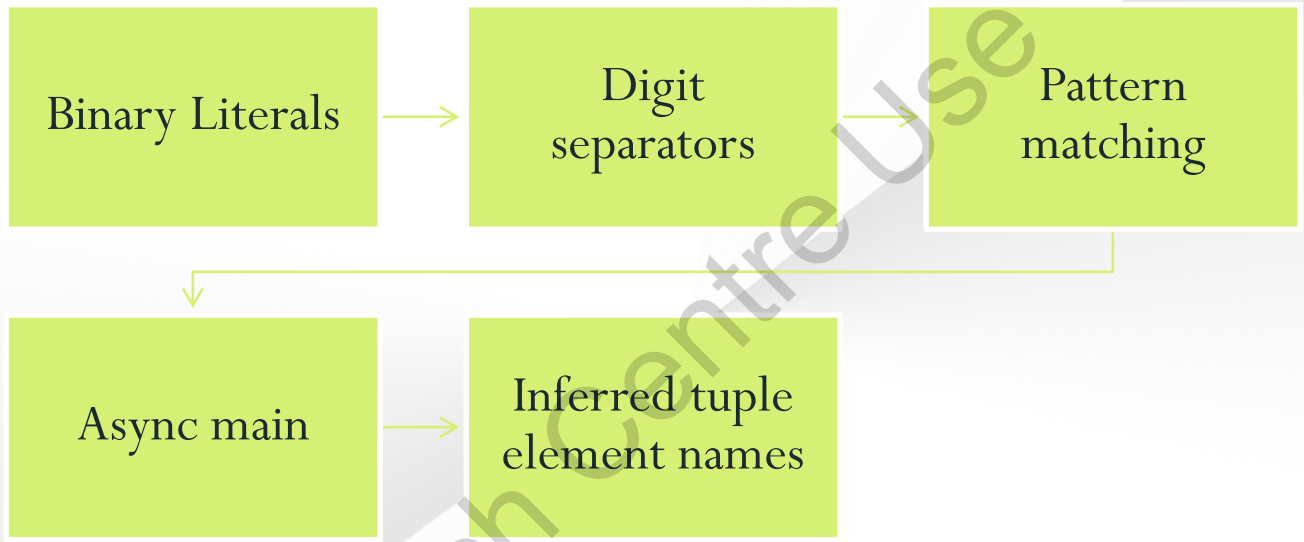
Session 13: More Features of C#

Objectives

- Describe ref returns and ref locals
- Explain improvements made to out variables, tuples, asynchronous Main(), and throw expressions
- Describe new expression-bodied members
- Outline Pattern matching enhancements
- List fit and finish features in C# 9.0
- Identify performance and interop improvements in C# 9.0
- Explain support for code generator

Additional Features of C#

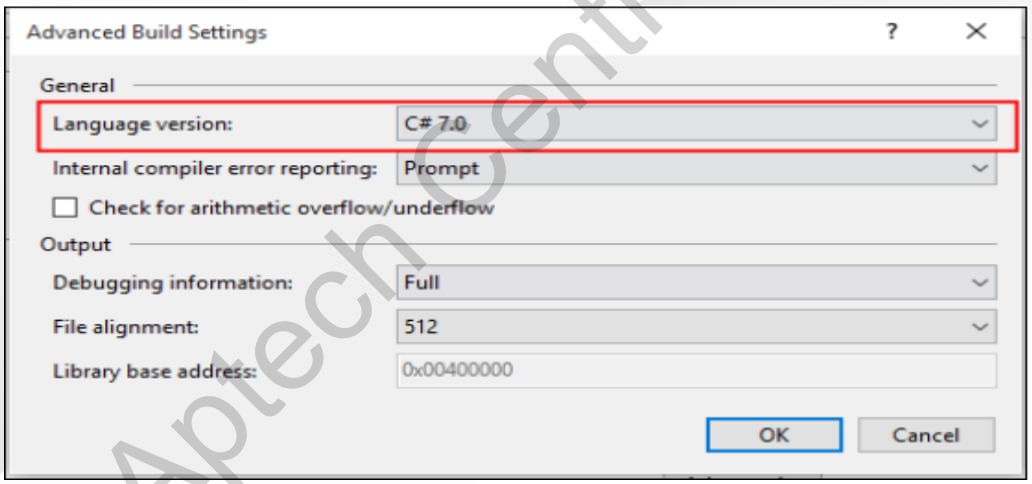
- From C# 7.0 onwards,



- All minor and major features of recent versions C# work together to ensure clearer, more efficient, and more productive code.

More Ways of Selecting the Language Version

- Ability to make the compiler work as per the preferred language version.
- Developer is required to modify project's configuration settings for specifying intended language version.



