m: Write a program that: o Reads two integers from the user and divides them. o Catches DivideByZeroException and displays an appropriate message. o Uses a finally block to print "Operation complete" regardless of success or failure. • Question: What is the purpose of the finally block?

Here's a simple C# example of your program:

```
using System;

class Program
{
    static void Main()
    {
        try
        {
            Console.Write("Enter the first integer: ");
            int num1 = int.Parse(Console.ReadLine());

            Console.Write("Enter the second integer: ");
            int num2 = int.Parse(Console.ReadLine());

            int result = num1 / num2;
            Console.WriteLine("Result: " + result);
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Error: Cannot divide by zero.");
        }
        finally
        {
            Console.WriteLine("Operation complete");
        }
    }
}
```

Purpose of the `finally` block:
The `finally` block contains code that will *always* execute after the `try` and `catch` sections, **regardless** of whether an exception occurred.
It's typically used for cleanup actions — such as closing files, releasing resources, or displaying a message like here — that should happen no matter what.

In other words: even if your code crashes or runs perfectly, `finally` still shows up like the one reliable friend who never cancels plans.

```
public static void TestDefensiveCode()

{

  int x, y;



  // Get X

  while (true)

  {
```

```
    Console.Write("Enter a positive integer for X: ");
    if (int.TryParse(Console.ReadLine(), out x) && x > 0)
      break;
    Console.WriteLine("Invalid input. Please enter a positive integer.");
  }

  // Get Y
  while (true)
  {
    Console.Write("Enter a positive integer for Y (greater than 1): ");
    if (int.TryParse(Console.ReadLine(), out y) && y > 1)
      break;
    Console.WriteLine("Invalid input. Please enter a positive integer greater than 1.");
  }

  Console.WriteLine($"You entered X = {x} and Y = {y}");
}
```

How int.TryParse() improves robustness compared to int.Parse():

int.Parse() will throw a FormatException if the input is not a valid integer (e.g., the user types "abc"), forcing you to handle the exception with try/catch.

int.TryParse() returns true or false instead of throwing an exception. This lets you check validity without risking a crash or entering exception-handling overhead.

In short:

int.Parse() → Crashes if invalid input (unless you catch exceptions).

int.TryParse() → Gracefully fails by returning false, making it safer for user-driven input and improving program stability.

Nullable integer + null-coalescing operator

```
using System;
```

```
class Program
{
    static void Main()
    {
        int? number = null;
        int result = number ?? 10; // default to 10 if null
        Console.WriteLine($"Value: {result}");

        Console.WriteLine($"HasValue: {number.HasValue}");
        // Accessing Value when null will throw
        // Console.WriteLine(number.Value); // Uncomment → exception
    }
}
```

Question: *What exception occurs when trying to access `Value` on a null `Nullable<T>`?*
Answer: `InvalidOperationException` — occurs when accessing `.Value` while `HasValue` is `false`.

---

2. 1D array + IndexOutOfRangeException

```
class ArrayTest
{
    static void Main()
    {
        int[] arr = new int[5];
        try
        {
            arr[5] = 10; // invalid index (0-4 valid)
        }
        catch (IndexOutOfRangeException ex)
        {
            Console.WriteLine("Error: " + ex.Message);
        }
    }
}
```

Question: *Why check array bounds?*
Answer: To prevent `IndexOutOfRangeException` and avoid reading/writing invalid memory, which could crash the program.

---

3. 3x3 array sum by row & column

```
class MatrixSum
{
    static void Main()
    {
        int[,] matrix = new int[3,3];
        for (int i = 0; i < matrix.GetLength(0); i++)
        {
            for (int j = 0; j < matrix.GetLength(1); j++)
            {
                Console.Write($"Enter value for [{i},{j}]: ");
```

```
                matrix[i, j] = int.Parse(Console.ReadLine());
            }
        }

        for (int i = 0; i < matrix.GetLength(0); i++)
        {
            int rowSum = 0;
            for (int j = 0; j < matrix.GetLength(1); j++)
                rowSum += matrix[i, j];
            Console.WriteLine($"Row {i} sum: {rowSum}");
        }

        for (int j = 0; j < matrix.GetLength(1); j++)
        {
            int colSum = 0;
            for (int i = 0; i < matrix.GetLength(0); i++)
                colSum += matrix[i, j];
            Console.WriteLine($"Column {j} sum: {colSum}");
        }
    }
}
```

Question: *How is `GetLength(dimension)` used?*
Answer: `GetLength(0)` returns number of rows, `GetLength(1)` returns number of columns in a multidimensional array.

