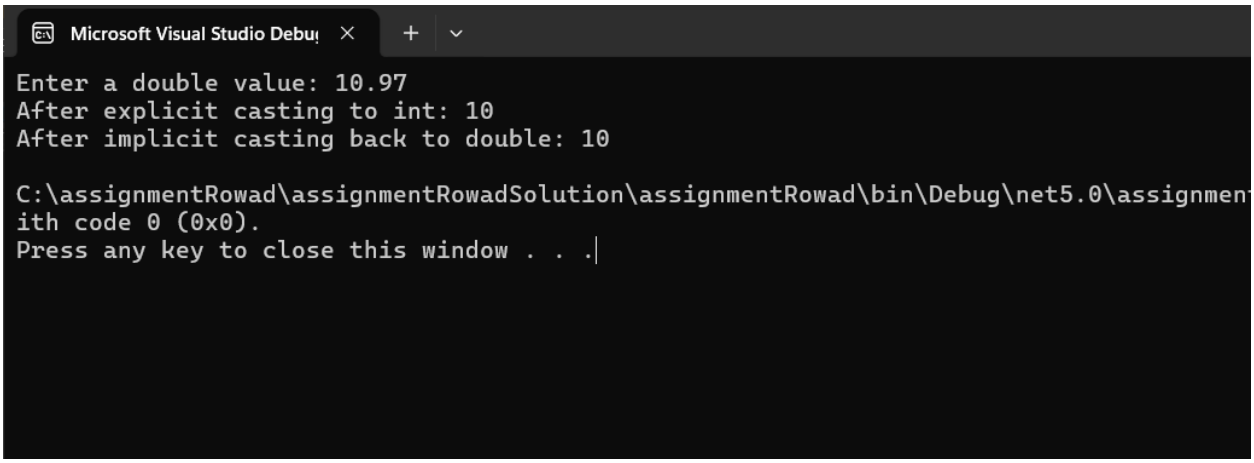
if (a > 0 && b > 0)

& (Bitwise AND)

- **Used with numbers (bit by bit) or booleans**
- **No short-circuit: both sides are always evaluated**
- **Operates on binary representation**

int result = 5 & 3; // 101 & 011 = 001 → 1





Explicit casting is required when converting a double to an int because:

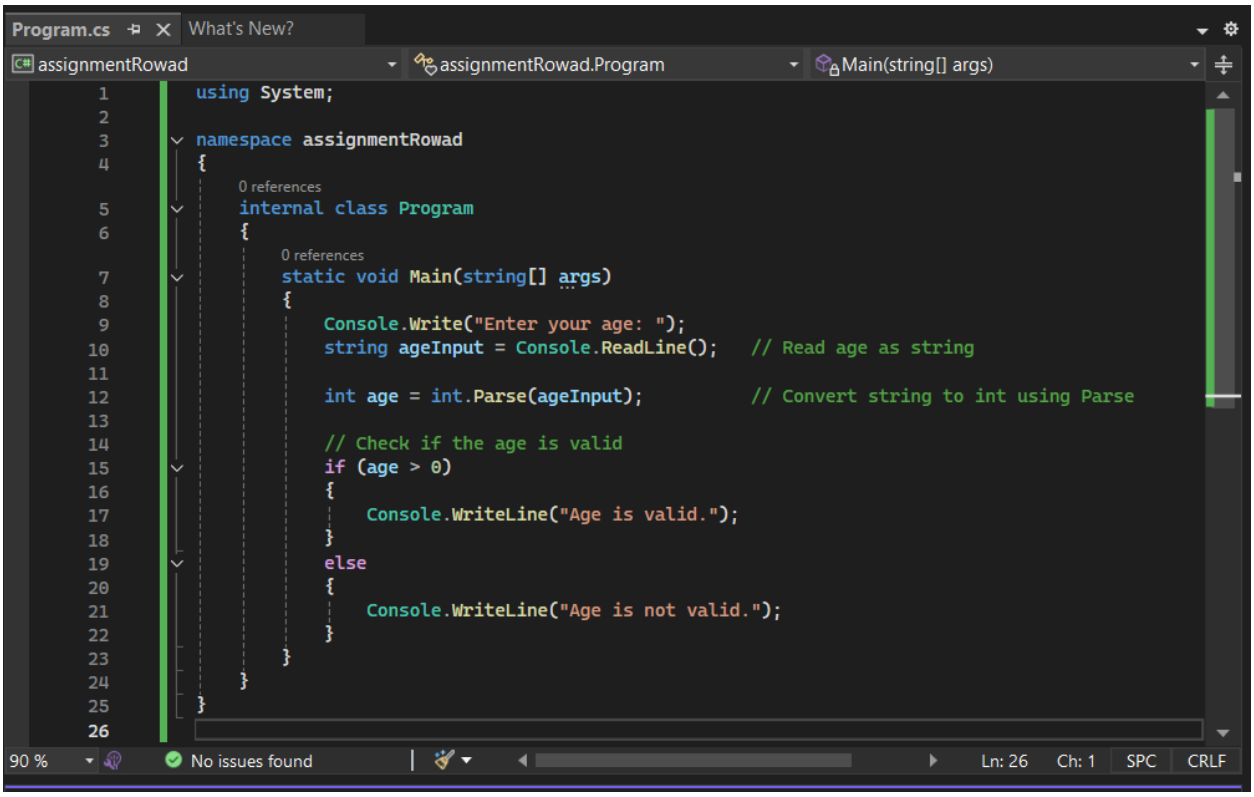
- **A double can contain decimal values, while an int cannot.**
- **Converting may cause data loss (the decimal part is removed).**

