**Solved Task05 C#**

**Part01**

**Problem 1**

**Question: What is the purpose of the finally block?**

The **purpose of the finally block** is to ensure that a certain section of code **always executes**, regardless of whether an exception is thrown or not in the try block.

**Key points:**

* Runs **after** the try (and any catch blocks) finish.
* Used for **cleanup tasks** such as closing files, releasing database connections, freeing resources, or logging actions.
* Executes whether:
  + No exception occurs.
  + An exception occurs and is caught.
  + An exception occurs and is **not** caught (it still runs before the program terminates, unless the process is forcefully stopped).

**Problem 2**

**Question: How does int.TryParse() improve program robustness compared to int.Parse()?**

int.TryParse() improves program robustness compared to int.Parse() because it does not throw an exception when the input is invalid. Instead, it returns false and outputs a default value (0), allowing the program to handle invalid input gracefully without crashing. This prevents unexpected runtime errors, improves performance by avoiding exception handling overhead, and provides better control over the program’s flow.

**Problem 3**

**Question: What exception occurs when trying to access Value on a null Nullable?**

Accessing Value on a null **Nullable<T>** causes an **InvalidOperationException** with the message:

"Nullable object must have a value."

This happens because Value tries to return the stored value, but there is none when the variable is null. The safe way is to check HasValue before accessing Value.

**Problem 4**

**Question: Why is it necessary to check array bounds before accessing elements?**

It is necessary to check array bounds before accessing elements to **prevent runtime exceptions** like IndexOutOfRangeException. Accessing an invalid index can cause the program to crash or behave unexpectedly, because arrays have fixed sizes and only store elements within their defined index range (from 0 to Length - 1). Bounds checking ensures you only work with valid data, improving **program safety, stability, and robustness**.

**Problem 5**

**Question: How is the GetLength(dimension) method used in multi-dimensional arrays?**

GetLength(dimension) is used to return the number of elements in a specific dimension of a multi-dimensional array. The dimension parameter is zero-based, where 0 refers to the first dimension (e.g., rows in a 2D array), 1 refers to the second dimension (e.g., columns), and so on. This method is useful for writing loops that adapt to arrays of different sizes without hard-coding their dimensions.

**Problem 6**

**Question: How does the memory allocation differ between jagged arrays and rectangular arrays?**

a rectangular array (e.g., int[,]) is stored as a single contiguous memory block where all rows have the same length. In contrast, a jagged array (e.g., int[][]) is an array of array references; each row is a separate object in memory and can have different lengths. This means rectangular arrays have better locality of reference, while jagged arrays offer more flexibility in row sizes but use extra memory for storing references to each row**.**

**Problem 7**

**What is the purpose of nullable reference types in C#?**

The purpose of nullable reference types in C# is to help developers write safer code by distinguishing between variables that can hold null and those that cannot. When enabled, the compiler warns you if you try to assign null to a non-nullable reference type or access a member of a nullable reference without checking for null first. This reduces the risk of NullReferenceException at runtime and encourages explicit handling of null values.

**Problem 8**

**What is the performance impact of boxing and unboxing in C#?**

Boxing and unboxing in C# can negatively impact performance because they involve extra memory allocation and type conversion at runtime. Boxing copies a value type into a newly allocated object on the heap, while unboxing extracts the value type from the object, which also requires type checking and casting. These operations are relatively expensive compared to working directly with value types and can lead to increased garbage collection if done frequently.

**Problem 9**

**Why must out parameters be initialized inside the method?**

out parameters must be initialized inside the method because they are meant to return data back to the caller. The compiler enforces this rule to ensure that the caller always receives a definite value before the method exits. Unlike ref parameters, out parameters do not require initialization before being passed in, but they must be assigned a value within the method to guarantee a valid result.

**Problem 10**

**Why must optional parameters always appear at the end of a method's parameter list?**

Optional parameters must appear at the end of a method’s parameter list because, during a method call, arguments are matched to parameters in order from left to right. If an optional parameter were placed before a required one, the compiler would not be able to correctly match arguments without ambiguity. Placing optional parameters last ensures that all required parameters are provided first, and any omitted arguments automatically use their default values.

**Problem 11**

**How does the null propagation operator prevent NullReferenceException?**

The **null propagation operator** (?.) prevents a NullReferenceException by first checking whether the object before it is null before attempting to access its members. If the object is null, the expression returns null instead of throwing an exception. This allows safe navigation of object members without explicit if checks for null values.

**Example:**

string name = person?.Name; // Returns null if person is null

**Problem 12**

**When is a switch expression preferred over a traditional if statement?**

A switch expression is preferred over a traditional if statement when you need to compare a single value against multiple possible constant patterns and produce a result in a concise, readable way. Switch expressions are more compact, eliminate repetitive syntax, and clearly express mapping from one value to another, making the code easier to maintain. In contrast, if statements are more suitable for complex, non-constant conditions or when multiple variables and ranges need to be evaluated**.**

**Problem 13**

**What are the limitations of the params keyword in method definitions?**

The **params** keyword in C# has several limitations:

1. **Only one params parameter** is allowed in a method, and it must be the **last parameter** in the parameter list.
2. All arguments passed through params must be of the same specified type (or implicitly convertible to it).
3. You cannot use ref or out modifiers with a params parameter.
4. Overuse of params can lead to extra array allocations at runtime, which may impact performance in performance-critical code.