C# - Anonymous Type

Anonymous type, as the name suggests, is a type that doesn't have any name. **C# allows you to create an object with the *new* keyword without defining its class. The implicitly typed variable- var is used to hold the reference of anonymous types**.

Example: Anonymous Type

```
var myAnonymousType = new { firstProperty = "First",
    secondProperty = 2,
    thirdProperty = true
};
```

In the above example, myAnonymousType is an object of anonymous type created using the new keyword and object initializer syntax. It includes three properties of different data types.

An anonymous type is a temporary data type that is inferred based on the data that you include in an object initializer. Properties of anonymous types will be **read-only** properties so you cannot change their values.

Internally, the compiler automatically generates the new type for anonymous types. You can check the type of an anonymous type as shown below.

```
static void Main(string[] args)
{
    var myAnonymousType = new { firstProperty = "First",
                               secondProperty = 2,
                               thirdProperty = true
                           };

    Console.WriteLine(myAnonymousType.GetType().ToString());
}
```

Output

<>f__AnonymousType0'3[System.String,System.Int32,System.Boolean]

Scope of Anonymous Type

An anonymous type will always be local to the method where it is defined. **Usually, you cannot pass an anonymous type to another method; however, you can pass it to a method that accepts a parameter of dynamic type. Please note that Passing anonymous types using dynamic is not recommended.**

Example: Passing Anonymous Type

```csharp
static void Main(string[] args)
{

    var myAnonymousType = new
                          {
                              firstProperty = "First Property",
                              secondProperty = 2,
                              thirdProperty = true
                          };

    DoSomethig(myAnonymousType);
}

static void DoSomethig(dynamic param)
{
    Console.WriteLine(param.firstProperty);
}
```

Output

First Property

C# - Dynamic Type

C# 4.0 (.NET 4.5) introduced a new type that avoids compile time type checking. You have learned about the implicitly typed variable- **var in the previous section where the compiler assigns a specific type based on the value of the expression. A dynamic type escapes type checking at compile time; instead, it resolves type at run time**.

A dynamic type can be defined using the `dynamic` keyword.

Example: dynamic type variable

```csharp
dynamic dynamicVariable = 1;
```

The compiler compiles dynamic types into object types in most cases. The above statement would be compiled as:

dynamic type at compile time:

```csharp
object dynamicVariable = 1;
```

The actual type of dynamic would resolve at runtime. You can check the
type of the dynamic variable, as below:

Example: Get the actual type of dynamic type at runtime

```
static void Main(string[] args)
{
    dynamic dynamicVariable = 1;

    Console.WriteLine(dynamicVariable.GetType().ToString());
}
```

Output:

System.Int32

**A dynamic type changes its type at runtime based on the value of the
expression to the right of the "=" operator**. The following example shows
how a dynamic variable changes its type based on its value:

Example: dynamic

```
static void Main(string[] args)
{
    dynamic dynamicVariable = 100;
    Console.WriteLine("Dynamic  variable  value:  {0},  Type:  {1}",dynamicVariable,
dynamicVariable.GetType().ToString());

    dynamicVariable = "Hello World!!";
    Console.WriteLine("Dynamic  variable  value: {0},  Type: {1}",  dynamicVariable,
dynamicVariable.GetType().ToString());

    dynamicVariable = true;
    Console.WriteLine("Dynamic  variable  value:  {0},  Type: {1}",  dynamicVariable,
dynamicVariable.GetType().ToString());

    dynamicVariable = DateTime.Now;
    Console.WriteLine("Dynamic  variable  value:  {0},  Type: {1}",  dynamicVariable,
dynamicVariable.GetType().ToString());
}
```

Output:

Dynamic variable value: 100, Type: System.Int32
Dynamic variable value: Hello World!!, Type: System.String

Dynamic variable value: True, Type: System.Boolean
Dynamic variable value: 01-01-2014, Type: System.DateTime

Methods and Properties of Dynamic Type

If you assign class object to the dynamic type then the compiler would not check for correct methods and properties name of a dynamic type that holds the custom class object. Consider the following example.