- **C# forces explicit casting to make sure the programmer is aware of this loss.**

Example:

(int)3.7 becomes 3

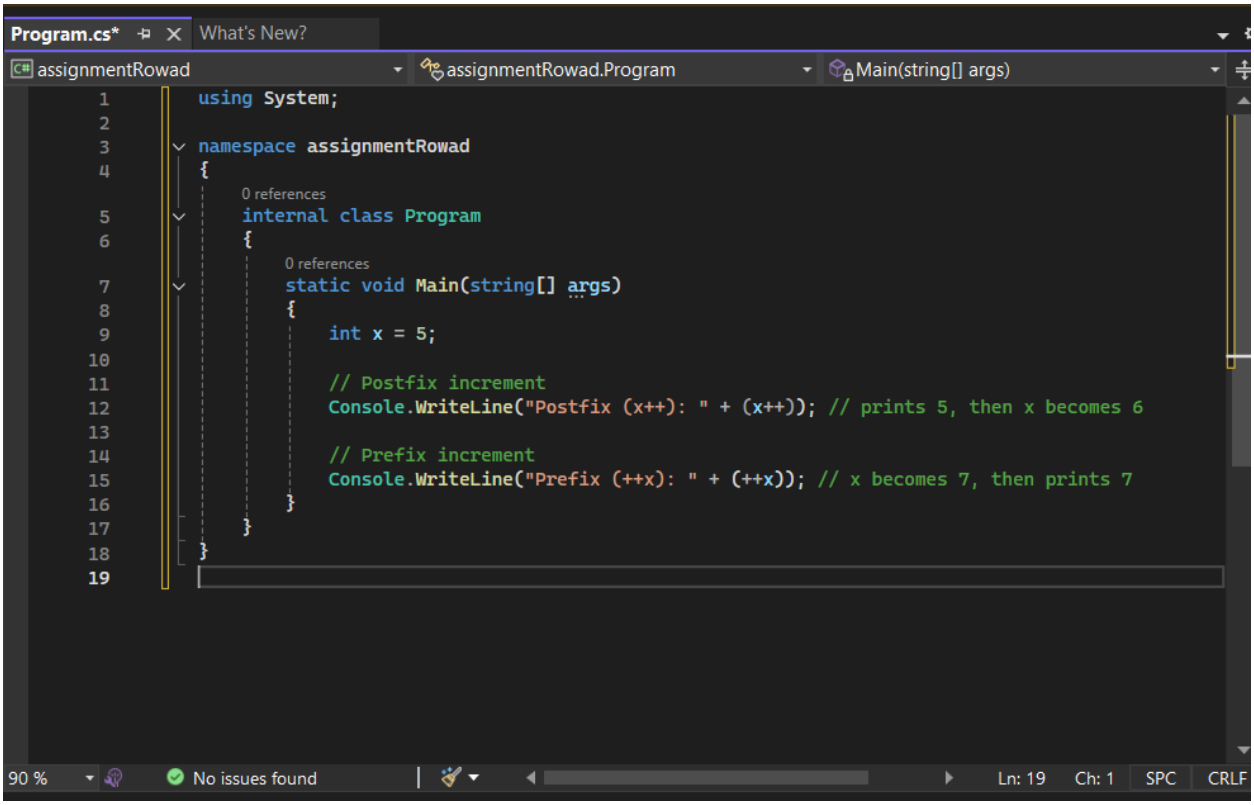
So, explicit casting prevents accidental loss of information.

