Part01

Problem1

```
using System;
class Program
{
   static void Main()
   {
      try
      {
         Console.Write("Enter first number: ");
         int num1 = int.Parse(Console.ReadLine());

         Console.Write("Enter second number: ");
         int num2 = int.Parse(Console.ReadLine());
         int result = num1 / num2;

         Console.WriteLine("Result: " + result);
      }
      catch (DivideByZeroException)
      {
         Console.WriteLine("Error: You cannot divide by zero!");
      }
      finally
      {
         Console.WriteLine("Operation complete");
      }
```

```
    }
}
```

Question1:

الغرض من الـ finally block

هو تنفيذ كود معين **مهما حصل** — سواء حصل خطأ (Exception) أو لأ، فهي تشتغل في كل الحالات.

مثال: تنظيف موارد، إغلاق ملف، أو طباعة رسالة انتهاء العملية.

Problem2

```
using System;

class Program
{
    static void TestDefensiveCode()
    {
        int x, y;

        do
        {
            Console.Write("Enter a positive integer for X: ");
        } while (!(int.TryParse(Console.ReadLine(), out x) && x > 0));

        do
        {
            Console.Write("Enter a positive integer greater than 1 for Y: ");
        } while (!(int.TryParse(Console.ReadLine(), out y) && y > 1));

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

```
    }

    static void Main()
    {
        TestDefensiveCode();
    }
}
```

Question2:

كيف يحسن int.TryParse() من قوة البرنامج مقارنة بـ int.Parse()؟

- وبيوقف البرنامج إلا إذا **Exception** بيحاول يحول النص لرقم، **ولو الإدخال مش رقم بيعمل** int.Parse() تعاملنا مع الخطأ بـ try/catch.
- int.TryParse() بيرجع **true أو false** بدل ما يرمي Exception، وده يخلي البرنامج:
    - أكتر أمان (ما بيقعش فجأة).
    - أسرع في المعالجة (مفيش تكلفة معالجة استثناءات).
    - أسهل في التحقق من صحة المدخلات قبل استخدامها.

Problem3

```
using System;

class Program
{
    static void Main()
    {
        int? number = null; // متغير Nullable بدون قيمة

        int result = number ?? 100;
```

```
        Console.WriteLine("Result: " + result); // 100 هتكون



        Console.WriteLine("HasValue: " + number.HasValue); // False


        try
        {
            Console.WriteLine("Value: " + number.Value); // هيعمل Exception
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine("Error: " + ex.Message);
        }
    }
}
```

Question3:

الاستثناء اللي بيحصل لو حاولنا نستخدم Value على Nullable<T> وهو null:
هو **InvalidOperationException** برسالة غالبًا:

"Nullable object must have a value."

Problem4

```
using System;


class Program
{
    static void Main()
```

```
  {
    int[] numbers = new int[5] { 1, 2, 3, 4, 5 };

    try
    {
      Console.WriteLine(numbers[10]);
    }
    catch (IndexOutOfRangeException ex)
    {
      Console.WriteLine("Error: " + ex.Message);
    }
  }
}
```

Question4:

ليه لازم نتحقق من حدود المصفوفة قبل الوصول للعناصر؟

- وده يوقف البرنامج لو ما **IndexOutOfRangeException** لأن الوصول لفهرس خارج الحدود بيعمل اتعالجش.
- التحقق من الحدود يحافظ على **استقرار البرنامج** ويمنع **أخطاء منطقية** أو **انهيار** أثناء التشغيل.
- كمان ده بيخلي البرنامج يتعامل مع الأخطاء **بطريقة متوقعة وآمنة** بدل ما ينهار فجأة.

Problem5

