.NET Framework 4.6 and C# 7.0

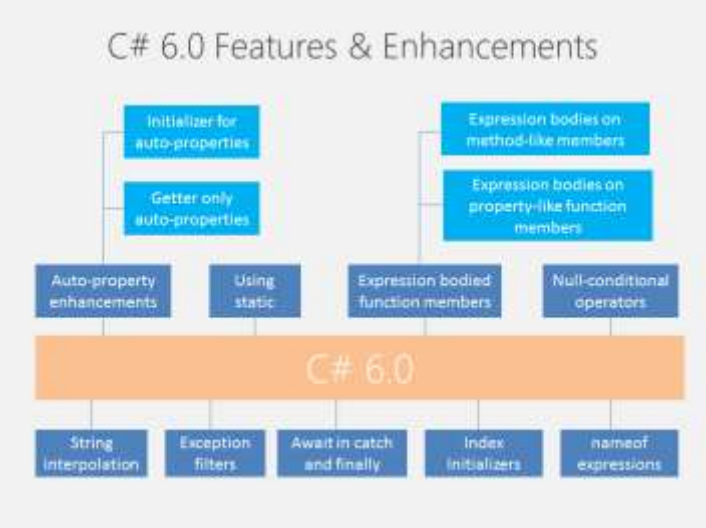
Lesson 14 : C# 6.0 Features

Capgemini

New Features of C# 6.0

- Using static
- String interpolation
- Dictionary Initializers
- Auto property Initializers & Getter only auto properties
- nameOf Operator
- Await in catch/finally
- Null conditional operator and Null Propogation
- Expression bodied members
- Using static with Extension Methods
- Exception filters

New Features of C# 6.0



Static Types as using

- As with the namespaces, we can include a static class in the using statement similar to a namespace.
- Makes our code less cluttered and will reduce duplications

```
Using System;
using static System.Console;/* Magic happens here */
namespace NewInCSharp
{
   class Program
   {
      static void Main(string[] args)
      {
            WriteLine("C# 6.0 New Features");
            ReadKey();
      }
   }
}
```

We are all quite familiar with this notion of accessing static class members using the qualification first.
It is not required now. Importing static using can save the day.

For example, before C# 6.0, we had to access ReadKey(), WriteLine() methods of the Console class by explicitly defining the static class qualifier.

Old Way

```
using System;
namespace NewInCSharp
{
   class Program
   {
      static void Main(string[] args)
      {
         MySelf.WhoAmI();
         Console.ReadKey();
      }
   }
   static class MySelf
```

```
    {
        public static void WhoAmI()
        {
            Console.WriteLine("I'm Fizz!");
        }
    }
}
```

String Interpolation

- Placeholders in strings are not needed now to replace them with real values.
- Using string interpolation, you can now directly write your arguments instead of referring them with placeholders inside a string.

- C# 6 has a new feature called string interpolation using which you can now directly write your arguments instead of referring them with placeholders inside a string.
- You can also do whatever you would have done previously with String.Format() function.

```
using System;
using System.Collections.Generic;
using static System.Console;

namespace NewInCSharp
{
   class Program
   {
      private static void Main(string[] args)
      {
         string name = “Sheldon";
         string planet = “Earth";

         WriteLine("{0} is actually named after {1}", planet, name);// Old Way
```

```
            WriteLine($"{planet} is actually named after {name}");

            ReadLine();
        }
    }
}
```

Dictionary Initializers

- C# 6.0 changed the way you can initialize Dictionary.
- Previously on C# 5, you would have to initialize the Dictionary with this type of syntax, {"Key", "Value"}.
- Now in C# 6, you can just place the key between two curly brackets ["Key"] and then set the value of the key ["Key"] = "value"; just like you set the value for other types of variable.

```
using System.Collections.Generic;
using static System.Console;
namespace NewInCSharp
{
   class Program
   {
      private static void Main(string[] args)
      {
         Dictionary<string, string=""> dictionaryObj = new Dictionary<string,
string="">()
         {
            /*{"Name", "Fizzy"},
            {"Planet", "Kepler-452b"}*/
                  // The new and friendly way
                     ["Name"] = "Fizzy",
            ["Planet"] = "Kepler-452b"


         };
```

```
foreach (KeyValuePair<string, string=""> keyValuePair in dictionaryObj)
        {
           WriteLine(keyValuePair.Key + ": " + keyValuePair.Value + "\n");
        }
        ReadLine();
    }
  }
}
```

Auto-Property Initializers

- A new concept of initializing class properties inline rather than initializing them within the type's constructor.

```
/* Property with inline initialization */
  public decimal Salary { get; set; } = 10000;
```

- Another handy technique is, if you want to make the setter of a property private to block users from setting value in the property by an instance, you can just declare a getter only property.

```
/* Getter only property with inline initialization */
  public string Name { get; } = "Rahul Shah"
```

Old Way

```
using static System.Console;
namespace NewInCSharp
{
   class Program
   {
      static void Main(string[] args) {
         Employee employee = new Employee();
         WriteLine("Name: " +
         employee.Name + "\nSalary: " + employee.Salary);
         ReadKey();
      }
      public class Employee{
         public string Name { get; set; }
         public decimal Salary { get; set; }
         public Employee(){
                  /* Initializing property through constructor */
                  Name = "Rahul Shah";
```

```
                Salary = 10000;
            }
        }
    }
}
```

nameof Expression

- Showing a specific type name with an error message can be a quick way to find the code block where the exception just occurred. But we should also consider refactoring issues.
- We cannot just simply append a hard coded type name string with an error message and show it to the user because the type name can be changed anytime while refactoring but hard coded string won't change accordingly. So C# 6.0 introduced this concept of nameof expression.

```
int? number = null;
if (number == null)
{
        throw new Exception(nameof(number) + " is null");
}
```