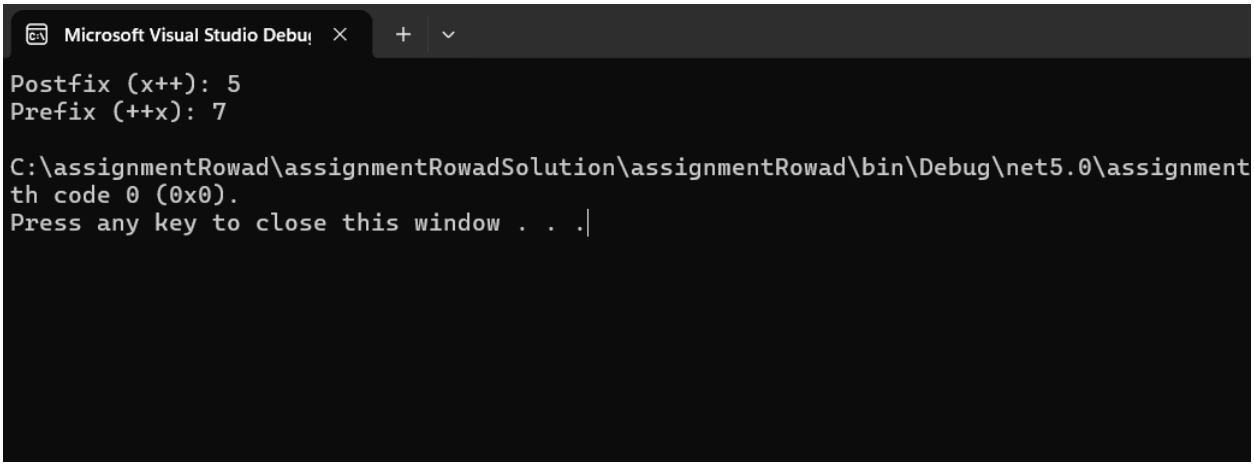


- **FormatException: occurs when the input is not a valid number (for example, letters instead of digits).**

- **OverflowException: occurs when the input number is too large or too small to fit in an int.**

These exceptions can be handled using exception handling (try-catch) to prevent the program from crashing and to show a proper error message.