Changing Language Version Through Project Properties Window

Using Light Bulb Icon

- The IDE automatically discovers that user intends to use the new language version. Accordingly, it then prompts to update the project.
- To use the light bulb, the user can try to specify a default literal in the code. This triggers a light bulb icon for fixing the code.

Light Bulb Icon

Modifying the .csproj File

- Developers can update `.csproj` file to manually change language version. To do so, they must set `</LangVersion>` element in the first `<PropertyGroup>` element of the file.
- Following code shows modified file with lines in bold indicating the change:

Snippet

```
<PropertyGroup>
<OutputType>Exe</OutputType>
<RootNamespace>ConsoleApp1</RootNamespace>
<AssemblyName>ConsoleApp1</AssemblyName>
<TargetFrameworkVersion>v4.7</TargetFrameworkVersion>
<FileAlignment>512</FileAlignment>
<AutoGenerateBindingRedirects>true</AutoGenerateBindingRedirects>
<LangVersion>latest</LangVersion>
</PropertyGroup>
```

- Following are the values that this tag can hold:

default	The `default` value indicates the latest major version, which is C# 9.0 at present.
latest	The `latest` value indicates the latest minor version, which is C# 9.0.
ISO-1/ISO-2 3, 4, 5, 6, 7, 7.1, 8, 9	Other values are explicit options that the Visual Studio shall use if specified, even if a newer version is present. For instance, specifying 6 indicates the use of C# 6 even C# 7.0 is available.

Creating the .props File

- Use the file with the `.props` extension and is in the form, `Directory.Build.props`.
- The repository's root and has all common properties that are applicable to all Visual Studio projects.
- Allows making customizations to projects that are within a directory. For example, a user can set common properties such as the company, origin, creator, and publisher.
- Code to specify intended language version:

Snippet

```
<Project>
<PropertyGroup>
<LangVersion>latest</LangVersion>
</PropertyGroup>
</Project>
```

- While running MSBuild, `Microsoft.Common.props` looks for the `.props'` directory structure. If it is found, MSBuild imports the property and merges the .props file with `.csproj` file.

- This is how Visual Studio automatically imports the `LangVersion` of `.props` file for all the existing and new projects.

ref Returns, ref Locals, and Improved out Variables

- In C#, a method can contain a parameter that a developer can pass by reference using the `ref` keyword.
- Mandatory to specify this keyword in the calling method as well as in the definition of the called method.
- Let's assume that a method has three parameters, of which the last one is passed by reference.
- In previous versions of C#, this is done using the `ref` keyword. `Main()` is the caller of this method.
- However, prior to invoking this method, the Main() method initializes the third parameter. Thus, it is mandatory to initialize a `ref` variable prior to specifying it as a parameter.

ref Returns 1-2

- In C# 7.1 onwards, the `ref` keyword comes with some more abilities.
- A developer can use it to return values passed by reference.
- The keyword is now capable of storing values passed by reference in local variables.
- Following code shows a simple example:

Snippet

```
class Program {
 static void Main(string[] args) {
 string[] writers = {"Emy George", "Lee Mein", "John Wash", "Sicily
    Wang"};
 ref string writer2 = ref new Program().FindWriter(1, writers);
 Console.WriteLine("Original writer:{0}", writer2);
 Console.WriteLine();
 writer2 = "Johan Muller";
 Console.WriteLine("Replaced writer:{0}", writers[1]);
 Console.ReadKey();
 }
 public ref string FindWriter(int num, string[] names) {
        if (names.Length > 0)
              return ref names[num];
        throw new IndexOutOfRangeException($"{nameof(num)}
        unavailable.");
 } }
```

ref Returns 2-2

- In the previous code:

However, prior to invoking this method, the Main() method initializes the third parameter. Thus, it is mandatory to initialize a `ref` variable prior to specifying it as a parameter.

When invoked, in the `Main()`, the `ref` keyword is again used to store the value in `writer2`.

In this way, developers can use the references later. This improvement has made it easier to deal with requirements in which references should be returned in a specified sequence for replacing one or more values.

ref Locals 1-3

- In C# 7.1 onwards, the `ref` keyword comes with some more abilities. A developer can use it to return values passed by reference.
- The keyword is also now capable of storing values passed by reference in local variables.
- This modification to the `ref` keyword makes it easy to manage situations wherein references must be returned to make inline replacements.
- Following code demonstrate the use of ref locals:

Snippet

```
class Program {
 static void Main(string[] args) {
 string[] writers = {"Emy George", "Lee Mein", "John Wash","Sicily Wang"};
 ref string writer2 = ref new Program().FindWriter(1, writers);
 Console.WriteLine("Original writer:{0}", writer2);
 Console.WriteLine();
 writer2 = "Johan Muller";
 Console.WriteLine("Replaced writer:{0}", writers[1]);
 Console.ReadKey();
 }
 public ref string FindWriter(int num, string[] names) {
 if (names.Length > 0)
 return ref names[num];
 throw new IndexOutOfRangeException($"{nameof(num)} unavailable.");
 }
}
```

ref Locals 2-3

- In the previous code:
  - `ref` is specified just prior to the char type, `SeekCharRef()`. This denotes that the method shall return the variable's reference.
  - When it is invoked in the `Main()` method, the returned reference is stored in `cref`, which is a ref local variable.
  - In the output of this code, the first character would be replaced by `p`.
- The ref returns and ref locals features are useful for replacing placeholders or reading from large data structures.
- For example, an application can store its data in an array of structs. This is usually done for getting rid of the unwanted pauses during garbage collection.
- For the purpose of reading or changing the stored data, the related methods can then, return a reference to this array

ref Locals 3-3

- **ref locals** and **ref returns** are useful features, but there are two restrictions levied on them.

- Possible to return only safe references, which are those that point to object fields and the ones that are passed to the application.

- Impossible to modify **ref locals** to point to another storage space, as they are initialized and confined to a fixed memory location.

Improved out Variables 1-3

The `out` keyword, just as `ref`, is useful for passing a parameter by reference. However, in any version of C# prior to 7.0, it is mandatory to declare the variable prior to specifying it as an `out` parameter.

Similar to `ref` parameters, using an out parameter involves specifying the out keyword in the calling method as well as in the definition of called method.

Although declaring out parameters is considered mandatory, it can be cumbersome to implement. This is because a developer has to state the full type such as int length or string `authorName` instead of using the var keyword to do so.

It was not possible to declare the type of these parameters in the method itself as it would give a compile-time error.

Improved out Variables 2-3

- However, in C# 7.0 and later versions, developers can now specify data type of all out parameters inline instead of declaring them prior to passing them as parameters.
- Following code demonstrates an example of `out` variable:

Snippet

```
class BookApplication {
 static void Main(string[] args) {
        BookByOutArg(out string bName, out string bAuthor);
        Console.WriteLine("Book: {0}, Author: {1}", bName, bAuthor);
        Console.ReadKey();
 }
 static void BookByOutArg(out string name, out string author)
 {
        name = "Harry Potter Part I ";
        author = "J. K. Rowling";
 }
}
```

Improved out Variables 3-3

- In the previous code, the type of each variable specified as an `out` parameter follows the `out` keyword.
- The `out` parameters are within the scope of their own enclosing block. Thus, the succeeding line can refer to them.
- As variables having `out` keyword are declared straightaway as `out` parameters, it is possible for the compiler to determine their data type.
- There should not be any contradictory overloads.
- Allows developers to declare them using the `var` keyword. So, it is valid to declare an `out` parameter as:

```
out var bookName
```

- The out parameters are typically useful in the `try` block in which a returned Boolean value indicates true and the `outs` parameters hold the fetched outcome.

Discards 1-2

Discard a returned value if it is not required.

If a local variable is nowhere use, it is sensible to discard it. Logically, such a variable should neither have a name nor a value. This is exactly what **discard** does.

The developer just specify that this variable is discarded and required to use the underscore symbol

A discard parameter is a write-only parameter, which means it is impossible to read or obtain its value.

Developers can discard any number of variables, including the out parameters.

Discards 2-2

- Following code demonstrates the use of Discards:

Snippet

```
Console.WriteLine("Specify a number");
if int.TryParse(Console.ReadLine(), out int _))
{
      Console.WriteLine("Given data is number");
      Console.ReadKey();
}
```

- In this code:
  - When the user specifies a numeric value in the input, the program processes it and displays "Given data is number".
  - When the user specifies a non-numeric value, it is simply ignored.

Improved Tuples 1-4

Tuples are not completely new in C# 7.0. In .NET Framework 4.0, a set of Tuple classes has been introduced in the System.Tuple namespace.

Tuples in C# 7.0 provides a better mechanism to return multiple values from a method.

It also enables tuples to deduce the names of their elements from the inputs. For instance, a developer can now write `(emp.name, emp.age)` rather than `(name: emp.name, age: emp.age)`.

The compiler can easily infer these names due to which there is no necessity to set them explicitly.