Example: dynamic

```csharp
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int Age { get; set; }
    public int StandardID { get; set; }

    public void DisplayStudentDetail()
    {
        Console.WriteLine("Name: {0}", this.StudentName);
        Console.WriteLine("Age: {0}", this.Age);
        Console.WriteLine("Standard: {0}", this.StandardID);
    }
}

class Program
{
    static void Main(string[] args)
    {
        dynamic dynamicStudent = new Student();

        dynamicStudent.FakeMethod();
    }
}
```

In the above example, we have assigned Student object to a dynamic variable. In the second statement in Main() method, we call FakeMethod() method, which is not exists in the Student class. However, the compiler will not give any error for FakeMethod() because it skips type checking for dynamic type, instead you will get a runtime exception.

Dynamic Type as a Method Parameter

A method can have dynamic type parameters so that it can accept any type of parameter at run time.

Example: dynamic as a Parameter

```csharp
class Program
{
    static void PrintValue(dynamic val)
    {
        Console.WriteLine(val);
    }

    static void Main(string[] args)
    {

        PrintValue("Hello World!!");
        PrintValue(100);
        PrintValue(100.50);
        PrintValue(true);
        PrintValue(DateTime.Now);
    }
}
```

Output:

Hello World!!
100
100.50
True
01-01-2014 10:10:50

Points to Remember :

1. The dynamic types are resolved at runtime instead of compile time.
2. The compiler skips the type checking for dynamic type. So it doesn't give any error about dynamic types at compile time.
3. The dynamic types do not have intellisense support in visual studio.
4. A method can have parameters of the dynamic type.
5. An exception is thrown at runtime if a method or property is not compatible.

Nullable Type in C#

As you know, a value type cannot be assigned a null value. For example, *int i = null* will give you a compile time error.

C# 2.0 introduced nullable types that allow you to assign null to value type variables. You can declare nullable types using `Nullable<t>` where T is a type.

Example: Nullable type

```csharp
Nullable<int> i = null;
```

A nullable type can represent the correct range of values for its underlying value type, plus an additional *null* value. For example, Nullable<int> can be assigned any value from -2147483648 to 2147483647, or a null value.

The Nullable types are instances of `System.Nullable<T>` struct. Think it as something like the following structure.

Example: Nullable struct

```csharp
[Serializable]
public struct Nullable<T> where T : struct
{
    public bool HasValue { get; }

    public T Value { get; }

    // other implementation
}
```

A nullable of type *int* is the same as an ordinary *int* plus a flag that says whether the *int* has a value or not (is null or not). All the rest is compiler magic that treats "null" as a valid value.

Example: HasValue

```csharp
static void Main(string[] args)
{
    Nullable<int> i = null;

    if (i.HasValue)
        Console.WriteLine(i.Value); // or Console.WriteLine(i)
    else
        Console.WriteLine("Null");
}
```


Output:

Null

The `HasValue` returns **true** if the object has been assigned a value; if it has not been assigned any value or has been assigned a null value, it will return **false**.

Accessing the value using NullableType.value will throw a runtime exception if nullable type is null or not assigned any value.

Accessing the value using NullableType.value will throw a runtime exception if nullable type is null or not assigned any value. For example, i.Value will throw an exception if i is null:

```
static void Main(string[] args)
{
    Nullable<int> i = null;          ❌  InvalidOperationException – run time

    Console.WriteLine(i.Value);

}
```

Use the GetValueOrDefault() method to get an actual value if it is not null and the default value if it is null. For example:

Example: GetValueOrDefault()

```
static void Main(string[] args)
{
    Nullable<int> i = null;

    Console.WriteLine(i.GetValueOrDefault());
}
```

Output

0

Shorthand Syntax for Nullable Types

You can use the '?' operator to shorthand the syntax e.g. int?, long? instead of using Nullable<T>.

Example: Shorthand syntax for Nullable types

```
int? i = null;
double? D = null;
```

?? Operator

Use the '??' operator to assign a nullable type to a non-nullable type.