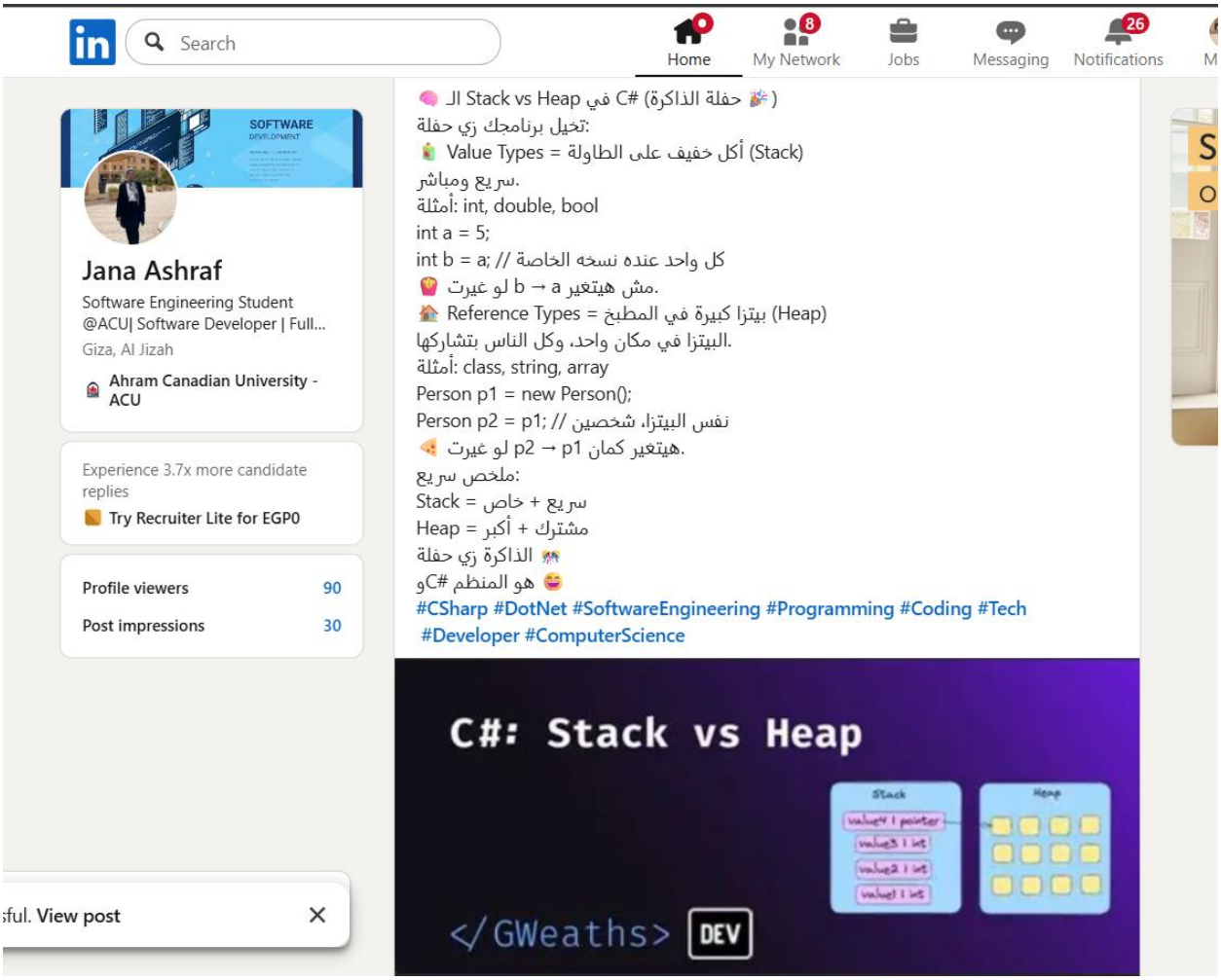




- **Given the code below, what is the value of x after execution? Explain why int x = 5; int y = ++x + x++; : ++x increases x first → x becomes 6**
- **x++ uses the value first, then increases → x becomes 7**

So after execution, x = 7.

Part02:



✅ Compiled Languages

- **Code is converted into machine code before running.**
- **The program runs fast.**
- **Examples: C, C++**

What happens:
Source code → Compiler → Machine code → Run

✅ Interpreted Languages

- **Code is executed line by line at runtime.**
- **Usually slower than compiled languages.**
- **Examples: Python, JavaScript**

What happens:
Source code → Interpreter → Runs line by line

🔍 So what about C#?

C# is both compiled and interpreted.

🟦 How it works:

1. **Compiled into Intermediate Language (IL)**
   (not machine code yet)

2. Executed by the .NET runtime (CLR) which interprets or JIT-compiles it into machine code

✅ 1. Implicit Casting (Automatic)

- **No data loss**
- **Automatic conversion**
- **Usually from smaller type to larger type**

📌 **Example:**

```
int a = 10;
double b = a;   // implicit casting
```

✅ 2. Explicit Casting (Manual)

- **Possible data loss**
- **You must write the cast**
- **Used when converting from larger type to smaller type**

📌 **Example:**

```
double a = 9.8;
```

```
int b = (int)a;   // explicit casting
```

✅ 3. Convert Class

- **Converts between different data types**
- **Handles null values and returns default values**
- **Safer than Parse**

📌 **Example:**

```
string s = "123";
int n = Convert.ToInt32(s);
```

✅ 4. Parse Method

- **Converts string to a specific type**
- **Throws exception if string is invalid**
- **Faster but less safe**

📌 **Example:**

```
string s = "123";
int n = int.Parse(s);
```

Part03 Bonus:

Sure! Here is a clean and organized self-study summary for C# Session 2, covering all the topics you listed.
You can use this as a quick revision sheet or study notes.

