Session: 14

Advanced Methods and Types

Objectives

- Describe anonymous methods
- Define extension methods
- Explain anonymous types
- Explain partial types
- Explain nullable types

Anonymous Methods

- An anonymous method is an inline nameless block of code that can be passed as a delegate parameter.
- Delegates can invoke one or more named methods that are included while declaring the delegates.
- Prior to anonymous methods, if you wanted to pass a small block of code to a delegate, you always had to create a method and then pass it to the delegate.
- With the introduction of anonymous methods, you can pass an inline block of code to a delegate without actually creating a method.
- The following code displays an example of anonymous method:

```
void Action()
{
    System.Threading.Thread objThread = new
    System.Threading.Thread
    (delegate()
        {
            Console.Write("Testing… ");
            Console.WriteLine("Threads.");
        });
    objThread.Start();
}
```

Anonymous Method

- An anonymous method is used in place of a named method if that method is to be invoked only through a delegate.
- An anonymous method has the following features:
  - It appears as an inline code in the delegate declaration.
  - It is best suited for small blocks.
  - It can accept parameters of any type.
  - Parameters using the `ref` and `out` keywords can be passed to it.
  - It can include parameters of a generic type.
  - It cannot include jump statements such as `goto` and `break` that transfer control out of the scope of the method.

Creating Anonymous Methods 1-3

- An anonymous method is created when you instantiate or reference a delegate with a block of unnamed code.

- Following points need to be noted while creating anonymous methods:
  - When a `delegate` keyword is used inside a method body, it must be followed by an anonymous method body.
  - The method is defined as a set of statements within curly braces while creating an object of a delegate.
  - Anonymous methods are not given any return type.
  - Anonymous methods are not prefixed with access modifiers.

Creating Anonymous Methods 2-3

- The following figure and snippet display the syntax and code for anonymous methods respectively:

```
// Create a delegate instance

<access modifier> delegate <return type>
<DelegateName> (parameters);

// Instantiate the delegate using an anonymous method

<DelegateName> <objDelegate> = new <DelegateName>
(parameters)
{ /* ... */ };
```

Snippet

```
using System;
class AnonymousMethods
{
     //This line remains same even if named methods are used
     delegate void Display();
     static void Main(string[] args)
     {
         //Here is where a difference occurs when using
         // anonymous methods
         Display objDisp = delegate()
         {
           Console.WriteLine("This illustrates an anonymous method");
         };
        objDisp();
         }
}
```

- In the code:
  - A delegate named **`Display`** is created.
  - The delegate **`Display`** is instantiated with an anonymous method.
  - When the delegate is called, it is the anonymous block of code that will execute.

Output

```
This illustrates an anonymous method
```

Referencing Multiple Anonymous Methods 1-2

- C# allows you to create and instantiate a delegate that can reference multiple anonymous methods.
- This is done using the += operator.
- The += operator is used to add additional references to either named or anonymous methods after instantiating the delegate.
- The following code shows how one delegate instance can reference several anonymous methods:

Snippet

```
using System;
class MultipleAnonymousMethods
{
    delegate void Display();
    static void Main(string[] args)
    {
        //delegate instantiated with one anonymous
        // method reference
        Display objDisp = delegate()
        {
            Console.WriteLine("This illustrates one anonymous
             method");
        };
```

Referencing Multiple Anonymous Methods 2-2

```
            //delegate instantiated with another anonymous method
            // reference
            objDisp += delegate()
            {
                Console.WriteLine("This illustrates another anonymous
                 method with the same delegate instance");
            };
            objDisp();
        }
}
```

- In the code:
  - An anonymous method is created during the delegate instantiation and another anonymous method is created and referenced by the delegate using the += operator.

Output

```
This illustrates one anonymous method

This illustrates another anonymous method with the
same delegate instance
```

Outer Variables in Anonymous Methods

- An anonymous method can declare variables, which are called outer variables.
- These variables are said to be captured when they get executed.
- They exist in memory until the delegate is subjected to garbage collection.
- The scope of a local variable is only within the method in which it is declared.
- However, if the anonymous method uses local variables, they exist until the execution of the anonymous method ends.
- This is true even if the methods in which they are declared are already executed.

Passing Parameters 1-2

- C# allows passing parameters to anonymous methods.
- The type of parameters that can be passed to an anonymous method is specified at the time of declaring the delegate.
- These parameters are specified within parentheses.
- The block of code within the anonymous method can access these specified parameters just like any normal method.
- You can pass the parameter values to the anonymous method while invoking the delegate.
- The following code demonstrates how parameters are passed to anonymous methods:

Snippet

```
using System;
class Parameters
{
    delegate void Display(string msg, int num);
    static void Main(string[] args)
    {
    Display objDisp = delegate(string msg, int num)
    {
        Console.WriteLine(msg + num);
    };
    objDisp("This illustrates passing parameters to
    anonymous methods. The int parameter passed is: ", 100);
    }
}
```

- In the code:
  - A delegate **`Display`** is created.
  - Two arguments are specified in the delegate declaration, a `string` and an `int`.
  - The delegate is then instantiated with an anonymous method to which the `string` and `int` variables are passed as parameters.
  - The anonymous method uses these parameters to display the output.