Improved Tuples 2-4

- Following code demonstrates the use of improved tuples:

Snippet

```
string ename = "Emy George";
int e_age = 30;
var empTuple = (ename: ename, e_age: e_age);
Console.WriteLine(empTuple.ename); //Emy George
Console.WriteLine(empTuple.e_age); //30
Console.ReadKey();
```

- This code defines two elements of a tuple without any name inference. For this code to work, it is essential to import a NuGet package named `System.ValueTuple`.
- To select and install this package, right-click **References** and select **Manage NuGet Packages**.
- Following code demonstrates how to specify element names manually:

Snippet

```
string ename = "Emy George";
int e_age = 30;
var empTuple = (ename: ename, e_age: e_age);
Console.WriteLine(empTuple.ename); //Emy George
Console.WriteLine(empTuple.e_age); //30
```

- In this code, the element is inferred by a meaningful name. Item1, Item 2, and so on are not required to access the individual elements
- The compiler for C# 7.1 onwards is capable of inferring the element names of a tuple from local variables, null conditional members, and other members such as properties.

Improved Tuples 3-4

- Following code demonstrates inferred element names:

Snippet

```
string ename = "Emy George";
int e_age = 30;
var empTuple = (ename, e_age);
Console.WriteLine(empTuple.ename); //Emy George
Console.WriteLine(empTuple.e_age); //30
```

- Following code demonstrates the inferred element names through a property:

Snippet

```
class Employee {
 string ename;
 int e_age;
 static void Main() {
            Employee emp = new Employee { ename = "Emy George", e_age = 30};
            var empTuple = (emp.ename, emp.e_age);
            Console.WriteLine(empTuple.ename); // Displays Emy George
            Console.WriteLine(empTuple.e_age); //Displays 30
            Console.ReadLine();
 }
}
```

- Following code demonstrates the inferred element names through a null conditional property:

Snippet

```
class Employee {
 string ename;
 int eage;
 static void Main() {
            Employee emp = null;
            var empTuple = (emp?.ename, emp?.eage);
            Console.WriteLine(empTuple.ename); //Displays null or blank
            Console.WriteLine(empTuple.eage); //Displays null or blank
 }
}
```

Improved Tuples 4-4

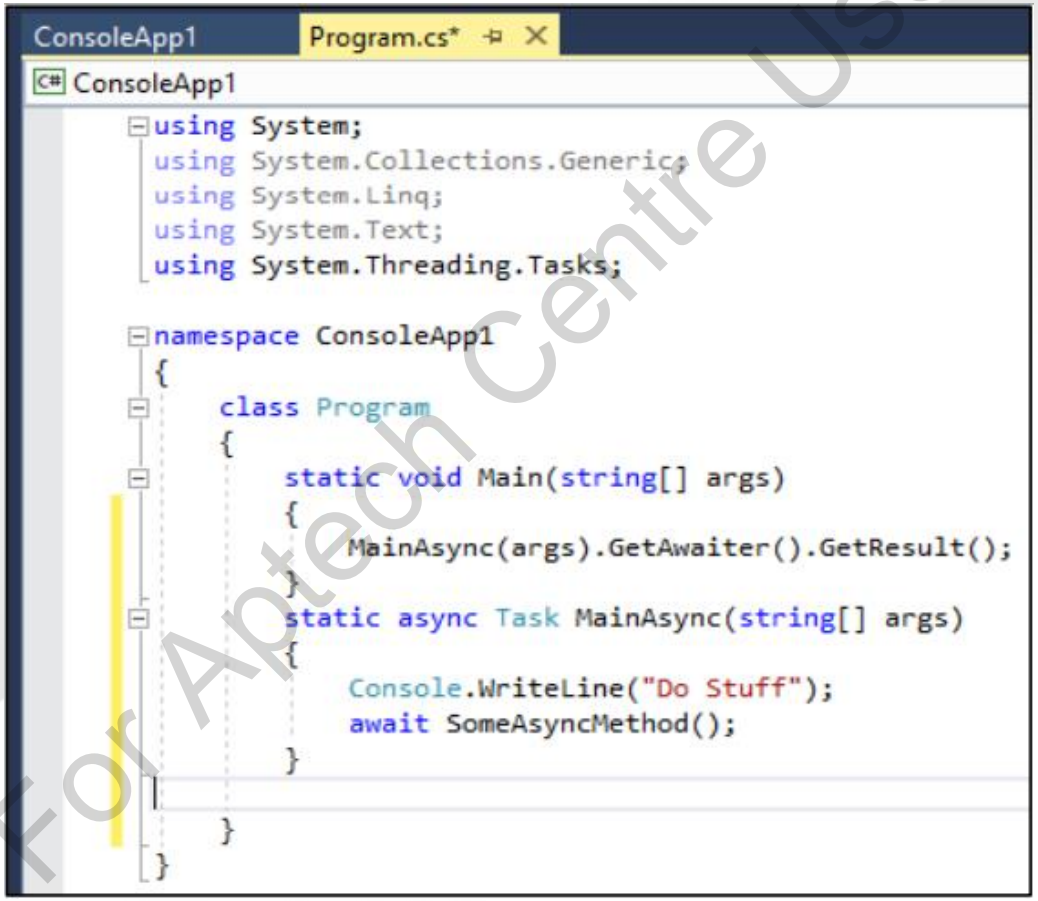
- Elements of a tuple are public and changeable fields, while tuples are value types.
- Two tuples are equal if their elements are the same pairwise. Even hash code of these two tuples and their elements is the same.
- As a result, tuples are useful for fulfilling some more requirements apart from returning multiple values.
- For example, a developer can use a tuple as a key if a dictionary with several keys is required.
- Similarly, if the requirement is of a list with several values at each position, defining a tuple at each position is an ideal option.