⭐ C# Session 2 Self-Study Summary

✅ 1. Creating Objects (Reference Types)

- **Reference types** (class, array, string, etc.) are stored in the **heap**.
- When you create an object using new, the **reference** is stored in the **stack**.
- **Example:**
- **Person p = new Person();**

✅ 2. Memory Leak

- Occurs when **unused objects are still referenced**, so memory can't be freed.
- Common cause: **event handlers, static lists, long-lived references.**

✅ 3. Garbage Collector (GC)

- **Automatically frees unreferenced objects from the heap.**
- **Runs when needed (not predictable).**
- **Helps prevent memory leaks.**

✅ 4. Customizing Garbage Collector

- **You can influence GC behavior using:**
  - **GC.Collect()**
  - **GC.SuppressFinalize()**
  - **GC.WaitForPendingFinalizers()**
- **Use carefully, not recommended in normal code.**

✅ 5. Unmanaged Resources

Examples:

- **Database connections**
- **File streams**
- **Network connections**

✅ 6. Scientific Notation / FPU

- **float and double use floating-point representation.**
- **They can store very large or very small numbers.**
- **Example:**
- **float f = 1.23e-4f;**
- **double d = 3.14e+6;**

✅ 7. When to Use Decimal?

Use decimal when precision matters, especially in:

- **Financial calculations**
- **Currency values**

✅ 8. Difference Between float, double, and decimal

Type	Precision	Use Case	Memory
float	7 digits	Scientific	4 bytes
double	15 digits	Most calculations	8 bytes
decimal	28 digits	Money/finance	16 bytes

✅ 9. Checked & Unchecked Blocks

Used to handle overflow in numeric operations.

```
checked
{
int x = int.MaxValue;
x++; // throws OverflowException
}

unchecked
{
int x = int.MaxValue;
x++; // no exception, wraps around
}
```

✅ 10. Parsing Null Values

Example:

```
int x = int.Parse(null); // throws
ArgumentNullException
```

✅ 11. Parse vs Convert (Performance)

- **Parse is faster but throws exception on invalid input.**
- **Convert is safer, handles null, but slightly slower.**

✅ 12. Stack vs Heap

- **Stack: fast, stores value types and references.**
- **Heap: slower, stores objects (reference types).**

✅ 13. Deallocation

- **Value types are deallocated automatically when the scope ends.**
- **Reference types are deallocated by GC when no longer referenced.**

✅ 14. Bitwise Operators

Used for bit-level operations:

- **& (AND)**
- **| (OR)**
- **^ (XOR)**

- ~ (NOT)
- << (left shift)
- >> (right shift)

Example:

```
int a = 5;   // 0101
int b = 3;   // 0011
int c = a & b; // 0001 = 1
```

Great question! Here's a simple and clear explanation:

✅ What does "C# is managed code" mean?

When we say **C# is managed code**, it means:

🛡️ The .NET runtime (CLR) controls your program

The Common Language Runtime (CLR) handles:

- Memory allocation
- Memory deallocation
- Garbage collection
- Type safety
- Security checks

So you don’t manually manage memory like in C++.

🔍 Why is this important?

✓ It prevents memory errors

Examples:

- **No manual free() or delete**
- **Less chance of memory leaks (but still possible if references stay)**

✓ It makes code safer and more reliable

Because the CLR checks and manages resources.

✅ What does “struct is considered like class before” mean?

In C#, structs and classes are both used to create custom data types.
So when people say “struct is like class before”, they usually mean:

✓ Both can contain:

- **Fields (variables)**
- **Methods**

- **Constructors**
- **Properties**

Example:

```
struct Student
{
public string Name;
public int Age;
}
```

This looks very similar to a class.

🔍 The main difference (important!)

Even though they look similar, they behave differently:

🔹 Struct (Value Type)

- **Stored in stack**
- **Copied when assigned**
- **Best for small data**

🔹 Class (Reference Type)

- **Stored in heap**
- **References are copied**

- **Best for larger objects**