Output

```
This illustrates passing parameters to anonymous
 methods. The int parameter passed is: 100
```

Extension Methods 1-7

- Extension methods allow you to extend an existing type with new functionality without directly modifying those types.
- Extension methods are static methods that have to be declared in a static class.
- You can declare an extension method by specifying the first parameter with the `this` keyword.
- The first parameter in this method identifies the type of objects in which the method can be called.
- The object that you use to invoke the method is automatically passed as the first parameter.

Syntax

```
static return-type MethodName (this type obj, param-list)
```

where:

- `return-type`: the data type of the return value
- `MethodName`: the extension method name
- `type`: the data type of the object
- `param-list`: the list of parameters (optional)

- The following code creates an extension method for a string and converts the first character of the string to lowercase:

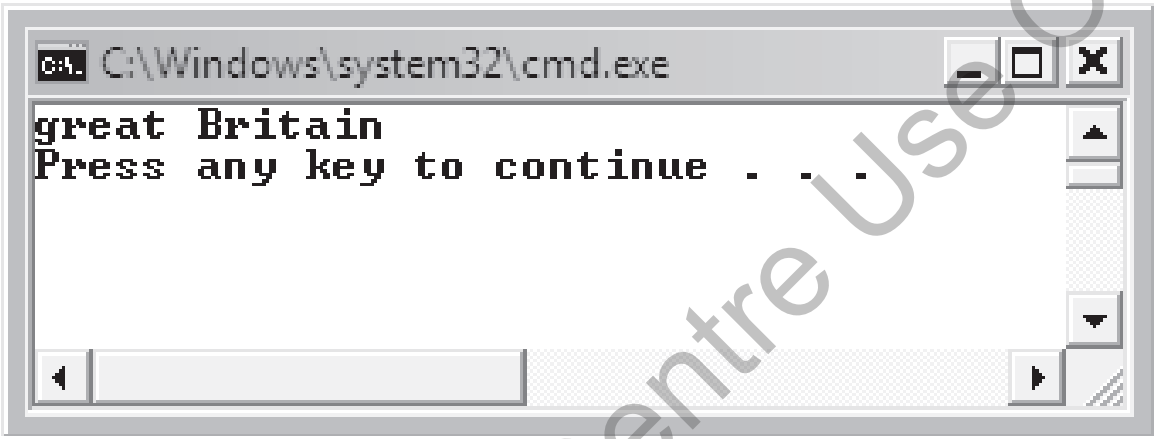
Snippet

```
using System;

/// <summary>
/// Class ExtensionExample defines the extension method
/// </summary>

static class ExtensionExample
{
   // Extension Method to convert the first character to
   //lowercase
    public static string FirstLetterLower(this string result)
    {
          if (result.Length > 0){
              char[] s = result.ToCharArray();
              s[0] = char.ToLower(s[0]);
              return new string(s);
          }
      return result;
      }
}
```

```
class Program
{
   public static void Main(string[] args)
   {
    string country = "Great Britain";
    // Calling the extension method
    Console.WriteLine(country.FirstLetterLower());
   }
}
```

- In the code:
  - An extension method named **`FirstLetterLower`** is defined with one parameter that is preceded with `this` keyword.
  - This method converts the first letter of any sentence or word to lowercase.
  - Note that the extension method is invoked by using the object, **`country`**.
  - The value ‘Great Britain’ is automatically passed to the parameter result.

- The following figure depicts the output:



- The advantages of extension methods are as follows:
  - You can extend the functionality of the existing type without modification. This will avoid the problems of breaking source code in existing applications.
  - You can add additional methods to standard interfaces without physically altering the existing class libraries.

- The following code is an example for an extension method that removes all the duplicate values from a generic collection and displays the result.
- This program extends the generic `List` class with added functionality.

Snippet

```
using System;
using System.Collections.Generic;
/// <summary>
/// Class ExtensionExample defines the extension method
/// </summary>
static class ExtensionExample
{
     // Extension method that accepts and returns a collection.
     public static List<T> RemoveDuplicate<T>(this List<T> allCities)
     {
          List<T> finalCities = new List<T>();
          foreach (var eachCity in allCities)
          if (!finalCities.Contains(eachCity))
          finalCities.Add(eachCity);
          return finalCities;
     }
}
```

```
class Program
{
    public static void Main(string[] args)
    {
        List<string> cities = new List<string>();
        cities.Add("Seoul");
        cities.Add("Beijing");
        cities.Add("Berlin");
        cities.Add("Istanbul");
        cities.Add("Seoul");
        cities.Add("Istanbul");
        cities.Add("Paris");
        // Invoke the Extension method, RemoveDuplicate().
        List<string> result = cities.RemoveDuplicate();
        foreach (string city in result)
        Console.WriteLine(city);
    }
}
```

- In the code:
  - The extension method **`RemoveDuplicate()`** is declared and returns a generic `List` when invoked.
  - The method accepts a generic `List<T>` as the first argument:

```
public static List<T> RemoveDuplicate<T>(this List<T>
 allCities)
```

- The following lines of code iterate through each value in the collection, remove the duplicate values, and store the unique values in the `List`, **`finalCities`**:

```
foreach (var eachCity in allCities)
if (!finalCities.Contains(eachCity))
finalCities.Add(eachCity);
```

- The following figure displays the output:

Output

- Anonymous type:
  - Is basically a class with no name and is not explicitly defined in code.
  - Uses object initializers to initialize properties and fields. Since it has no name, you need to declare an implicitly typed variable to refer to it.

