**Part01🡪Questions**

**Question: What is the shortcut to comment and uncomment a selected block of code in Visual**

**Studio?**

**--Ctrl+k+C 🡪comment**

**--Ctrl+k+U🡪 uncomment**

**Question: Explain the difference between a runtime error and a logical error with examples.**

**Runtime Error:**

* **Definition: Errors that occur while the program is running, causing it to crash or behave abnormally.**
* **Cause: Typically due to invalid operations, such as dividing by zero or accessing an invalid index in an array.**
* **Example:**

**int a = 10; int b = 0;**

**Console.WriteLine(a / b); // Runtime error: Division by zero**

* **Result: Program crashes with an exception like System.DivideByZeroException.**

**Logical Error:**

**Logical Error:**

* **Definition: Errors in the logic of the program that produce incorrect results without crashing the program.**
* **Cause: Incorrect formulas, conditions, or implementation of logic.**
* **Example:**

**int a = 10; int b = 20; int sum = a - b; // Logical error: Should be addition, not subtraction**

**Console.WriteLine(sum); // Output: -10 (incorrect result)**

**Result: Program runs successfully but gives the wrong output**

**Question: Why is it important to follow naming conventions such as PascalCase in C#?**

**1-Improves Readability:**

**Consistent naming makes the code easier to understand and follow.**

**2-Enhances Maintainability:**

**Well-named variables and methods help developers quickly identify their purpose, reducing time spent on debugging or updates.**

**3-Promotes Team Collaboration:**

**Following standard conventions ensures that code written by different developers is consistent, making it easier for teams to work together.**

**4-Aligns with Industry Standards:**

**Conventions like PascalCase for class and method names match Microsoft's C# coding standards, ensuring professionalism.**

**5-Avoids Confusion:**

**Predictable naming reduces ambiguity, e.g., distinguishing between local variables (camelCase) and methods/classes (PascalCase).**

**6-Supports IDE Features:**

**Code editors like Visual Studio provide better intellisense and readability when conventions are followed.**

**Question: Explain the difference between value types and reference types in terms of memory**

**allocation.**

**value Types:**

* **Memory Allocation**: Value types are stored directly in the **stack** memory.
* **Data Copying**: When you assign a value type to another, a **copy** of the actual value is made. Each variable has its own independent copy of the data.
* **Examples**: int, float, bool, struct.
* **Behavior**: Since the data is stored directly, modifying one variable does not affect the other. Each value is independent.
* **Performance**: Typically faster because stack memory is more efficient than heap memory, and the stack does not require garbage collection.

**Reference Types:**

**Memory Allocation:** Reference types are stored in the heap memory, but the reference (or pointer) to that object is stored in the stack.

**Data Copying:** When you assign a reference type to another, only the reference (memory address) is copied, not the actual object. Both variables point to the same object in memory.

**Examples:** class, string, array, object.

**Behavior:** Since both variables point to the same object, modifying the object through one reference will reflect in all other references pointing to that object.

**Performance:** Slower compared to value types because the heap requires dynamic memory management and garbage collection.

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

**int a = 2, b = 7;**

**Console.WriteLine(a % b);**

**Output🡪 2 why?**

**a % b returns the remainder when a is divided by b. Since 2 divided by 7 gives a quotient of 0 and a remainder of 2, the result of 2 % 7 is 2.**

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

**1. && (Logical AND) Operator:**

* **Purpose: It is used in conditional expressions to evaluate Boolean values.**
* **Behavior:**
  + **Evaluates two Boolean expressions.**
  + **Short-circuiting: If the first operand is false, it does not evaluate the second operand, because the result of false && anything is always false. This makes it more efficient in certain cases.**
* **Operands: It operates on Boolean values (true or false).**
* **Use case: Typically used in if conditions or loops to combine multiple Boolean conditions.**

**2. & (Bitwise AND) Operator:**

**Purpose: It is used to perform a bitwise AND operation between two integral types (e.g., int, byte).**

**Behavior:**

**It operates on bitwise representations of numbers.**

**No short-circuiting: Both operands are always evaluated, regardless of the first operand’s value.**

**Results: It compares each corresponding pair of bits in two operands and returns 1 if both bits are 1, otherwise 0.**

**Operands: It operates on integer types or bit fields (e.g., int, long, byte).**

**Use case: Commonly used for operations involving binary data, such as masks or flags.**

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

**Explicit casting is required when converting a double to an int in C# because there is a potential for data loss and precision loss. Here's why:**

**1. Data Loss:**

**A double is a floating-point type that can hold decimal values, whereas an int is an integer type that can only hold whole numbers.**

**When converting from double (which can represent fractions) to int (which only represents whole numbers), any fractional part of the double will be lost.**

**2. Precision Loss:**

**The double type can represent much larger numbers with greater precision (up to 15–16 digits), while an int type has a smaller range (only up to 2 billion) and does not store decimal values.**

**For instance, if you try to cast a very large or small double value to an int, the result may lose significant digits or even result in an overflow if the double value exceeds the int range.**

**3. Explicit Casting:**

**C# does not perform an implicit conversion from double to int because this kind of conversion involves possible loss of data. To make it clear that you are aware of this potential loss, you must use explicit casting (i.e., (int)myDouble) to convert from double to int.**