Improved Asynchronized Main() 1-4

- While the `asynchronous Main()` function was intended for C# 7.0, it was made available only with C# 7.1.
- This feature makes it simpler to use asynchronous methods with the help of async and await keywords.
- Prior to this version of C#, the `Main()` method acting as the entry point of the program's execution had following signatures:
  - `public static void main();`
  - `public static void main(string[] args);`
  - `public static int main();`
  - `public static int main(string[] args);`
- In older C# versions prior to 7.1, it was possible to await an asynchronous method only if it is invoked from inside another asynchronous method.
- For instance, a C# 7.0 console application cannot mark the `Main()` as `async`.

Improved Asynchronized Main() 2-4

Following figure shows the only option to add some more code so that it could work to use asynchronous code:



Improved Asynchronized Main() 3-4

- From C# 7.1 onwards, it is possible to await asynchronous methods within the `Main()` method directly.
- It is now the compiler's responsibility to generate the mandatory code for proper functioning of the program's entry point.
- Extra signatures that the `Main()` method now supports for asynchronous code:
  - `public static Task main();`
  - `public static Task main(string[] args);`
  - `public static Task<int> main();`
  - `public static Task<int> main(string[] args);`
- It can be seen that the method has two additional return types namely, `Task` and `Task<int>`.
- The `Task` class denotes an action that runs asynchronously and has no return value. It is simple to know the status of a task using its `Status`, `IsCompleted`, `IsCanceled`, and `IsFaulted` properties.
- Unlike older versions prior to it, C# 7.1 supports general types as return types of asynchronous methods.
- Supports `ValueTask<T>` struct type, which avoids allotting a `Task<T>` object when the outcome exists while in the await mode.
- Significantly bring down the number allocations in asynchronous scenarios for handling buffering.
- It is now possible to specify the `Main()` method of a C# 7.1 application as `async`.
- Simplifies code and eliminates the additional boilerplate code.

Improved Asynchronized Main() 4-4

- Following code shows another code that uses await keyword for enforcing crisp coding:

Snippet

```
class TaskTest {
 static async Task Main(string[] args) {
            Console.WriteLine($"It is {System.DateTime.Now}");
            await Task.Delay(3000);
            Console.WriteLine($"It is {System.DateTime.Now}");
            Console.ReadKey();
 }
}
```

- In this code, `Task.Delay` postpones the execution of the second `Console.WriteLine()` by three seconds. The code is crisper, as there is no necessity to invoke `GetAwaiter.GetResult()`.
- Following code shows the use of `Task<int>` by calling an asynchronous method in `Main()`:

Snippet

```
class TaskTest {
 static async Task<int> Main(string[] args) {
            var no = 6;
            Console.WriteLine($"Factorial of {no}: {await AsFact(no)}");
            Console.ReadKey();
            return 0;
 }
private static Task<int> AsFact(int num) {
 return Task.Run(() => Compute(num));
 //Local function
 int Compute(int p) {
            if (p == 1) {
                        return 1;
            }
 else {
            return p * Compute(p - 1);
            }
 }
}
}
```

Improved throw Expressions

- In C#, `throw` statements are used while catching exceptions or take an in-between action.
- A `throw` statement is a separate statement, which means it is impossible to merge it with another statement or expression.
- Following code shows usage of throw:

Snippet

```
if (num1 == null) (
 throw new ArgumentNullException(nameof(num1));
}
```

- However, in C# 7.0, it is now possible to include the throw statement in an expression, making it a throw expression.
- Developers can use it in the middle of an expression such as a conditional ternary operator.
- Following code demonstrate null-coalescing and assignment operators:

Snippet

```
myNum = num1 ?? throw new ArgumentNullException(nameof(num1));
```

- As it is possible to use throw in expressions, developers can use the ?: operator to represent an `if/else` statement as shown in the following code:

Snippet

```
returnval <10 ? val : throw new
ArgumentOutOfRangeException("Value has to be less than 10");
```

More Expression-bodied Members 1-4

C# 6.0 introduced expression-bodied members, which were usually methods. This feature simplified the syntax of C# methods.