In enterprise level applications, lines of code run like a mad horse. So there is no means of avoiding exception handling where it is necessary.

```
using System;
using static System.Console;
namespace NewInCSharp
{
   class Program
   {
      private static void Main(string[] args)
      {  try
         {          CallSomething();                  }
         catch (Exception exception)
         {          WriteLine(exception.Message);          }
      }
      private static void CallSomething()
      {  int? x = null;
         if (x == null)
         {
```

```
            throw new Exception("x is null");     /* x is the type name. What if someone
changes the type name from x to i? The exception below would be inappropriate. */
          }
        }
    }
}
```

Await in catch/finally Block

- The most awaited feature in C# where you can write asynchronous code inside catch and finally block is now in 6.0

```
private async Task FileNotFound()  {}
private async Task ExitProgram()  {}

catch {       await FileNotFound();       }
finally {     await ExitProgram();        }
```

```
public async void FileReadOperation()
      {
         try
         {
            StreamReader sr = File.OpenText("D:\\data.txt");
            WriteLine(" The first line of the file is: \{sr.ReadLine()}");
            sr.Close();
         }
         catch { await FileNotFound(); }
         finally { await ExitProgram(); }
      }
      private async Task FileNotFound()
      {
         WriteLine(" File not found. Please check the file name and file
location.");
      }
private async Task ExitProgram()
      {
         WriteLine("\n Press any key to exit");
```

```
}
```

Null Conditional Operator & Null Propagation

- A new notion of null conditional operator where you can remove declaring a conditional branch to check to see if an instance of an object is null or not with this new ?. ?? null conditional operator syntax.
- The ?. is used to check if an instance is null or not, if it's not null then execute the code after ?. but if it is not, then execute code after ??.

var result = A?.B??C // If A is null B is assigned to result else C will be assigned.

```
using System;
using static System.Console;
namespace NewInCSharp
{
   class Program
   {
      private static void Main(string[] args)
      {
         SuperHero hero = new SuperHero();
         if (hero.SuperPower == String.Empty)
         {
            hero = null;
         }
         /* old syntax of checking if an instance is null or not */
         WriteLine(hero != null ? hero.SuperPower : "You aint a super hero.");

       /* New null conditional operator */
         WriteLine(hero?.SuperPower ?? "You aint a super hero.");
         ReadLine();
```

```
        }
    }
    public class SuperHero
    {
        public string SuperPower { get; set; } = "";
    }
}
```

Expression Bodied Function & Property

- You can now write functions and computed properties like lamda expressions to save extra headache of defining function and property statement block.
- Just use the lambda operator => and then start writing your code that goes into your function/property body.

- Here is a simple example:

```
/* Expression bodied function */
private static double AddNumbers(double x, double y) => x + y;

/* Expression bodied computed property */
public string FullName => FirstName + " " + LastName;
```

Static Using with Extension Methods

- Related to the using static.
- You can explicitly write the static type name like in C# 5 or previous versions or you can simply execute the function on one of its accepted types.
- In C# 6.0, importing static types and accessing extension methods of them will give you an error. The following code will generate the "The name [name of the function] does not exist in the current context".

Those of you who are wondering that if I import static types in my namespace, how would I deal with the extension methods. Well, extension methods are also static methods.
Well, you can explicitly write the static type name like in C# 5 or previous versions or you can simply execute the function on one of its accepted types.
Let me make it clear by an example:

```
using System;
namespace NewInCSharp
{   class Program
   {       private static void Main(string[] args)
          {         Shape shape = new Shape();
                    ShapeUtility.GenerateRandomSides(shape);
                    Console.WriteLine(ShapeUtility.IsPolygon(shape));
          }
   }
   public class Shape    {       public int Sides { get; set; }    }
public static class ShapeUtility
   {       public static bool IsPolygon(this Shape shape)
```

```
        {           return shape.Sides >= 3;       }
         public static void GenerateRandomSides(Shape shape)
        {            Random random = new Random();
                      shape.Sides = random.Next(1, 6);
        }
    }
}
```

Static Using with Extension Methods

```
private static void Main(string[] args)
{
        Shape shape = new Shape();
        GenerateRandomSides(shape);
        WriteLine(IsPolygon(shape));

//So to get over this issue, you can simply modify your code like this:
 /* You can write WriteLine(ShapeUtility.IsPolygon(shape));.But here I'm executing extension method on shape type,  that's why they are called extension methods since there are just a extension of your type */

        WriteLine(shape.IsPolygon());
        ReadLine();
}
```

Exception Filtering

- Exception filtering is nothing but some condition attached to the catch block. Execution of the catch block depends on this condition. Let me give you a simple example.

```
catch (Exception ex)
{
        if(ex.Message.Equals("400"))
            Write("Bad Request");
        else if (ex.Message.Equals("401"))
            Write("Unauthorized");
}
```

```
using System;
using static System.Console;
namespace NewInCSharp
{   class Program
    {   private static void Main(string[] args)
        {
            Random random = new Random();
            var randomExceptions = random.Next(400, 405);
            WriteLine("Generated exception: " + randomExceptions);
            Write("Exception type: ");
            try {       throw new Exception(randomExceptions.ToString());       }
            catch (Exception ex)
            {   if(ex.Message.Equals("400"))
                    Write("Bad Request");
                else if (ex.Message.Equals("401"))
                    Write("Unauthorized");
                else if (ex.Message.Equals("402"))
                    Write("Payment Required");
                else if (ex.Message.Equals("403"))
```

```
                Write("Forbidden");
            else if (ex.Message.Equals("404"))
                Write("Not Found");
        }
    }
  }
}
```