Syntax

```
new { identifierA = valueA, identifierB =
valueB, …… }
```

where,

- `identifierA, identifierB, …`: Identifiers that will be translated into read-only properties that are initialized with values

- The following code demonstrates the use of anonymous types:

Snippet

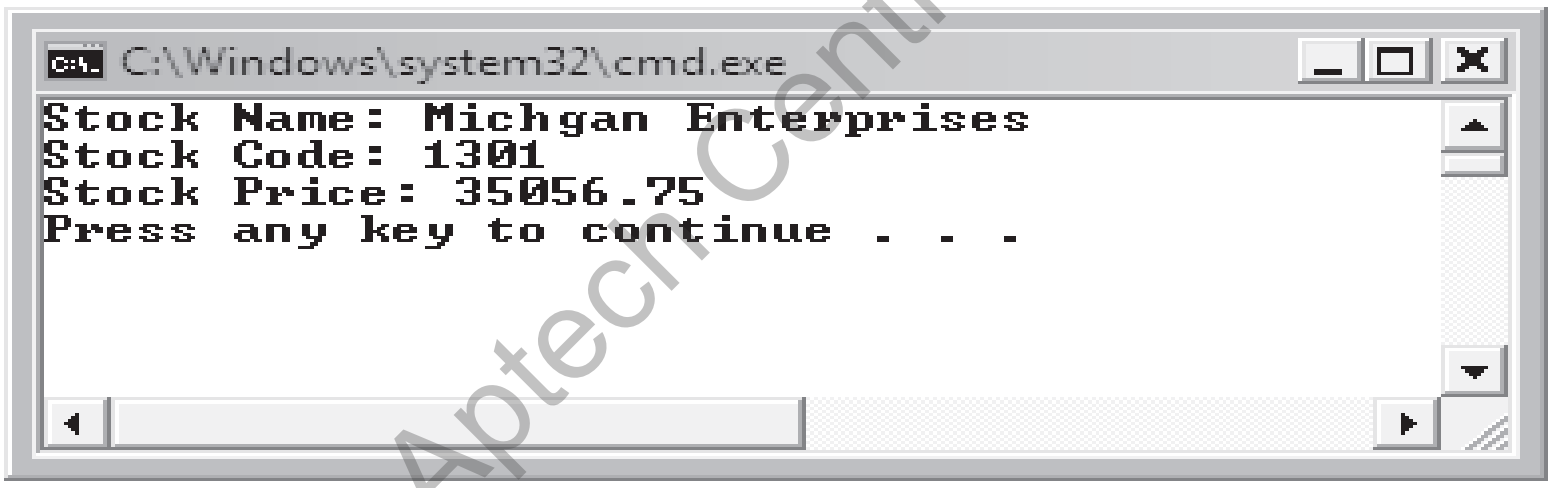
```
using System;
/// <summary>
/// Class AnonymousTypeExample to demonstrate anonymous type
/// </summary>
class AnonymousTypeExample
{
    public static void Main(string[] args)
    {
        // Anonymous Type with three properties.
        var stock = new { Name = "Michgan Enterprises", Code = 1301,
         Price = 35056.75 };
        Console.WriteLine("Stock Name: " + stock.Name);
        Console.WriteLine("Stock Code: " + stock.Code);
        Console.WriteLine("Stock Price: " + stock.Price);
    }
}
```

- Consider the following line of code:

```
var stock = new { Name = "Michgan Enterprises", Code =
1301, Price = 35056.75 };
```

- The compiler creates an anonymous type with all the properties that is inferred from object initializer.
- In this case, the type will have properties **Name**, **Code**, and **Price**.

- The compiler automatically generates the `get` and `set` methods, as well as the corresponding private variables to hold these properties.
- At runtime, the C# compiler creates an instance of this type and the properties are given the values Michgan Enterprises, 1301, and 35056.75 respectively.
- The following figure displays output:



- When an anonymous type is created, the C# compiler carries out the following tasks:
  - Interprets the type
  - Generates a new class
  - Use the new class to instantiate a new object
  - Assigns the object with the required parameters

- The compiler internally creates a class with the respective properties when code is compiled.
- In this program, the class might look like the one that is shown in code.