```
using System;

class Program
{
  static void Main()
```

```
{
    int[,] matrix = new int[3, 3];

    for (int i = 0; i < matrix.GetLength(0); i++) // الصفوف
    {
        for (int j = 0; j < matrix.GetLength(1); j++) // الأعمدة
        {
            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($"Sum of row {i}: {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($"Sum of column {j}: {colSum}");
    }
  }
}
```

Question5:

ستخدام GetLength(dimension) في المصفوفات متعددة الأبعاد:

- GetLength(0) → يعطي عدد الصفوف (Rows).
- GetLength(1) → يعطي عدد الأعمدة (Columns).
- البارامیتر dimension يبدأ من 0 لأول بعد، و1 للبعد الثاني، وهكذا.
- لجعل الكود ديناميكي ويتكيف مع GetLength() للأبعاد، نستخدم (Hardcode) الفائدة: بدل ما نكتب أرقام ثابتة أي حجم مصفوفة

Problem6

```
using System;

class Program
{
  static void Main()
  {
    // إنشاء jagged array بثلاث صفوف
    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}, column {j}: ");
                jagged[i][j] = int.Parse(Console.ReadLine());
            }
        }

        Console.WriteLine("\nJagged Array Elements:");
        for (int i = 0; i < jagged.Length; i++)
        {
            for (int j = 0; j < jagged[i].Length; j++)
            {
                Console.Write(jagged[i][j] + " ");
            }
            Console.WriteLine();
        }
    }
}
```

Question6:

- **Jagged** → مرونة أكبر، أحجام صفوف مختلفة، لكن أقل كفاءة في التخزين المتجاور.
- **Rectangular** → أداء أسرع في الوصول، لكن كل الصفوف لازم نفس الحجم.

Problem7

```
using System;


class Program
{
   static void Main()
   {
      string? name; // Nullable string


      Console.Write("Enter your name (leave empty for null): ");
      string input = Console.ReadLine();


      if (string.IsNullOrWhiteSpace(input))
         name = null;
      else
         name = input;


      Console.WriteLine("Name length: " + name!.Length);
   }
}
```

Question7:

الغرض من nullable reference types في C#:

- وقت **الـ** null عن طريق تفعيل فحص الـ **NullReferenceException** هي ميزة بتساعدك **تتجنب أخطاء** بدل ما تكتشفه أثناء التشغيل **compile**.
- بدون null وهو (reference type) يحذرك لو فيه احتمال تستخدم متغير مرجعي (compiler) بتخلي المترجم ما تتحقق منه.
- بتفرق بين:

- string → غير قابلة تكون null (لازم دايمًا تحتوي على قيمة).
- string? → ممكن تكون null.

Problem8

```
using System;

class Program
{
   static void Main()
   {
      int num = 42; // Value type

      object boxed = num;
      Console.WriteLine("Boxed value: " + boxed);

      try
      {
         int unboxed = (int)boxed;
         Console.WriteLine("Unboxed value: " + unboxed);

         // Unboxing الغلط (هنا هيعمل InvalidCastException)
         double wrongUnbox = (double)boxed;
      }
      catch (InvalidCastException ex)
      {
         Console.WriteLine("Error: " + ex.Message);
```

```
    }
  }
}
```

Question8:

تأثير الـ Boxing والـ Unboxing على الأداء:

- **Boxing → بيعمل نسخة جديدة من الـ value type في الـ heap بدل الـ stack وده بياخد وقت ومساحة إضافية.**
- **Unboxing → بيعمل عملية تحويل وفك التغليف، وبرضه بياخد وقت إضافي.**
- **التكرار الكتير لعمليات boxing/unboxing بيأثر سلبي على الأداء والذاكرة.**
- **الأفضل نتجنبهم باستخدام Generics أو أنواع مناسبة من البداية.**

Problm9

```
using System;

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

  static void Main()
  {
    int x = 5, y = 3;
    int total, multi;
```

```
    SumAndMultiply(x, y, out total, out multi);


    Console.WriteLine($"Sum: {total}");

    Console.WriteLine($"Product: {multi}");

