Question1:

The finally block is used to ensure that certain code **always executes**, regardless of whether an exception occurs or not.

It runs after the try and catch blocks:

If no exception is thrown → finally still executes.

If an exception is thrown and caught → finally executes after the catch.

If an exception is thrown but not caught → finally executes before the program terminates.

Question2:

int.Parse()

- Converts a string to an integer.
- If the input is invalid (like "abc" or empty), it throws an exception (FormatException).
- This can cause the program to crash if not handled properly.

- int.TryParse()
- Attempts to convert a string to an integer.
- Returns a boolean (true if successful, false if not).
- Does not throw exceptions on invalid input.
- Allows the program to handle errors gracefully without breaking.

Why It Improves Robustness

- Prevents runtime crashes due to invalid user input.
- Encourages defensive programming by validating input before using it.
- Makes the program more reliable when dealing with unpredictable or messy input (like user typing letters instead of numbers).
- Provides a safer way to enforce rules (like requiring positive integers or Y > 1) without relying on exception handling.

Question3:

When you attempt to access the .Value property of a Nullable that is null, the program throws an InvalidOperationException.

Question4:

Checking array bounds is necessary because arrays in C# (and most programming languages) have a fixed size. Each element is stored at a specific index, and valid indices range from 0 to Length - 1.

If you try to access an index outside this range:

- The runtime throws an IndexOutOfRangeException.
- This can cause the program to crash if not handled properly.

Question5:

The GetLength(dimension) method in C# is used to determine the size of a specific dimension in a multi-dimensional array.

Question6:

Rectangular arrays ([,])

Memory is allocated as a single contiguous block.

Every row has the same number of columns, so the compiler knows the exact layout in memory.

Jagged arrays ([][])

Memory is allocated as an array of references, where each row points to a separate array.

Each row can have a different length, so memory is not contiguous.

Question7:

Nullable reference types were introduced in C# to help developers write safer code by explicitly handling the possibility of null values.

Key Purposes

- Prevent NullReferenceException:
  One of the most common runtime errors in C# occurs when you try to use a reference that is null. Nullable reference types make this risk visible at compile time.
- Explicit Nullability:
  By declaring string?, you indicate that the variable may hold null. By declaring string (non-nullable), you promise it will never be null. This makes your intent clear.
- Compiler Warnings:
  The compiler warns you if you try to assign null to a non-

nullable reference or if you dereference something that might be null. This shifts error detection from runtime to compile time.

- Safer APIs:
  When libraries use nullable reference types, consumers can immediately see which parameters or return values might be null, reducing misuse.

Question8:

Boxing and unboxing have a negative performance impact compared to working directly with value types because they involve extra processing and memory operations.

- Boxing
  - Occurs when a value type (like int) is converted into an object.
  - The runtime allocates memory on the heap and copies the value there.
  - This is more expensive than keeping the value on the stack.
- Unboxing
- Occurs when you convert the object back into a value type.

- Requires type checking and then copying the value back from the heap to the stack.
- If the cast is invalid, it throws an InvalidCastException.

Performance Impact

- Extra memory allocation: Boxing creates a new object on the heap.
- Garbage collection overhead: More heap allocations mean more work for the garbage collector.
- Slower execution: Both boxing and unboxing add runtime overhead compared to direct value type usage.
- Best practice: Avoid unnecessary boxing/unboxing by using generics or working directly with value types.

Question9:

In C#, out parameters are designed to return values from a method. Because of this:

- Mandatory assignment: The compiler enforces that every out parameter must be assigned a value inside the method before the method returns.

- Caller expectation: The caller doesn’t need to initialize out parameters before passing them in, but after the method call, they are guaranteed to hold a value.
- Safety: This rule prevents the caller from accidentally using an uninitialized variable, ensuring that the method always provides a valid result.

Question10:

Optional parameters in C# must be placed at the end of a method’s parameter list because of how the compiler resolves method calls:

- Default values: Optional parameters have default values that the compiler substitutes if the caller doesn’t provide them.
- Ambiguity prevention: If optional parameters were in the middle of the list, the compiler wouldn’t know whether missing arguments should be skipped or matched to subsequent parameters, leading to confusion.
- Clear call syntax: Placing them at the end ensures that required parameters are always provided first, and optional ones can be omitted without breaking the method call.

Question11:

The null propagation operator (?.) in C# allows you to safely access members of an object that might be null.

- Normal access: If you try to access a property or method on a null object, you get a NullReferenceException.
- With ?.: Instead of throwing an exception, the expression simply evaluates to null if the object is null.

Question12:

A switch expression is preferred over traditional if-else statements when:

- Multiple discrete values: You're mapping one input to different outputs based on distinct cases (e.g., days of the week, menu options, status codes).
- Conciseness: Switch expressions are shorter and more readable compared to long chains of if-else.
- Direct value mapping: Unlike if-else, switch expressions return a value directly, making them ideal for assignments.

- Maintainability: Easier to update or extend with new cases without cluttering the code.
- Clarity: The intent is clearer — you're explicitly saying "map this input to that output" rather than writing multiple conditional checks.

Question13:

The params keyword in C# allows you to pass a variable number of arguments to a method, but it comes with some important limitations:

- Only one params parameter per method: You cannot define multiple params parameters in the same method signature.
- Must be the last parameter: The params parameter must appear at the end of the parameter list, because the compiler needs to know where the variable-length arguments stop.
- Fixed type: All arguments passed through params must be of the same type (or convertible to that type). You can't mix different types unless you use object[].
- No overload resolution priority: If there are multiple overloads, the compiler may prefer a non-params

overload, which can sometimes lead to unexpected behavior.

- Array creation overhead: When you pass individual values, the compiler creates an array behind the scenes, which can add slight overhead compared to passing a pre-existing array.