Example: ?? operator with Nullable Type

```
int? i = null;

int j = i ?? 0;

Console.WriteLine(j);
```

<u>Output</u>:

0

In the above example, i is a nullable int and if you assign it to the non-nullable int j then it will throw a runtime exception if i is null. So to mitigate the risk of an exception, we have used the '??' operator to specify that if i is null then assign 0 to j.

Assignment Rules

A nullable type has the same assignment rules as a value type. It must be assigned a value before using it if nullable types are declared in a function as local variables. If it is a field of any class then it will have a null value by default.

For example, the following nullable of int type is declared and used without assigning any value. The compiler will give **"Use of unassigned local variable 'i'"** error:

```
static void Main(string[] args)
{
    Nullable<int> i;

    Console.WriteLine(i);
}
```

✗ Use of unassigned local variable 'i'

In the following example, a nullable of int type is a field of the class, so it will not give any error.

In the following example, a nullable of int type is a field of the class, so it will not give any error.

Example: Nullable type as Class Field

```
class MyClass
{
    public Nullable<int> i;
}



class Program
{
    static void Main(string[] args)
    {
        MyClass mycls = new MyClass();

        if(mycls.i == null)
            Console.WriteLine("Null");
    }
}
```

Output:

Null

Covariance and Contravariance in C#

Covariance and contravariance allow us to be flexible when dealing with class hierarchy.

Consider the following class hierarchy before we learn about covariance and contravariance:

Example: Class Hierarchy

```
public class Small
{

}
public class Big: Small
{

}
public class Bigger : Big
{

}
```

As per the above example classes, small is a base class for big and big is a base class for bigger. The point to remember here is that a derived class will always have something more than a base class, so the base class is relatively smaller than the derived class.

Now, consider the following initialization:

Class initialization

As you can see above, **a base class can hold a derived class but a derived class cannot hold a base class**. **In other word, an instance can accept big even if it demands small, but it cannot accept small if it demands big.**

Now, let's learn about covariance and contravariance.

Covariance in C#

Covariance allows you to use a derived class where a base class is expected (rule: can accept big if small is expected).

Covariance can be applied on delegate, generic, array, interface, etc.

Covariance with Delegate

Covariance in delegates allows flexiblity in the return type of delegate methods.

Example: Covariance with Delegate

```
public delegate Small covarDel(Big mc);

class Program
{

    static Big Method1(Big bg)
    {
        Console.WriteLine("Method1");

        return new Big();
    }
```

```csharp
    static Small Method2(Big bg)
    {
        Console.WriteLine("Method2");

        return new Small();
    }

    static void Main(string[] args)
    {
        covarDel del = Method1;

        Small sm1 = del(new Big());

        del= Method2;
        Small sm1 = del(new Big());
    }
}
```

Output:

Method1
Method2

As you can see in the above example, delegate expects a return type of small (base class) but we can still assign Method1 that returns Big (derived class) and also Method2 that has same signature as delegate expects.

Thus, covariance allows you to assign a method to the delegate that has a less derived return type.

C# Contravariance

Contravariane is applied to parameters. Contravariance allows a method with the parameter of a base class to be assigned to a delegate that expects the parameter of a derived class.

Continuing with the example above, add Method3 that has a different parameter type than delegate:

Example: Contravariance with Delegte

```csharp
delegate Small covarDel(Big mc);

class Program
{
    static Big Method1(Big bg)
    {
        Console.WriteLine("Method1");
        return new Big();
    }
```

```csharp
    static Small Method2(Big bg)
    {
        Console.WriteLine("Method2");
        return new Small();
    }

    static Small Method3(Small sml)
    {
        Console.WriteLine("Method3");

        return new Small();
    }
    static void Main(string[] args)
    {
        covarDel del = Method1;
        del += Method2;
        del += Method3;

        Small sm = del(new Big());
    }
}
```

Output:

Method1
Method2
Method3

As you can see, Method3 has a parameter of Small class whereas delegate expects a parameter of Big class. Still, you can use Method3 with the delegate.