  }

}
```

Question9:

داخل الميثود؟ (initialization) يتم تهيئتهم out parameters ليه لازم الـ

- **معناها إن المتغير ده هيستلم القيمة من الميثود، مش هيدخل بقيمة موجودة outلأن الكلمة.**
- **قبل ما تخرج من الميثود علشان تتأكد إن out parameterبيجبرك تدي قيمة لكل C# في compiler الـ المتغير هيكون فيه قيمة صالحة بعد الاستدعاء.**
- **"Use of unassigned out parameter"لو ما عملتش كده، الكومبايلر هيدي خطأ.**

Problem10

```
using System;


class Program

{


  static void PrintText(string text, int count = 5)

  {

    for (int i = 0; i < count; i++)

    {

      Console.WriteLine(text);

    }
```

```
    }

    static void Main()
    {
        PrintText("Hello");

        Console.WriteLine();

        PrintText(count: 3, text: "C# is awesome!");
    }
}
```

Question10:

ليه لازم الباراميترات الاختيارية تبقى في آخر قائمة الباراميترات؟

- **لأن لو كان فيه باراميتر اختياري في النص، واستدعيت الميثود بدون تحديد قيم لكل الباراميترات اللي بعده، المترجم مش هيقدر يميز القيم رايحة لأنهي باراميتر.**
- **بوضعهم في الآخر، أي قيم مش بتُمرر هتاخد القيمة الافتراضية مباشرة، وده بيمنع الغموض في الاستدعاء**

Problem11

```
using System;

class Program
{
    static void Main()
    {
        int[]? numbers = null; // مصفوفة Nullable
```

```
    int? length = numbers?.Length;


    Console.WriteLine("Array length: " + (length.HasValue ? length.Value.ToString() :
"null"));


    numbers = new int[] { 1, 2, 3 };
    Console.WriteLine("Array length: " + numbers?.Length);
  }
}
```

Question11:

إزاي null propagation operator يمنع NullReferenceException؟

- **لما تستخدم ?. المترجم بيفحص لو المتغير مش null، ساعتها بيكمل الوصول للخاصية أو الميثود.**
- **لو المتغير null النتيجة كلها بتبقى null، من غير ما يرمي NullReferenceException.**
- **قبل كل وصول للخصائص if (x != null) ده بيخلي الكود آمن وأقصر بدل ما نكتب شرط.**

Problem12

```
using System;


class Program
{
  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
    };

    if (dayNumber > 0)
      Console.WriteLine($"The number for {day} is {dayNumber}");
    else
      Console.WriteLine("Invalid day entered.");
  }
}
```

Question12:

متى نفضل استخدام switch expression على if التقليدي؟

- **لما يكون عندك قيمة واحدة بتقارنها مع احتمالات متعددة، وعاوز تكتب الكود بشكل أقصر وأوضح.**
- **بيكون أفضل في switch expression:**
  - **تحويل قيمة لقيمة تانية (mapping).**
  - **تقليل التكرار مقارنة بـ if-else if-else.**
  - **كتابة الكود بأسلوب functional أكتر، خصوصًا لما يكون النتيجة مباشرة بدون منطق معقد.**
- **أما if التقليدي، فهو أنسب لو عندك شروط معقدة أو أكتر من متغير بتتحقق منه.**

Problem13

```
using System;
class Program
{
  static int SumArray(params int[] numbers)
  {
    int sum = 0;
    foreach (int num in numbers)
    {
      sum += num;
    }
    return sum;
  }
  static void Main()
  {
    int total1 = SumArray(1, 2, 3, 4, 5);
    Console.WriteLine($"مجموع القيم الفردية: {total1}");
    int[] arr = { 10, 20, 30 };
    int total2 = SumArray(arr);
    Console.WriteLine($"مجموع عناصر المصفوفة: {total2}");
  }
}
```

Question13:

- **يخليك تمرر عدد غير محدد من القيم أو تمرر مصفوفة مباشرة** params int[] numbers.
- **لو بتمرر قيم فردية، المترجم هيحوّلهم أوتوماتيك لمصفوفة.**
- **لو بتمرر مصفوفة موجودة، هتتقبل بشكل مباشر**

Part02

1:

```
using System;

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

      for (int i = 1; i <= number; i++)
      {
         Console.Write(i);
         if (i < number)
         {
            Console.Write(", "); // فاصلة بين الأرقام
         }
      }
   }
}
```

2:

```
using System;
```

```
class Program
{
   static void Main()
   {

      Console.Write("Enter a number: ");
      int number = int.Parse(Console.ReadLine());

      for (int i = 1; i <= 12; i++)
      {
         Console.WriteLine($"{number} x {i} = {number * i}");
      }
   }
}
```

3:

```
using System;

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

```
    for (int i = 1; i <= number; i++)
    {
      if (i % 2 == 0)
      {
        Console.Write(i);

        if (i < number && i + 2 <= number)
        {
          Console.Write(", ");
        }
      }
    }
  }
}
```

4:

```
using System;

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

    Console.Write("Enter the exponent: ");
```

```
        int exponent = int.Parse(Console.ReadLine());

        double result = Math.Pow(baseNumber, exponent);

        Console.WriteLine($"{baseNumber} ^ {exponent} = {result}");
    }
}
```

5:

```
using System;

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

        char[] charArray = input.ToCharArray(); // نحول النص لمصفوفة حروف
        Array.Reverse(charArray); // نعكس ترتيب الحروف

        string reversed = new string(charArray);
        Console.WriteLine("Reversed string: " + reversed);
    }
}
```

6:

```
using System;

class Program
{
   static void Main()
   {
      Console.Write("Enter an integer: ");
      string number = Console.ReadLine();

      char[] arr = number.ToCharArray();
      Array.Reverse(arr);

      string reversed = new string(arr);

      Console.WriteLine("Reversed number: " + reversed);
   }
}
```

7:

```
using System;

class Program
{
   static void Main()
   {
```

```
Console.Write("Enter the size of the array: ");
int N = int.Parse(Console.ReadLine());

int[] arr = new int[N];

Console.WriteLine("Enter the array elements:");
for (int i = 0; i < N; i++)
{
   arr[i] = int.Parse(Console.ReadLine());
}

int longestDistance = 0;

for (int i = 0; i < N; i++)
{
   for (int j = N - 1; j > i; j--)
   {
      if (arr[i] == arr[j])
      {
         int distance = j - i - 1; // طرح 1 عشان نحسب الخلايا بينهم
         if (distance > longestDistance)
         {
            longestDistance = distance;
         }
         break;
      }
```

```
        }
      }

      Console.WriteLine("The longest distance is: " + longestDistance);
   }
}
```

8:

```
using System;

class Program
{
   static void Main()
   {
      string sentence = Console.ReadLine();
      Console.WriteLine(string.Join(" ", sentence.Split(' ')[^1..^0]));
   }
}
```

Wafaa Mohammed · أنت
الآن

الموضوع اللي بيأكل من سرعة برنامجك وانت مش واخد بالك!
يا سيدي، لو إنت مبرمج أو حتى لسه بتتعلم، أكيد سمعت كلمة Boxing و Unboxing.

وأنا متأكد إنك أول ما سمعتهم عقلك راح على البوكسات والهدايا.
وده في الحقيقة مش بعيد عن المعنى البرمجي!
تخيل إنك بتشتغل في شركة برمجة، وبتكتب كود، وفجأة الكود بيبدأ يبطئ من غير سبب واضح... هنا بقى الموضوع ممكن يكون ليه علاقة بالحاجتين دول، حتى لو إنت مش واخد بالك.

الفرق بينهم بيبان بسيط، لكن في الأداء (Performance) الموضوع ممكن يبقى كارثي لو ما فهمتوش صح.

