.NET Framework 4.7 and C# 8.0

Lesson 15 : C# 7.0 New Features

New Features of C# 7.0

- Out Variable Enhancements
- Pattern Matching
- Tuples
- Local or Nested Functions
- ref Returns and Locals
- Expression Bodied Members (some more expression bodied members in C# 7)
  - Expression bodied constructor
  - Expression bodied destructor
  - Expression bodied getters
  - Expression bodied setters
- Throw Expressions
- Literal Improvements
- Generalized async Return Types

Out Variable Enhancements

- We usually use out variables when with TryParse method.
- In earlier C# versions, before we use TryParse, we had to define those variables.
- Ex -

```
//Till C#-6.0
string rollNoData = "1001";
int rollNo; //Need to declare the variable first before we use
if (int.TryParse(rollNoData, out rollNo))
{
    WriteLine($"Conversion Successful!");
}
WriteLine($"rollNo is {rollNo}");
```

Out Variable Enhancements (Cont…)

- That is not the case anymore with C# 7.0+. We can now directly declare it where we are passing it as a parameter.
- Ex –

```
//Now in C#-7.0+
string rollNoData = "10011";
//No need to declare variable before we use, you can directly use it
if (int.TryParse(rollNoData, out int rollNo))
{
    WriteLine($"Conversion Successful!");
}
//Variable declared inside if, can also be used outside if scope
WriteLine($"rollNo is {rollNo}");
```

- Though variable is declared inside if, it can be accessible outside if scope.
- You can also use var instead int while declaring variable.

Pattern Matching - Switch Statements

- Pattern Matching was used for evaluating a switch case with the const pattern till C#-6.0.
- But now in C#-7.0, we can use *type* & *var* pattern as well.
- Ex -

```
Student student = new Student("Kamlesh", "Jadhaw");
switch (student)
{
    //This is Constant pattern
    case null: Console.WriteLine("It's Constant pattern"); break;

    //This is a type pattern
    case Student s when s.FirstName.StartsWith("M"): Console.WriteLine(s.FirstName);
break;

    //This is a var pattern with the type Student
    case var x: Console.WriteLine(x?.GetType().Name); break;
}
```

Tuples

- Many times in code you need to return value(s) from method but options available in older versions of C# are less than optimal.
  - Creating Classes: Code overhead for a type whose purpose is just to temporarily group a few values.
  - Out Parameters: Can get quite ugly & don’t works with async methods.
  - Anonymous Types & dynamic keyword: Performance overhead & no static type checking support.
  - System.Tuple<...>: Verbose to use & requires tuple object allocation as its reference type.
- To do better at this, C# 7.0 adds ***tuple***
  - Tuples are very useful to replace hash table or dictionary easily.
  - Additionally you can use it instead of List where you store multiple values at single position.
  - .NET already has a Tuple type but it is a reference type and that leads to performance issue
  - But C# 7.0 bring a Tuple with value type which is faster in performance and a mutable type.
- [Note: Need to first install System.ValueTuple through NuGet Package to support this feature]

Tuples

- Ex –

```
private static (double min, double max, double avg) GetResult(List<double> numbers)
{
    return (numbers.Min(), numbers.Max(), numbers.Average());
}

static void Main(string[] args)
{
    List<double> numbers = new List<double> { 52, 45, 120, 56, 98, 304, 20, 69 };

    var res = GetResult(numbers);
    Console.WriteLine($"Lowest: {res.min}, Highest:{res.max}, Average:{res.avg}");
    //OR
    (double l, double h, double a) = GetResult(numbers);
    Console.WriteLine($"Lowest: {l}, Highest:{h}, Average:{a}");
}
```

Local or Nested Functions

- Sometimes a specific function of a class could use some code splitting and that particular code makes sense only inside of that method.
- Ex -

```
public int Fib(int num)
{
    if (num < 0)
        throw new Exception("Number should be >= 0");
    return FibMemo(num);

    int FibMemo(int n)
    {
        if(n <= 1)
            return n;
        return FibMemo(n - 1) + FibMemo(n - 2);
    }
}
```

Some More Expression Bodied Members

- C#-6.0 introduced Expression Bodied Methods & Properties, but other members were not supported.
- C# 7.0 adds accessors, constructors and finalizers to the list of things that can have expression bodies.
- Ex -

```
class Product
{
   Dictionary<int, decimal> productPriceList = new Dictionary<int, decimal>();
   public int ProductId { get; set; } = 1;
   public decimal Price //Expression bodied getter-setter
   {
      get => productPriceList[ProductId];
      set => productPriceList[ProductId] = value;
   }
   public Product() => Price = 4000; //Expression bodied constructor
   ~Product() => Console.WriteLine("Expression bodied destructor");
}
```

Throw Expression (Throwing Exception from Expression)

- We can throw an exception directly through expression.
- Below code snippet can directly throw exception from return statement.

```
static void Main(string[] args)
{
    var a = Divide(10, 0);
}
public static double Divide(int x, int y)
{
    //Directly throwing DivideByZeroException from expression
    return y != 0 ? x % y : throw new DivideByZeroException();
}
```

Literal Improvements

- C#-7.0 introduced some literals to improve the readability in code.
- Digit Separator: Allows _ to occur as a digit separator inside number literals.
- Hexadecimal Literal: Specify hexadecimal number directly to assign the values to variable
- Binary Literals: Specify bit patterns directly instead of having to know hexadecimal notation by heart
- Ex -

```
var lit1 = 478_1254_3698_44;        //Digit Separator
var lit2 = 0Xa3_7e;                 //Hexadecimal literal
var lit3 = 1100_1011_0100_1010_1001;//Binary literal
Console.WriteLine($"lit1: {lit1}, lit2: {lit2}, lit3: {lit3}");
```

Generalized async Return Types

- Up until now, an async method had to return Task, Task<T>, or void.
- However, returning Task or Task<T> can create performance bottlenecks as the reference type needs allocating.
- For C#7, we can now return other types from async methods, including the new ValueTask<T>, enabling us to have better control over these performance concerns.
- Ex -

```
async Task<int> LoadCache()
{  //simulating async work
    await Task.Delay(100);
    cacheResult = 100;
    cache = true;
    return cacheResult;
}
```

```
bool cache = false;
int cacheResult;
public ValueTask<int> CachedFunc()
{
    return (cache) ? new
ValueTask<int>(cacheResult) : new
ValueTask<int>(LoadCache());
}
```

Summary

- In this lesson, you learned following New Features of C#-7.0
  - Out Variable Enhancements
  - Pattern Matching using Is Expression & Switch case
  - System.ValueTuple – New way to use Tuple
  - Local or Nested Functions
  - ref Returns and Locals
  - Some more Expression Bodied Members, like constructor, destructor, getters & setters
  - Throwing Exception from Expression
  - Improving the readability in code using Literal

Review Question

1. How using out keyword is enhanced in C#-7.0 ?
2. Which was the only pattern allowed in switch case till C#-6.0?
3. What can be use instead of List to store multiple values at single position?
4. List out the members which also support Expression bodied feature in C#-7.0
5. Which character in C#-7.0 allows to occur as a digit separator inside number literals?