---

4. Jagged array

```
class JaggedExample
{
    static void Main()
    {
        int[][] jagged = new int[3][];
        jagged[0] = new int[2];
        jagged[1] = new int[3];
        jagged[2] = new int[4];

        for (int i = 0; i < jagged.Length; i++)
        {
            for (int j = 0; j < jagged[i].Length; j++)
            {
                Console.Write($"Enter value for row {i}, col {j}: ");
                jagged[i][j] = int.Parse(Console.ReadLine());
            }
        }

        foreach (var row in jagged)
        {
            Console.WriteLine(string.Join(" ", row));
        }
    }
}
```

Question: *How does memory allocation differ?*
Answer:

- **Rectangular arrays** allocate a single contiguous block for all elements.
- **Jagged arrays** allocate separate memory blocks for each row (rows can have different lengths).

---

5. Nullable reference types

```
#nullable enable
class NullableRefDemo
{
    static void Main()
    {
        string? name = null;
        Console.Write("Enter your name (or press Enter to skip): ");
        var input = Console.ReadLine();
        if (!string.IsNullOrWhiteSpace(input))
            name = input;

        Console.WriteLine($"Hello, {name!}"); // null-forgiveness operator
    }
}
```

Question: *Purpose of nullable reference types?*
Answer: They help detect potential `null` usage at compile time, reducing `NullReferenceException` at runtime.

---

6. Boxing & unboxing

```
class BoxingExample
{
    static void Main()
    {
        int num = 42;
        object boxed = num; // boxing
        try
        {
            int unboxed = (int)boxed; // unboxing OK
            Console.WriteLine(unboxed);

            string invalid = (string)boxed; // invalid unboxing
        }
        catch (InvalidCastException ex)
        {
            Console.WriteLine("Error: " + ex.Message);
        }
    }
}
```

Question: *Performance impact?*
Answer: Boxing/unboxing involves heap allocation and copying, which is slower than working directly with value types.

---

7. Method with out parameters

```
class OutParamExample
{
    static void SumAndMultiply(int a, int b, out int sum, out int product)
    {
        sum = a + b;
        product = a * b;
    }

    static void Main()
    {
        SumAndMultiply(3, 4, out int s, out int p);
        Console.WriteLine($"Sum: {s}, Product: {p}");
    }
}
```

Question: *Why must `out` parameters be initialized inside the method?*
Answer: Because they're meant to output a value — the caller should not rely on any pre-existing value.

---

8. Optional parameters + named arguments

```
class OptionalParams
{
    static void Repeat(string message, int times = 5)
    {
        for (int i = 0; i < times; i++)
            Console.WriteLine(message);
    }

    static void Main()
    {
        Repeat(times: 3, message: "Hello"); // named parameters
    }
}
```

Question: *Why must optional parameters be last?*
Answer: To avoid ambiguity in method calls and allow omitted arguments to match correctly.

---

9. Nullable array + null-propagation

```
class NullPropagationDemo
{
    static void Main()
    {
        int[]? arr = null;
```

```
        Console.WriteLine($"Array length: {arr?.Length}"); // prints
nothing instead of throwing
    }
}
```

Question: *How does null-propagation prevent `NullReferenceException`?*
Answer: If the object before `?.` is `null`, the expression short-circuits and returns `null` without evaluating further.

---

10. Switch expression

```
class SwitchExpressionDemo
{
    static void Main()
    {
        Console.Write("Enter a day of the week: ");
        string day = Console.ReadLine();

        int dayNumber = day.ToLower() switch
        {
            "monday" => 1,
            "tuesday" => 2,
            "wednesday" => 3,
            "thursday" => 4,
            "friday" => 5,
            "saturday" => 6,
            "sunday" => 7,
            _ => 0
        };

        Console.WriteLine($"Day number: {dayNumber}");
    }
}
```

Question: *When is a switch expression preferred?*
Answer: When mapping discrete values to results in a concise, readable way, especially for immutable pattern matching without complex branching logic

Here's the `SumArray` example:

```
using System;

class Program
{
    // Method that accepts variable number of integers
    static int SumArray(params int[] numbers)
    {
        int sum = 0;
        foreach (int num in numbers)
            sum += num;
        return sum;
    }

    static void Main()
    {
```

```
        // Call with individual values
        int total1 = SumArray(1, 2, 3, 4, 5);
        Console.WriteLine($"Sum (individual values): {total1}");

        // Call with an array
        int[] values = { 10, 20, 30 };
        int total2 = SumArray(values);
        Console.WriteLine($"Sum (array): {total2}");
    }
}
```

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

1. **Only one `params` parameter** is allowed per method signature.
2. **It must be the last parameter** in the parameter list.
3. All arguments passed via `params` must be of the declared element type (or implicitly convertible to it).
4. Overuse can lead to ambiguity if there are multiple overloads accepting arrays.