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

**If the user enters invalid input (like a non-numeric string), a FormatException might occur when trying to convert it into an integer using int.Parse().**

**How to handle it:**

**by using a try-catch block to catch the error and show a friendly message**

**Question: Given the code below, what is the value of x after execution? Explain why**

**int x = 5;**

**int y = ++x + x++;**

**After the execution, the value of x is 7.**

**Explanation:**

**++x increments x to 6 before using it.**

**x++ uses x (which is 6) and then increments it to 7.**

**So, y becomes 12 (6 + 6), and the final value of x is 7.**

**Part02**

**2- what's the difference between compiled and interpreted languages and in this way what**

**about Csharp?**

1. **Compiled languages**:
   * Translated into machine code before execution.
   * Faster at runtime.
   * Example: C, C++.
2. **Interpreted languages**:
   * Executed line-by-line by an interpreter.
   * Slower at runtime.
   * Example: Python, JavaScript.
3. **C#**:
   * Combines both approaches.
   * **Compiled** into Intermediate Language (IL) using a compiler.
   * **Interpreted** by the Common Language Runtime (CLR) during execution.

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

**In C#, casting is the process of converting a value from one data type to another. There are different ways to perform casting in C#, and each has its specific use case. The main types of casting in C# are implicit casting, explicit casting, Convert, and Parse. Here's a comparison:**

**1. Implicit Casting:**

**Definition: Implicit casting happens automatically when the conversion is safe, and no data loss occurs.**

**When to Use: It is used when converting from a smaller type to a larger type (e.g., int to long), where there is no risk of losing data.**

**Example:**

**int x = 10;**

**long y = x; // Implicit casting from int to long**

**Behavior: The compiler automatically performs the casting, and no explicit syntax is needed.**

**Limitations: Only works when converting from a lower-range type to a higher-range type (e.g., int to long, float to double).**

**2. Explicit Casting:**

**Definition: Explicit casting requires the use of a casting operator ((type)) because the conversion might result in data loss or overflow, and the developer must indicate that they acknowledge the potential risks.**

**When to Use: It is used when converting from a larger type to a smaller type (e.g., double to int), or when there's a potential for data loss.**

**Example:**

**double a = 9.78;**

**int b = (int)a; // Explicit casting, truncates the decimal part**

**Console.WriteLine(b); // Output: 9**

**Behavior: If the conversion is invalid (such as trying to convert a large double to int), it throws an exception (e.g., InvalidCastException).**

**Limitations: If the value is out of range for the target type, it can lead to truncation or overflow.**

**3. Convert:**

**Definition: The Convert class provides static methods to convert between different types. It can handle conversions that are not directly possible through casting (e.g., converting string to int).**

**When to Use: It’s often used when you need to convert between types that may not be implicitly castable (e.g., string to int).**

**Example:**

**string str = "123";**

**int num = Convert.ToInt32(str); // Using Convert to convert string to int**

**Console.WriteLine(num); // Output: 123**

**Behavior: It handles null values and converts them to default values (e.g., Convert.ToInt32(null) results in 0).**

**Limitations: Throws exceptions if the conversion is invalid (e.g., FormatException if the string is not a valid number).**

**4. Parse:**

**Definition: The Parse method is used to convert a string representation of a type to the actual type. It is commonly used to convert string values to numeric types.**

**When to Use: It’s used when you want to explicitly convert a string to a numeric type (e.g., string to int).**

**Example:**

**string str = "456";**

**int num = int.Parse(str); // Using Parse to convert string to int**

**Console.WriteLine(num); // Output: 456**

**Behavior: It throws exceptions (FormatException, ArgumentNullException) if the string is not a valid representation of the target type (e.g., "abc" can't be parsed to an int).**

**Limitations: You must ensure that the string is in the correct format, or else it will throw an exception. It doesn't handle null values by default.**

4- self study report>>CLR>> Garbage collector

**common Language Runtime (CLR):**

* **Definition: CLR is the runtime environment in .NET that manages the execution of .NET programs.**
* **Responsibilities:**
  + **Memory management.**
  + **Code execution and compilation (via JIT).**
  + **Exception handling.**
  + **Type safety and security.**

**Garbage Collector (GC):**

* **Definition: A part of CLR that automatically manages memory by reclaiming unused objects in the heap.**
* **Key Features:**
  + **Frees developers from manual memory allocation/deallocation.**
  + **Prevents memory leaks and dangling pointers.**

**How It Works:**

1. **Allocation:**
   * **Objects are created in the managed heap.**
   * **Memory is allocated automatically.**
2. **Collection:**
   * **GC identifies unused objects (no references).**
   * **Reclaims memory occupied by these objects.**
   * **Compacting reduces heap fragmentation.**
3. **Generations:**
   * **Objects are divided into generations:**
     + **Gen 0: Short-lived objects (e.g., temporary variables).**
     + **Gen 1: Objects surviving one GC cycle.**
     + **Gen 2: Long-lived objects (e.g., static data).**
   * **Collecting Gen 0 is faster than Gen 2.**