- In this program, the class might look like the one that is shown in the following code:

Snippet

```
class __NO_NAME__
{
    private string _Name;
    private int _Code;
    private double _Price;
    public string Name
    {
       get { return _Name; }
       set { _Name = value; }
    }
    public int Code
    {
       get { return _Code; }
       set { _Code = value; }
    }
    public double Price
    {
       get { return _Price; }
       set { _Price = value; }
    }
}
```

- The following code demonstrates passing an instance of the anonymous type to a method and displaying the details:

Snippet

```
using System;
using System.Reflection;
/// <summary>
/// Class Employee to demonstrate anonymous type.
/// </summary>
public class Employee
{
      public void DisplayDetails(object emp)
      {
        String fName = "";
        String lName = "";
        int age = 0;
        PropertyInfo[] attrs = emp.GetType().GetProperties();
        foreach (PropertyInfo attr in attrs)
        {
            switch (attr.Name)
            {
                case "FirstName":
                fName = attr.GetValue(emp, null).ToString();
                break;
                case "LastName":
                lName = attr.GetValue(emp, null).ToString();
                break;
                case "Age":
                age = (int)attr.GetValue(emp, null);
                break;
            }
        }
```

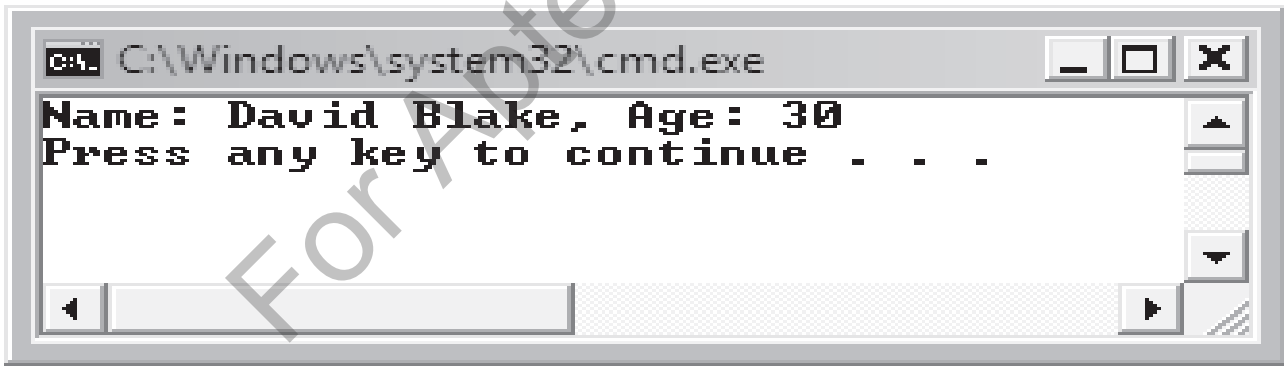
```
            Console.WriteLine("Name: {0} {1}, Age: {2}", fName, lName,
             age);
        }
}
class AnonymousExample
{
        public static void Main(string[] args)
        {
            Employee david = new Employee();
            // Creating the anonymous type instance and passing it
            // to a method.
            david.DisplayDetails(new { FirstName = "David", LastName =
             "Blake", Age = 30 });
        }
}
```

In the code:

- It creates an instance of the anonymous type with three properties, **`FirstName`**, **`LastName`**, and **`Age`** with values David, Blake, and 30 respectively.
- This instance is then passed to the method, **`DisplayDetails()`**.
- In **`DisplayDetails()`** method, the instance that was passed as parameter is stored in the object, emp.

- Then, the code uses reflection to query the object's properties.
- The **`GetType()`** method retrieves the type of the current instance, **`emp`** and **`GetProperties()`** method retrieves the properties of the object, **`emp`**.
- The details are then stored in the **`PropertyInfo`** collection, **`attr`**. Finally, the details are extracted through the `GetValue()` method of the **`PropertyInfo`** class.
- If this program did not make use of an anonymous type, a lot more code would have been required to produce the same output.

- The following figure displays the output:



Example

- Assume that a large organization has its IT department spread over two locations, Melbourne and Sydney.
- The overall functioning takes place through consolidated data gathered from both the locations.
- The customer of the organization would see it as a whole entity, whereas, in reality, it would be composed of multiple units.
- Now, think of a very large C# class or structure with lots of member definitions.
- You can split the data members of the class or structure and store them in different files.
- These members can be combined into a single unit while executing the program.
- This can be done by creating partial types.

Features of Partial Types

- The partial types feature facilitates the definition of classes, structures, and interfaces over multiple files.
- Partial types provide various benefits. These are as follows:
  - They separate the generator code from the application code.
  - They help in easier development and maintenance of the code.
  - They make the debugging process easier.
  - They prevent programmers from accidentally modifying the existing code.

- The following figure displays an example of a partial type:

File 1

```
partial struct Sample
{
 <MethodOne>;
}
```

File 2

```
partial struct Sample
{
 <MethodTwo>;
}
```

Merged Elements during Compilation 1-4

- The members of partial classes, partial structures, or partial interfaces declared and stored at different locations are combined together at the time of compilation.
- These members can include:
  - XML comments
  - Interfaces
  - Generic-type parameters
  - Class variables
  - Local variables
  - Methods
  - Properties
- A partial type can be compiled at the Developer Command Prompt for VS2012. The command to compile a partial type is:

```
csc /out:<FileName>.exe <CSharpFileNameOne>.cs <CSharpFileNameTwo>.cs
```

where,

`FileName`: Is the user specified name of the .exe file.
`CSharpFileNameOne`: Is the name of the first file where a partial type is defined.
`CSharpFileNameTwo`: Is the name of the second file where a partial type is defined.