An expression-bodied member replaces a block of code with a single statement for execution.

A developer can define its body as a lambda expression. However, this construct was only limited to methods.

From C# 7.0 onwards, developers can extend this feature to cover more members such as property accessors, destructors, and constructors.

More Expression-bodied Members 2-4

Following code demonstrate how to use this feature for methods, properties, as well as constructor and destructor in C#:

Snippet

```
class Program {
 static void Main(string[] args) {
       Console.WriteLine($"Using expression-bodied constructor and
           destructor:\n{new Demo().prodDetail()}");
       Console.ReadLine();
       var obj = new Demo();
       obj.PriceList.Add(101, 480);
       obj.PriceList.Add(102, 500);
       obj.prodId = 101;
       Console.WriteLine($"Using expression-bodied getters and setters");
       Console.WriteLine($"The price of Product {obj.prodId} is:
              {obj.Price}");
       obj.Price = 550;
       Console.WriteLine($"The updated price of Product {obj.prodId} is:
           {obj.Price}");
       Console.ReadLine();
 }
class Demo {
        public Dictionary<int, double> PriceList = new
                Dictionary<int, double>();
        public int prodId { get; set; } = 101;
        string prodName { get; } = "Box Nails";
        string  prodCategory { get; } = "Nails";
        double  prodPrice { get; }
```

More Expression-bodied Members 3-4

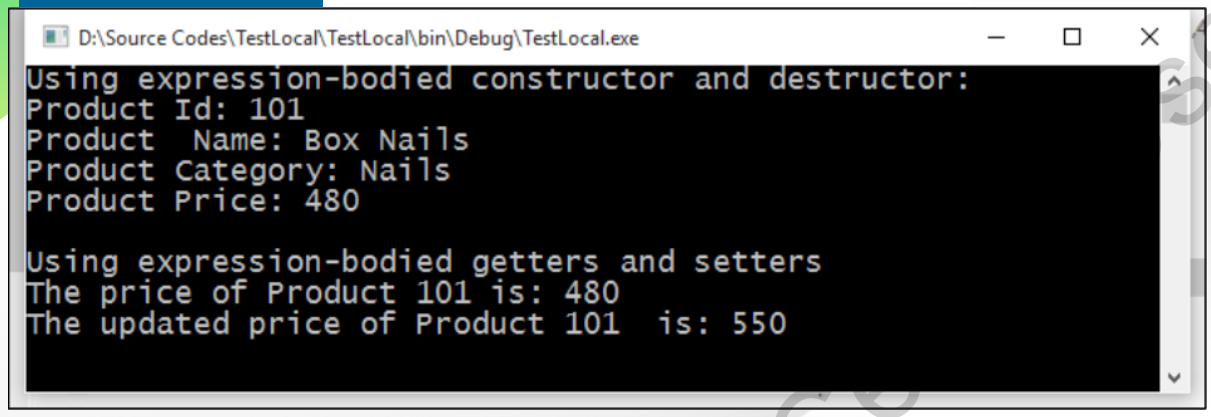
```
        public double Price {
                    /// Expression-bodied Getters and Setters
                    get => PriceList[prodId];
                    set => PriceList[prodId] = value;
                }
                /// <summary>
                /// Expression-bodied constructor
                /// </summary>
               public Demo() => prodPrice = 480;
                /// <summary>
                /// Expression-bodied destructor
                /// </summary>
                ~Demo() => Console.WriteLine("\n Destructor of Demo class");
                  public string prodDetail() => $"Product Id: {prodId} \nProduct Name:
                     {prodName}\nProduct Category: {prodCategory}\nProduct Price:
                            {prodPrice}";
}
```

- In this code, the expression-bodied syntax is applied to a constructor, a destructor, and property accessors.
- In case of property accessors, developers can implement an auto property for simplifying the code further.
- However, this is a better option when properties require an explicit accessor (getter or setter).

More Expression-bodied Members 4-4

The expression-bodied syntax is clean and brings down the number of lines of codes.

Output



Developers can also use an expression-bodied method to invoke an asynchronous method in `Main()`

Snippet

```
static async Task Main(string[] args) => WriteLine($"Factorial 6:
{await AsFact(6)}");
```

Pattern Matching Enhancements 1-4

- Pattern matching is a way to check if a particular grouping of characters, tokens or information exists among the given information.
- Microsoft first introduced pattern matching in C# 7.0 and later added other patterns.
- Pattern matching is utilized to decide if source records of significant level dialects are right.
- Any application that supports search function somehow utilizes a matching pattern.
- It is likewise used to discover and replace a matching pattern with design in a text or code with another text or code.