---

5. Print Numbers in a Range

```
using System;

class Program
{
    static void Main()
    {
        Console.Write("Enter a positive integer: ");
        int n = int.Parse(Console.ReadLine());

        for (int i = 1; i <= n; i++)
        {
            Console.Write(i);
            if (i < n) Console.Write(", ");
        }
    }
}
```

---

2. Multiplication Table up to 12

```
using System;

class Program
{
    static void Main()
    {
        Console.Write("Enter a number: ");
        int num = int.Parse(Console.ReadLine());

        for (int i = 1; i <= 12; i++)
        {
            Console.Write(num * i);
            if (i < 12) Console.Write(", ");
        }
    }
}
```

---

3. List Even Numbers

```
using System;

class Program
{
    static void Main()
    {
        Console.Write("Enter a number: ");
        int n = int.Parse(Console.ReadLine());

        for (int i = 2; i <= n; i += 2)
        {
            Console.Write(i);
            if (i + 2 <= n) Console.Write(", ");
        }
    }
}
```

---

4. Compute Exponentiation

```
using System;

class Program
{
    static void Main()
    {
        Console.Write("Enter base: ");
        int baseNum = int.Parse(Console.ReadLine());
        Console.Write("Enter exponent: ");
        int exp = int.Parse(Console.ReadLine());

        int result = 1;
        for (int i = 0; i < exp; i++)
            result *= baseNum;

        Console.WriteLine(result);
    }
}
```

---

5. Reverse a Text String

```
using System;

class Program
{
    static void Main()
    {
        Console.Write("Enter a string: ");
        string input = Console.ReadLine();

        char[] arr = input.ToCharArray();
        Array.Reverse(arr);
        Console.WriteLine(new string(arr));
    }
}
```

---

6. Reverse an Integer Value

```
using System;

class Program
{
```

```
    static void Main()
    {
        Console.Write("Enter an integer: ");
        int num = int.Parse(Console.ReadLine());

        string reversed = new
  string(num.ToString().ToCharArray().Reverse().ToArray());
        Console.WriteLine(reversed);
    }
}
```

---

7. Longest Distance Between Matching Elements

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        Console.Write("Enter array elements separated by spaces:
  ");
        int[] arr =
  Console.ReadLine().Split().Select(int.Parse).ToArray();

        int maxDistance = -1;
        for (int i = 0; i < arr.Length; i++)
        {
            for (int j = arr.Length - 1; j > i; j--)
            {
                if (arr[i] == arr[j])
                {
                    int dist = j - i - 1;
                    if (dist > maxDistance) maxDistance = dist;
                    break;
                }
            }
        }

        Console.WriteLine($"Longest distance: {maxDistance}");
    }
}
```

---

8. Reverse Words in a Sentence

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        Console.Write("Enter a sentence: ");
        string sentence = Console.ReadLine();

        string result = string.Join(" ", sentence.Split('
  ').Reverse());
        Console.WriteLine(result);
    }
}
```

الموضوع باختصار — Boxing & Unboxing في C#:
تقدر تعتبره زي لما يكون معاك حاجة صغيرة في جيبك (stack)، وفجأة تقول: "لا، أنا هحطها في شنطة كبيرة (heap) عشان أتعامل معاها بشكل عام أكتر.
بس المشكلة إنك كل مرة تعمل كده، بتاخد وقت ومجهود.

Boxing:

بتحصل لما تاخد Value Type (زي int, bool, double) وتحطه جوه Reference Type (زي object أو dynamic أو حتى عناصر ArrayList).

القيمة بتتنقل وتتبنسخ من stack لـ heap.

مثال:
int points = 50;
object data = points; // Boxing: القيمة اتنقلت للـ heap
Unboxing:

العكس، لما تستخرج القيمة الأصلية من الـ object وتعملها cast للنوع اللي كانت عليه.

لازم النوع يكون مطابق 100% وإلا هتاخد InvalidCastException.

مثال:
object data = 50;
int points = (int)data; // Unboxing: رجعناها للـ stack
العيوب والمشاكل:

كل عملية Boxing/Unboxing = أداء أبطأ وميموري أكتر → نسخ + نقل بيانات.

لو بتحصل داخل loop أو دالة بتشتغل كتير، هتحس بالبطء.

مثال يوضح الخطورة:

```
ArrayList list = new ArrayList();

for (int i = 0; i < 100000; i++)
{
 list.Add(i); // كل مرة Boxing هنا بيحصل
}

for (int i = 0; i < list.Count; i++)
{
 int number = (int)list[i]; // كل مرة Unboxing هنا بيحصل
}
```

الحل: استخدم List<int> بدل ArrayList → مفيش Boxing/Unboxing.

الخلاصه :

استخدم Generics (List<T>, Dictionary<K,V>) لتفادي Boxing.

لو بتتعامل مع Structs كتير، خلي بالك من التحويلات التلقائية.

راقب الأداء لو الكود فيه تحويلات كتير بين Value و Reference Types.