Merged Elements during Compilation 2-4

- You can directly run the .exe file to see the required output. This is demonstrated in the following code:

Snippet

```
using System;
using System.Collections.Generic;
using System.Text;
//Stored in StudentDetails.cs file
namespace School
{
    public partial class StudentDetails
    {
        int _rollNo;
        string _studName;
        public StudentDetails(int number, string name)
        {
            _rollNo = number;
            _studName = name;
        }
    }
}
using System;
using System;
using System.Collections.Generic;
using System.Text;
//Stored in Students.cs file
namespace School
{
    public partial class StudentDetails
    {
    public void Display()
    {
```

Merged Elements during Compilation 3-4

```
        Console.WriteLine("Student Roll Number: " + _rollNo);
        Console.WriteLine("Student Name: " + _studName);

    }
    }
    public class Students
    {
    static void Main(string[] args)
    {
        StudentDetails objStudents = new StudentDetails(20,
        "Frank");
        objStudents.Display();
    }
    }
}
```

- In the code:
  - The partial class **`StudentDetails`** exists in two different files.
  - When both these files are compiled at the Visual Studio 2005 Command Prompt, an .exe file is created which merges the **`StudentDetails`** class from both the files.
  - On executing the exe file at the command prompt, the student's roll number and name are displayed as output.

Merged Elements during Compilation 4-4

- The following code shows how to compile and execute the **`StudentDetails.cs`** and **`Students.cs`** files created in the examples using Developer Command Prompt for VS2012:

Rules for Partial Types

- There are certain rules for creating and working with partial types.
- These rules must be followed, without which a user might not be able to create partial types successfully.
- The rules are as follows:
  - The partial-type definitions must include the `partial` keyword in each file.
  - The partial keyword must always follow the `class`, `struct`, or `interface` keywords.
  - The partial-type definitions of the same type must be saved in the same assembly.
  - Generic types can be defined as partial. Here, the type parameters and its order must be the same in all the declarations.
- The partial-type definitions can contain certain C# keywords which must exist in the declaration in different files. These keywords are as follows:
  - `public`
  - `private`
  - `protected`
  - `internal`
  - `abstract`
  - `sealed`
  - `new`

Implementing Partial Types 1-4

- Partial types are implemented using the `partial` keyword.
- This keyword specifies that the code is split into multiple parts and these parts are defined in different files and namespaces.
- The type names of all the constituent parts of a partial code are prefixed with the `partial` keyword.
- For example, if the complete definition of a structure is split over three files, each file must contain a partial structure having the `partial` keyword preceding the type name.
- Each of the partial parts of the code must have the same access modifier.

- The following syntax is used to split the definition of a class, a struct, or an interface:

Syntax

```
[<access_modifier>] [keyword] partial <type>
<Identifier>
```

where,

- `access_modifier`: Is an optional access modifier such as `public`, `private`, and so on.
- `keyword`: Is an optional keyword such as `abstract`, `sealed`, and so on.
- `type`: Is a specification for a class, a structure, or an interface.
- `Identifier`: Is the name of the class, structure, or an interface.

- The following figure creates an interface with two partial interface definitions:

Snippet

```
using System;
//Program Name: MathsDemo.cs
partial interface MathsDemo{
    int Addition(int valOne, int valTwo);
}
//Program Name: MathsDemo2.cs
partial interface MathsDemo{
    int Subtraction(int valOne, int valTwo);
}
class Calculation : MathsDemo{
    public int Addition(int valOne, int valTwo) {
        return valOne + valTwo;
    }
    public int Subtraction(int valOne, int valTwo) {
       return valOne - valTwo;
    }
     static void Main(string[] args) {
       int numOne = 45;
       int numTwo = 10;
       Calculation objCalculate = new Calculation();
       Console.WriteLine("Addition of two numbers: " +
       objCalculate.Addition(numOne, numTwo));
       Console.WriteLine("Subtraction of two numbers: " +
       objCalculate.Subtraction(numOne, numTwo));
      }
}
```

- In the code:
  - A partial interface **Maths** is created that contains the **Addition** method.
  - This file is saved as **MathsDemo.cs**. The remaining part of the same interface contains the **Subtraction** method and is saved under the filename **MathsDemo2.cs**.
  - This file also includes the class **Calculation**, which inherits the interface **Maths** and implements the two methods, **Addition** and **Subtraction**.