You can also use covariance and contravariance in the same method as shown below.

Example: Covariance and Contravariance

```csharp
delegate Small covarDel(Big mc);

class Program
{

    static Big Method4(Small sml)
    {
        Console.WriteLine("Method3");

        return new Big();
    }

    static void Main(string[] args)
    {
        covarDel del = Method4;

        Small sm = del(new Big());
    }
```

```
}
```

:

Method4

C# - Func

A delegates can be defined as shown below.

Example: C# Delegate

```
public delegate int SomeOperation(int i, int j);

class Program
{
    static int Sum(int x, int y)
    {
        return x + y;
    }

    static void Main(string[] args)
    {
        SomeOperation add = Sum;

        int result = add(10, 10);

        Console.WriteLine(result);
    }
}
```
Output
20

C# 3.0 includes built-in generic delegate types `Func` and `Action`, so that you don't need to define custom delegates as above.

Func is a generic delegate included in the `System` namespace. It has zero or more *input* parameters and one *out* parameter. The last parameter is considered as an out parameter.

Signature: Func

```
namespace System
{
    public delegate TResult Func<in T, out TResult>(T arg);
```

```
}
```

The last parameter in the angle brackets <> is considered as the return type and remaining parameters are considered as input parameter types as shown in the following figure.

The following `Func` type delegate is the same as the above `SomeOperation` delegate, where it takes two input parameters of int type and returns a value of int type:

```
Func<int, int, int> sum;
```

You can assign any method to the above func delegate that takes two *int* parameters and returns an *int* value. Now, you can take Func delegate instead of someOperation delegate in the first example.

Example: Func

```
class Program
{
    static int Sum(int x, int y)
    {
        return x + y;
    }

    static void Main(string[] args)
    {
        Func<int,int, int> add = Sum;

        int result = add(10, 10);

        Console.WriteLine(result);
    }
}
```

Example: Func with Zero Input Parameter

```
Func<int> getRandomNumber;
```

C# Func with an Anonymous Method

You can assign an anonymous method to the Func delegate by using the delegate keyword.

Example: Func with Anonymous Method

```csharp
Func<int> getRandomNumber = delegate()
                            {
                                Random rnd = new Random();
                                return rnd.Next(1, 100);
                            };
```

Func with Lambda Expression

A Func delegate can also be used with a lambda expression, as shown below:

Example: Func with lambda expression

```csharp
Func<int> getRandomNumber = () => new Random().Next(1, 100);

//Or

Func<int, int, int>  Sum  = (x, y) => x + y;
```

Points to Remember :

1. Func is built-in delegate type.
2. Func delegate type must return a value.
3. Func delegate type can have zero to 16 input parameters.
4. Func delegate does not allow ref and out parameters.
5. Func delegate type can be used with an anonymous method or lambda expression

C# - Action Delegate

Action is also a delegate type defined in the System namespace. An Action type delegate is the same as Func delegate except that the Action delegate doesn't return a value. In other words, an Action delegate can be used with a method that has a void return type.

For example, the following delegate prints an int value.

Example: C# Delegate

```csharp
public delegate void Print(int val);

static void ConsolePrint(int i)
{
```

```
    Console.WriteLine(i);
}

static void Main(string[] args)
{
    Print prnt = ConsolePrint;
    Prnt(10);
}
```

<u>Output</u>:

```
10
```

You can use an Action delegate instead of defining the above Print delegate, for example:

Example: Action delegate

```
static void ConsolePrint(int i)
{
    Console.WriteLine(i);
}

static void Main(string[] args)
{
    Action<int> printActionDel = ConsolePrint;
    printActionDel(10);
}
```

You can initialize an Action delegate using the new keyword or by directly assigning a method:

```
Action<int> printActionDel = ConsolePrint;

//Or

Action<int> printActionDel = new Action<int>(ConsolePrint);
```
An Action delegate can take up to 16 input parameters of different types.