تعالى ناخدهم واحدة واحدة

📦 Boxing – لما تحط الحاجة في بوكس

تخيل معايا إنك معاك مفتاح صغير (ده الـ Value Type زي int، double في C#).
إنت عايز تبعته لصاحبك، بس هو مش بيستقبل غير الحاجات اللي جاية جوه بوكس شيك (وده الـ Object).
هتعمل إيه؟
تحط المفتاح جوه البوكس وتقفله... أهو بالظبط ده اللي بيحصل في Boxing.

في البرمجة:
لما بتحول value type لـ object type، الكمبيوتر بياخد القيمة دي ويحطها في "بوكس" في الذاكرة (Heap) بدل ما كانت في مكانها العادي (Stack).

📌 مثال:

```
int number = 10;
object obj = number; // حصل Boxing
```

هنا الـ 10 كانت رقم عادي (Value Type) واتحطت في "علبة" (Object).

📦 Unboxing – لما تفتح البوكس وتطلع الحاجة

بعد ما بعت المفتاح في البوكس، صاحبك قرر يفتحه ويطلع الحاجة اللي جواه عشان يستخدمها زي ما هي.
ده بالظبط اللي بيحصل في Unboxing.

في البرمجة:
لما ترجع تاخد الـ object type وتحوله تاني لـ value type، إنت كده بتعمل Unboxing.

📌 مثال:

```
object obj = 10;     // Boxing
int number = (int)obj; // Unboxing
```

هنا إحنا فتحنا البوكس (Object) ورجعنا الحاجة لشكلها الأصلي.

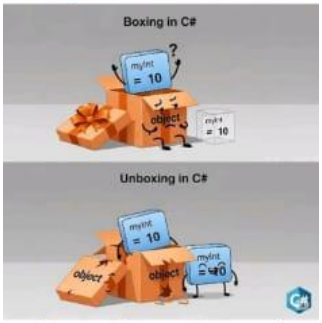
⚠️ ليه الموضوع مهم؟
كل مرة تعمل Boxing أو Unboxing، البرنامج بيستهلك وقت ومساحة زيادة.

البيانات بدل ما تتخزن في الـ Stack (السريع)، بتروح على الـ Heap (الأبطأ).

عمليات التحويل دي ممكن تبطأ برنامجك جدًا لو اتكررت كتير في لوب أو كود بيشتغل بشكل متكرر.

يعني الموضوع مش بس "تحويل"، ده كمان له تأثير مباشر على الأداء.

#CSharp #DotNet #Boxing #Unboxing
#PerformanceTips #CleanCode #ProgrammingTips
#TechInsights #LearnToCode



إعجاب · تعليق · إعادة نشر · إرسال

إضافة تعليق...

Part03 Bonus

يعني إيهCasting Operator ؟

تخيل معايا عندك رقم بس مكتوب ككلمة (string) زي "123"، وإنت محتاجه يبقى رقم صحيح (int) عشان تعمل عليه عمليات حسابية. هنا بييجي دور الـCasting ، اللي هو التحويل دا.

في ++C فيه حاجة اسمها Casting Operator Function، ودي Function انت اللي بتكتبها بإيدك جوه الكلاس، وظيفتها إنها تقول للكومبايلر: "لما تحب تحول الكائن دا (object) لنوع تاني، استخدم الطريقة اللي أنا حاططها."

💡مثال

تخيل عندك كلاس اسمه Meterبيمثل مسافة بالأمتار، وعايز لما أحوله لـ doubleيرجعلي القيمة بالأمتار، أو لما أحوله لـ intيرجعلي القيمة مقربة.