Output

```
Addition of two numbers: 55
 Subtraction of two numbers: 35
```

Partial Classes 1-3

- A class is one of the types in C# that supports partial definitions.
- Classes can be defined over multiple locations to store different members such as variables, methods, and so on.
- Although the definition of the class is split into different parts stored under different names, all these sections of the definition are combined during compilation to create a single class.
- You can create partial classes to store private members in one file and public members in another file.
- More importantly, multiple developers can work on separate sections of a single class simultaneously if the class itself is spread over separate files.
- The following code creates two partial classes that display the name and roll number of a student:

Snippet

```
using System;
//Program: StudentDetails.cs
public partial class StudentDetails
{
   public void Display()
```

```
    {
    Console.WriteLine("Student Roll Number: " + _rollNo);
    Console.WriteLine("Student Name: " + _studName);
    }
}
//Program StudentDetails2.cs
public partial class StudentDetails
{
    int _rollNo;
    string _studName;
    public StudentDetails(int number, string name)
    {
        _rollNo = number;
        _studName = name;
    }
}
public class Students
{
    static void Main(string[] args)
    {
        StudentDetails objStudents = new StudentDetails(20,
         "Frank");
        objStudents.Display();
    }
}
```

- In the code:
  - The class **StudentDetails** has its definition spread over two files, **StudentDetails.cs** and **StudentDetails2.cs**.
  - **StudentDetails.cs** contains the part of the class that contains the **Display()** method.
  - **StudentDetails2.cs** contains the remaining part of the class that includes the constructor.
  - The class **Students** creates an instance of the class **StudentDetails** and invokes the method **Display**.
  - The output displays the roll number and the name of the student.

Output

```
Student Roll Number: 20
Student Name: Frank
```

- Consider a partial class **Shape** whose complete definition is spread over two files.
- Now consider that a method **Create()** has a signature defined in **Shape**.
- The partial class **Shape** contains the definition of **Create()** in **Shape.cs**.
- The remaining part of partial class **Shape** is present in **RealShape.cs** and it contains the implementation of **Create()**.
- Hence, **Create()** is a partial method whose definition is spread over two files.

- A partial method is a method whose signature is included in a partial type, such as a partial class or `struct`.
- The method may be optionally implemented in another part of the partial class or type or same part of the class or type.
- The following code illustrates how to create and use partial methods.
- The code contains only the signature and another code contains the implementation:

Snippet

```
using System;
namespace PartialTest
{
/// <summary>
/// Class Shape is a partial class and defines a partial method.
/// </summary>
public partial class Shape
{
    partial void Create();
}
}
```

```
using System;

namespace PartialTest
{
/// <summary>
/// Class Shape is a partial class and contains the implementation
/// of a partial method.
/// </summary>
     public partial class Shape
     {
           partial void Create()
           {
             Console.WriteLine("Creating Shape");
           }
           public void Test()
           {
              Create();
           }
         }
         class Program
         {
           static void Main(String[] args)
           {
              Shape s = new Shape();
              s.Test();
           }
     }
}
```

- By separating the definition and implementation into two files, it is possible that two developers can work on them or even use a code-generator tool to create the definition of the method.
- Also, it is upto the developer whether to implement the partial method or not.
- It is also valid to have both the signature and implementation of **`Create()`** in the same part of **`Shape`**.

- The following figure demonstrates how you can define and implement a method in a single file:

Snippet

```
namespace PartialTest
{
/// <summary>
/// Class Shape is a partial class and contains the definition and
/// implementation of a partial method.
/// </summary>
public partial class Shape {
   partial void Create();
    . . .
    . . .
   partial void Create() {
     Console.WriteLine("Creating Shape");
   }
   public void Test() {
     Create();
   }
}
class Program {
   static void Main(String[] args) {
     Shape s = new Shape();
     s.Test();
   }
}
}
```

- It is possible to have only the signature of **`Create()`** in one part of **`Shape`** and no implementation of **`Create()`** anywhere.
- In that case, the compiler removes all references to **`Create()`**, including any method calls.
- A partial method must always include the `partial` keyword.
- Partial methods can be defined only within a partial class or type.
- If the class containing the definition or implementation of a partial method does not have the `partial` keyword, then a compile-time error would be raised.

- Some of the restrictions when working with partial methods are as follows:
  - The `partial` keyword is a must when defining or implementing a partial method
  - Partial methods must return `void`
  - They are implicitly `private`
  - Partial methods can return `ref` but not `out`
  - Partial methods cannot have any access modifier such as `public`, `private`, and so forth, or keywords such as `virtual`, `abstract`, `sealed`, or so forth
- Partial methods are useful when you have part of the code auto-generated by a tool or IDE and want to customize the other parts of the code.

Using Partial Types

- A large project in an organization involves creation of multiple structures, classes, and interfaces.
- If these types are stored in a single file, their modification and maintenance becomes very difficult.
- In addition, multiple programmers working on the project cannot use the file at the same time for modification.
- Thus, partial types can be used to split a type over separate files, allowing the programmers to work on them simultaneously.
- Partial types are also used with the code generator in Visual Studio 2012.