Pattern Matching Enhancements 2-4

Evolution of Pattern Matching

Declaration pattern

This pattern matching was first introduced in C#7.0 and was used to check the run type of an expression.

Constant pattern

This pattern was also introduced in C# 7.0. This pattern tests whether the expression result equals to a specified constant.

Var pattern

First introduced in C# 7.0 and is used to match any expression and assign the result obtained to a declared variable.

Property pattern

Property pattern was first introduced in C# 8.0 and it tests whether properties of expression or fields match nested patterns.

Positional pattern

Introduced in C# 8.0. It is used to deconstruct an expression result and test if the resulting values match the nested patterns.

Pattern Matching Enhancements 3-4

C#9.0

- **Type pattern** – It is introduced first time in C# 9.0 and is used to check the run time type of an expression.
- **Relational pattern** - It is used to compare an expression result with a constant.
- **Logical patterns** - They test whether an expression matches the logical combination of patterns. AND, OR, and NOT are the logical operators which are used for the matching process. These operators create logical patterns such as conjunctive, disjunctive, and negation.
  - **Conjunctive** - Conjunctives are AND pattern. It matches an expression when both patterns match the expression.
  - **Disjunctive** - Similarly, Disjunctive is an OR pattern. It matches an expression when either of patterns matches the expression.

Pattern Matching Enhancements 4-4

- Following code shows an example of relational pattern:

Snippet

```
Console.WriteLine(Classify(50)); // output: Too high
Console.WriteLine(Classify(double.NaN)); // output: Unknown
Console.WriteLine(Classify(2.5)); // output: Acceptable
static string Classify(double measurement) => measurement
switch
{
        < -4.0 => "Too low",
        > 10.0 => "Too high",
        double.NaN => "Unknown",
        _ => "Acceptable",
};
```

- The code describes the relation between outputs. You can use any of the relational operators such as '<,>, or =' for finding a relational match.
- The constant expression can be an integer, floating-point, char, or enum type.

Fit and Finish Features 1-4

C# 9.0 now also includes many other key features that are important to write efficient codes.

Target-typed new expressions:

- When the type is unknown, new expressions do not require type specification for constructors. Thus, when a created object's type is already known, one can omit the type in a new expression.
- For example, `private List<List name> _name = new();`

Static anonymous functions:

- Static anonymous functions or methods are an improvement to anonymous methods in C# 9.0. When a static modifier is applied to a lambda or an anonymous method, the outcome is known as a static anonymous function. Such a function may reference static members from the enclosing scope.

Target-typed Conditional expressions:

- C# 9.0 also includes a few enhancements with target-typed conditional expressions, namely improved ternary statements. The branches of ? .. : .. expressions are now permitted to have different types, as long as both of them convert to the target type.

Fit and Finish Features 2-4

- Following code shows code written in C# 8.0.

Snippet

```
public abstract class Publication {
}
public class Book : Publication {
}
public class Journal : Publication {
}
public class Program {
          static void Main(string[] args)  {
                    var flag = false;
                    Publication p = flag ? new Book() : new Journal();
                    Console.WriteLine("It is working!");
          }
}
```

- The code defines a base class named Publication and two inherited classes Book and Journal.
- Within Main, we attempt to check the value of the flag variable and assign instance p to an instance of either Book or Journal depending on whether flag is true or false, respectively.
- However, the code gives a compiler error in C# 8.0 or lower because that both sides of the expression should be of the same type.
- Target-typed conditional expression is a feature that can remove this restriction, but the feature is not available in C# 8.0 or lower versions.
- The error shown by the compiler is as follows:

```
Feature 'target-typed conditional expression' is not available in C# 8.0. Please
    use language version 9.0 or greater.
```

Fit and Finish Features 3-4

- The same code without any changes when executed with .NET 5.0 and C# 9.0 works successfully.
- This is because C# 9.0 supports target-typed conditional expressions, where both sides of a conditional expression can belong to different types.

Improved Support for code generators:

- A code generator is a C# feature that reads attributes or other code elements using the Roslyn (nickname for the .NET compiler) analysis APIs. Through that data, it generates and adds new code to the compilation.
- Source generators cannot modify any existing code in the compilation, they can only add code.
- To enhance performance, C# 9.0 has introduced two new support features for code generators: partial method syntax and module initializers.
- Before C# 9.0, partial methods were private and had some limitations, such as not permitting access modifiers, void returns, or out parameters.