إنت ممكن تعمل كده:

```
class Meter {
double meters;
public:
Meter(double m) : meters(m) {}

// Casting Operator to double
operator double() const {
return meters;
}

// Casting Operator to int
operator int() const {
return (int)meters;
}
};
```

```
int main() {
Meter m(5.75);

double d = m; // بيحول باستخدام operator double()
int i = m;    // بيحول باستخدام operator int()

std::cout << "Double: " << d << std::endl;
std::cout << "Int: " << i << std::endl;
}
```

🛠الفكرة

- إنت بتكتب operator نوع_البيانات () كـ Function.
- الكومبايلر لما يلاقيك بتحول الكائن لنوع معين، هيروح ينفذ الـ operator اللي انت كاتبه.
- الفنكشن دي ملهاش اسم، اسمها بيتحدد بالنوع اللي بتحوله ليه.

أول حاجة Upcasting :

- التعريف :لما تاخد Object من Class فرعي (Child) وتتعامل معاه كأنه من نوع Class أب.(Parent)
- دا بيحصل بشكل طبيعي (Implicit) من غير ما تكتب أي Casting ، لأن الكومبايلر عارف إن الـ Child هو نسخة مطورة من الـ Parent ، فمش هيعترض.
- زي إنك تكون مدير وتشتغل وظيفة موظف، المدير عارف كل شغل الموظف، فالموضوع سهل.

📌مثال:

```
class Animal {
public:
void speak() { std::cout << "Animal sound\n"; }
};
```

```
class Dog : public Animal {
public:
void bark() { std::cout << "Woof!\n"; }
};

int main() {
Dog myDog;
Animal* a = &myDog; // Upcasting
//      a->speak();   شغال
//      // a->bark();   مش هينفع، لإنه شايفه Animal بس
}
```

💡ليه بنعمله؟

- عشان نستخدم الـ Polymorphism (الدوال الافتراضية virtual functions).
- بيخلي الكود أكثر مرونة.

⬇️تاني حاجةDowncasting :

- التعريف :العكس، إنك تاخد Object من Parent Classوتحوله لـ Child Class.
- هنا لازم تكون عارف ومتأكد إن الكائن أصلاً معمول كـChild ، وإلا هتلاقيRuntime Error.
- بيحتاج Explicit Casting(إنت اللي بتكتبه) عشان الكومبايلر مايفترضش غلط.

📌مثال:

```
Animal* a = new Dog(); // Upcasting حصل هنا
Dog* d = (Dog*)a;     // Downcasting
//      d->bark();شغال
```

لو الكائن مش من نوع Dog فعليًا، هتدخل في مشاكل.
عشان كده في C++ ممكن تستخدم dynamic_castللتأكد:

```
Dog* d = dynamic_cast<Dog*>(a);
```

```
if (d) {
d->bark();
} else {
std::cout << "Not a Dog!\n";
}
```

الفكرة الأساسية

لما تيجي تبعت متغير لفنكشن في #C أو ++C أو غيرها، فيه طريقتين مشهورتين:

1. **By Value: الفنكشن بياخد نسخة من المتغير ويشتغل عليها، والأصل ما بيتأثرش.**
2. **By Reference: الفنكشن بيشتغل على نفس المتغير الأصلي، وأي تغيير بيحصل عليه بيأثر برا الفنكشن كمان.**

Ref Type Passing معناه إنك بتبعت المتغير للفنكشن بالمرجع (Reference) بدل ما تبعته بالنسخة.

باستخدام ref

```
using System;

class Program {
static void ChangeNumber(ref int num) {
num = 100; // هيغير الأصل
}

static void Main() {
int x = 5;
ChangeNumber(ref x);
Console.WriteLine(x); // النتيجة: 100
}
```

}

💡هنا كلمة ref معناها إن الفنكشن شايف نفس العنوان في الذاكرة، فلو غيرت القيمه جوه، هتتغير برا.

📌لو ما استخدمتش ref (By Value)

static void ChangeNumber(int num) {

num = 100; // هيغير النسخة

}

هنا القيمة بره الفنكشن هتفضل زي ما هي، لأنك بتعدل نسخة مش الأصل.