Inheriting Partial Classes 1-3

- You can add the auto-generated code into your file without recreation of the source file.
- You can use partial types for both these codes.
- A partial class can be inherited just like any other class in C#.
- It can contain virtual methods defined in different files which can be overridden in its derived classes.
- In addition, a partial class can be declared as an abstract class using the `abstract` keyword.
- Abstract partial classes can be inherited.
- The following code demonstrate how to inherit a partial class:

Snippet

```
//The following code is stored in Geometry.cs file
using System;
abstract partial class Geometry
{
   public abstract double Area(double val);
}
```

Inheriting Partial Classes 2-3

Snippet

```
//The following code is stored in Cube.cs file
using System;
abstract partial class Geometry
{
        public virtual void Volume(double val)
        {
        }
    }
    class Cube : Geometry
    {
       public override double Area (double side)
       {
          return 6 * (side * side);
       }
       public override void Volume(double side)
       {
          Console.WriteLine("Volume of cube: " + (side * side));
       }
       static void Main(string[] args)
       {
          double number = 20.56;
          Cube objCube = new Cube();
          Console.WriteLine ("Area of Cube: " +
          objCube.Area(number));
          objCube.Volume(number);
       }
}
```

- In the code:
  - The abstract partial class **Geometry** is defined across two C# files.
  - It defines an abstract method called **Area()** and a virtual method called **Volume()**.
  - Both these methods are inherited in the derived class called **Cube**.

Output

```
Area of Cube: 2536.2816
 Volume of cube: 422.7136
```

Nullable Types

- C# provides nullable types to identify and handle value type fields with null values.
- Before this feature was introduced, only reference types could be directly assigned null values.
- Value type variables with null values were indicated either by using a special value or an additional variable.
- This additional variable indicated whether or not the required variable was null.
- Special values are only beneficial if the decided value is followed consistently across applications.
- Creating and managing additional fields for such variables leads to more memory space and becomes tedious.
- These problems are solved by the introduction of nullable types.
- A nullable type is a means by which null values can be defined for the value types.
- It indicates that a variable can have the value `null`.
- Nullable types are instances of the `System.Nullable<T>` structure.
- A variable can be made `nullable` by adding a question mark following the data type.
- Alternatively, it can be declared using the generic `Nullable<T>` structure present in the `System` namespace.

- Nullable types in C# have the following characteristics:
  - They represent a value type that can be assigned a null value.
  - They allow values to be assigned in the same way as that of the normal value types.
  - They return the assigned or default values for nullable types.
  - When a nullable type is being assigned to a non-nullable type and the assigned or default value has to be applied, the ?? operator is used.

Implementing Nullable Types 1-3

- A nullable type can include any range of values that is valid for the data type to which the nullable type belongs.
- For example, a bool type that is declared as a nullable type can be assigned the values true, false, or null.
- Nullable types have two public read-only properties that can be implemented to check the validity of nullable types and to retrieve their values.

These are as follows:

- **The `HasValue` property**: `HasValue` is a `bool` property that determines validity of the value in a variable. The `HasValue` property returns a true if the value of the variable is `not null`, else it returns false.
- **The `Value` property:** The `Value` property identifies the value in a nullable variable. When the `HasValue` evaluates to true, the `Value` property returns the value of the variable, otherwise it returns an exception.

- The following code displays the employee's name, ID, and role using the nullable types:

Snippet

```
using System;
class Employee
{
     static void Main(string[] args)
     {
        int empId = 10;
        string empName = "Patrick";
        char? role = null;
        Console.WriteLine("Employee ID: " + empId);
        Console.WriteLine("Employee Name: " + empName);
        if (role.HasValue == true)
        {
            Console.WriteLine("Role: " + role.Value);
        }
        else
        {
            Console.WriteLine("Role: null");
        }
     }
}
```

- In the code:
  - **`EmpId`** is declared as an integer variable and it is initialized to value 10 and **`empName`** is declared as a `string` variable and it is assigned the name `Patrick`.
  - Additionally, `role` is defined as a nullable character with `null` value.
- The output displays the role of the employee as `null`.

Output

```
Employee ID: 10
Employee Name: Patrick
Role: null
```

Nullable Types in Expressions 1-2

- C# allows you to use nullable types in expressions that can result in a `null` value.
- Thus, an expression can contain both, nullable types and non-nullable types.
- An expression consisting of both, the nullable and non-nullable types, results in the value `null`.
- The following code demonstrates the use of nullable types in expressions:

Snippet

```
using System;

class Numbers
{
     static void Main (string[] args)
     {
          System.Nullable<int> numOne = 10;
          System.Nullable<int> numTwo = null;
          System.Nullable<int> result = numOne + numTwo;
          if (result.HasValue == true)
          {
              Console.WriteLine("Result: " + result);
          }
          else
          {
              Console.WriteLine("Result: null");
          }
     }
}
```

- In the code:
  - **numOne** and **numTwo** are declared as integer variables and initialized to values 10 and null respectively.
  - In addition, **result** is declared as an integer variable and initialized to a value which is the sum of **numOne** and **numTwo**.
  - The result of this sum is a null value and this is indicated in the output.