Fit and Finish Features 4-4

- This meant that developers had to edit the code produced by source generators that did not conform to these limitations. Productivity was reduced and chances of errors increased.
- Productivity was reduced and chances of errors increased.
- With the introduction of C# 9.0, these restrictions for partial method syntax were removed.
- Module Initializer is the second new improvement in code generators.
- Module Initializers are techniques through which a `ModuleInitializerAttribute` decorator is specified for methods to denote that the method is a module initializer.
- A module initializer is a method that is run when an assembly is first loaded.
- It is somewhat like a static constructor in C#, but instead of applying to one class, it applies to the entire assembly.
- Module initializers have certain conventions that apply, namely, they must be static, parameterless, and should return void.
- Furthermore, they must not be generic methods or contained in a generic class.

Performance and Native Interoperability (Interop) 1-4

- Besides .NET and C# code which are also called as managed code, there are other libraries, APIs and code which are not under the control of the .NET Framework or the CLR. Such libraries, APIs, and code are called as native or unmanaged code.
- There are some scenarios in which managed code has to interact with unmanaged code.
- Microsoft provides the technology to facilitate this 'interoperability' between native and managed code.
- Through this, developers can access functionality in existing native and COM libraries that is not exposed by the .NET Framework and take advantage of the speed and ease of .NET without requiring to rewrite existing code.

Performance and Native Interoperability (Interop) 2-4

- C# 9.0 has introduced the following three new features to help to enhance performance of a number of low-level libraries:
  - Native sized integers
  - Function pointers
  - Omitting the `localsinit` flag
- These features provide additional support to native interops.

Native sized integers

- The integers whose size is specific to the platform are known as native size integers.
- When running in a 32-bit process – these are 32-bit integers. While running in a 64-bit process, these are 64-bit integers.
- `nint` and `nuint` are contextual keywords used to define native size integers native-sized signed and unsigned integer types.
- `nint` and `nuint` are expressed by underlying types (a platform-specific type that is used to represent a pointer or a handle) namely, `System.IntPtr` and `System.UIntPtr`.

Performance and Native Interoperability (Interop) 3-4

Following code demonstrates native sized integers.

Snippet

```
public static void Main() {
         int aa = 300;
         nint bb = 300;
         nint cc = bb + 1;
         var type1 = typeof(nint); // displays System.IntPtr
         //when printed

        var type2 = typeof(nuint); // displays System.UIntPtr
         //when printed

        long v = 100;
         var type3 = (aa + v).GetType(); // displays
         System.Int64 when printed
         Console.WriteLine("{0}, {1}, {2}", type1, type2,
          type3);
}
```

Output

```
System.IntPtr, System.UIntPtr, System.Int64
```

Performance and Native Interoperability (Interop) 4-4

- **Function pointers**
  - C# supports usage of methods. Unmanaged code use the term 'function' for their subroutines.
  - A variable that stores the memory address of a function is called as a function pointer.
  - The function can later be called through that function pointer. Just as in a normal function call, function pointers can also be invoked and passed arguments.
  - Using the newly introduced delegate* syntax in C# 9.0, a function pointer can be easily declared. A delegate type is a pointer type.
- **Omitting the localsinit flag**
  - This feature is used to instruct the compiler to not emit the localsinit flag.
  - This is a specific flag that instructs the CLR to zero-initialize all local variables and was always the default since C# 1.0.
  - Now in C# 9.0, you can override that default behavior by telling the compiler to stop zero initialization for local variables.

Summary 1-2

- In recent versions of C# from 7.0 onwards, many new features such as binary literals, digit separators, pattern matching, async main, inferred tuple element names, and more, have been introduced.
- One of the most flexible features of C# is the ability to make the compiler work as per the preferred language version.
- Using the light bulb icon is perhaps the easiest way to enable C# and utilize any of its new features.
- Developers can update the project `.csproj` file to manually change the language version.
- Visual Studio 2017 allows you to set the desired language version for all projects at once.
- The `ref` keyword comes with some more abilities, where a developer can use it to return values passed by reference.

Summary 2-2

- The `ref` keyword is also now capable of storing values passed by reference in local variables.
- Using an out parameter involves specifying the out keyword in the calling method as well as in the definition of called method.
- C# enables developers to discard a returned value if it is not required.
- C# 7.1 added a minor enhancement to tuples, which is known as tuple name inference, which allows working with tuples as value types.
- asynchronous `Main()` makes it simpler to use asynchronous methods with the help of async and await keywords.
- In C#, `throw` statements are used while catching exceptions or take an in-between action.
- C# 6.0 introduced expression-bodied members, which were usually methods. This feature simplified the syntax of C# methods.
- Pattern matching is a way to check if a particular grouping of characters, tokens or information exists among the given information.
- Besides .NET and C# code which are also called as managed code, there are other libraries, APIs and code which are not under the control of the .NET Framework or the CLR. Such libraries, APIs, and code are called as native or unmanaged code.