Output

```
Result: null
```

The ?? Operator 1-2

- A nullable type can either have a defined value or the value can be undefined.
- If a nullable type contains a null value and you assign this nullable type to a non-nullable type, the complier generates an exception called `System.InvalidOperationException`.
- To avoid this problem, you can specify a default value for the nullable type that can be assigned to a non-nullable type using the `??` operator.
- If the nullable type contains a null value, the `??` operator returns the default value.
- The following code demonstrates the use of `??` operator:

Snippet

```
using System;

class Salary{
    static void Main(string[] args) {
        double? actualValue = null;
        double marketValue = actualValue ?? 0.0;
        actualValue = 100.20;
        Console.WriteLine("Value: " + actualValue);
        Console.WriteLine("Market Value: " + marketValue);
    }
}
```

- In the code:
  - The variable **`actualValue`** is declared as double with a `?` symbol and initialized to value null.
  - This means that **`actualValue`** is now a nullable type with a value of `null`.
  - When it is assigned to `marketValue`, a `??` operator has been used.
  - This will assign `marketValue` the default value of 0.0.

Output

```
Value: 100.2
 Market Value: 0
```

Converting Nullable Types 1-4

- C# allows any value type to be converted into nullable type or a nullable type into a value type.
- C# supports two types of conversions on nullable types:
  - Implicit conversion
  - Explicit conversion
- The storing of a value type into a nullable type is referred to as implicit conversion.
- A variable to be declared as nullable type can be set to null using the `null` keyword.
- This is illustrated in the following code:

Snippet

```
using System;
class ImplicitConversion
{
      static void Main(string[] args)
      {
         int? numOne = null;
         if (numOne.HasValue == true)
         {
            Console.WriteLine("Value of numOne before
            conversion: " + numOne);
         }
         else
         {
           Console.WriteLine("Value of numOne: null");
         }
```

```
            numOne = 20;
            Console.WriteLine("Value of numOne after implicit
            conversion: " + numOne);
        }
}
```

- In the code:
  - The variable **numOne** is declared as nullable.
  - The `HasValue` property is being used to check whether the variable is of a null type.
  - Then, **numOne** is assigned the value 20, which is of `int` type stored in a nullable type.
  - This is implicit conversion.

Output

```
Value of numOne: null
Value of numOne after implicit conversion: 20
```

Converting Nullable Types 3-4

- The conversion of a nullable type to a value type is referred to as explicit conversion.
- This is illustrated in the following code:

Snippet

```
using System;
class ExplicitConversion
{
     static void Main(string[] args)
     {
          int? numOne = null;
          int numTwo = 20;
          int? resultOne = numOne + numTwo;
          if (resultOne.HasValue == true)
          {
               Console.WriteLine("Value of resultOne before conversion: "
+ resultOne);
          }
          else
          {
              Console.WriteLine("Value of resultOne: null");
          }
          numOne = 10;
          int result = (int)(numOne + numTwo);
          Console.WriteLine("Value of result after implicit
          conversion: " + result);
     }
}
```

- In the code:
  - The **numOne** and **resultOne** variables are declared as null.
  - The `HasValue` property is being used to check whether the **resultOne** variable is of a null type.
  - Then, **numOne** is assigned the value 10, which is of `int` type stored in a nullable type.
  - The values in both the variables are added and the result is stored in the **result** variable of `int` type.
  - This is explicit conversion.

Output

```
Value of resultOne: null
Value of resultTwo after explicit conversion: 30
```

Boxing Nullable Types 1-2

- An instance of the `object` type can be created as a nullable type that can hold both null and non-null values.
- The instance can be boxed only if it holds a non-null value and the `HasValue` property returns true.
- In this case, only the data type of the nullable variable is converted to type `object`.
- While boxing, if the `HasValue` property returns false, the object is assigned a null value.
- The following code demonstrates how to box nullable types:

Snippet

```
using System;
class Boxing
{
     static void Main(string[] args)
     {
          int? number = null;
          object objOne = number;
          if (objOne != null)
          {
               Console.WriteLine("Value of object one: " +
               objOne);
          }
          else
          {
               Console.WriteLine("Value of object one: null");
          }
```

Boxing Nullable Types 2-2

```
            double? value = 10.26;
            object objTwo = value;
            if (objTwo != null)
            {
                Console.WriteLine("Value of object two: " +
                objTwo);
            }
            else
            {
                Console.WriteLine("Value of object two: null");
            }
        }
}
```

- In the code:
  - The **number** variable declared as nullable is boxed and its value is stored in **objOne** as null.
  - The **value** variable declared as nullable is boxed and its value is stored in **objTwo** as 10.26.

Output

```
Value of object one: null
Value of object two: 10.26
```

Summary

- Anonymous methods allow you to pass a block of unnamed code as a parameter to a delegate.
- Extension methods allow you to extend different types with additional static methods.
- You can create an instance of a class without having to write code for the class beforehand by using a new feature called anonymous types.
- Partial types allow you to split the definitions of classes, structs, and interfaces to store them in different C# files.
- You can define partial types using the partial keyword.
- Nullable types allow you to assign null values to the value types.
- Nullable types provide two public read-only properties, HasValue and Value.