An Anonymous method can also be assigned to an Action delegate, for example:

Example: Anonymous method with Action delegate

```
static void Main(string[] args)
{
    Action<int> printActionDel = delegate(int i)
                                {
                                    Console.WriteLine(i);
                                };
```

```
    printActionDel(10);
}
```

Output:

10

A Lambda expression also can be used with an Action delegate:

Example: Lambda expression with Action delegate

```csharp
static void Main(string[] args)
{

    Action<int> printActionDel = i => Console.WriteLine(i);

    printActionDel(10);
}
```

Advantages of Action and Func Delegates

1. Easy and quick to define delegates.
2. Makes code short.
3. Compatible type throughout the application.

Points to Remember :

1. Action delegate is same as func delegate except that it does not return anything. Return type must be void.
2. Action delegate can have 1 to 16 input parameters.
3. Action delegate can be used with anonymous methods or lambda expressions.

C# - Partial Class

Each class in C# resides in a separate physical file with a .cs extension. C# provides the ability to have a single class implementation in multiple .cs files using the *partial* modifier keyword. The *partial* modifier can be applied to a class, method, interface or structure.

For example, the following MyPartialClass splits into two files, PartialClassFile1.cs and PartialClassFile2.cs:

Example: PartialClassFile1.cs

```csharp
public partial class MyPartialClass
{
    public MyPartialClass()
    {
    }

    public void Method1(int val)
    {
        Console.WriteLine(val);
    }
}
```

Example: PartialClassFile2.cs

```csharp
public partial class MyPartialClass
{
    public void Method2(int val)
    {
        Console.WriteLine(val);
    }
}
```

MyPartialClass in PartialClassFile1.cs defines the constructor and one public method, Method1, whereas PartialClassFile2 has only one public method, Method2. The compiler combines these two partial classes into one class as below:

Example: Partial class

```csharp
public class MyPartialClass
{
    public MyPartialClass()
    {
    }

    public void Method1(int val)
    {
        Console.WriteLine(val);
    }

    public void Method2(int val)
    {
        Console.WriteLine(val);
    }
}
```

Partial Class Requirements:

- All the partial class definitions must be in the same assembly and namespace.
- All the parts must have the same accessibility like public or private, etc.

- If any part is declared abstract, sealed or base type then the whole class is declared of the same type.
- Different parts can have different base types and so the final class will inherit all the base types.
- The Partial modifier can only appear immediately before the keywords class, struct, or interface.
- Nested partial types are allowed.

Advantages of Partial Class

- Multiple developers can work simultaneously with a single class in separate files.
- When working with automatically generated source, code can be added to the class without having to recreate the source file. For example, Visual Studio separates HTML code for the UI and server side code into two separate files: .aspx and .cs files.

Partial Methods

A partial class or struct may contain partial methods. A partial method must be declared in one of the partial classes. A partial method may or may not have an implementation. If the partial method doesn't have an implementation in any part then the compiler will not generate that method in the final class. For example, consider the following partial method with a partial keyword:

Example: PartialClassFile1.cs:

```csharp
public partial class MyPartialClass
{
    partial void PartialMethod(int val);

    public MyPartialClass()
    {

    }

    public void Method2(int val)
    {
        Console.WriteLine(val);
    }
}
```

Example: PartialClassFile2.cs:

```csharp
public partial class MyPartialClass
{
    public void Method1(int val)
    {
        Console.WriteLine(val);
```

```
    }

    partial void PartialMethod(int val)
    {
        Console.WriteLine(val);
    }
}
```
PartialClassFile1.cs contains the declaration of the partial method and PartialClassFile2.cs contains the implementation of the partial method.

Requirements for Partial Method

- The partial method declaration must began with the partial modifier.
- The partial method can have a ref but not an out parameter.
- Partial methods are implicitly private methods.
- Partial methods can be static methods.
- Partial methods can be generic.

Points to Remember :

1. Use the partial keyword to split interface, class, method or structure into multiple .cs files.
2. The partial method must be declared before implementation.
3. All the partial class, method , interface or structs must have the same access modifiers.

C# - Extension Method

Extension methods, as the name suggests, are additional methods. Extension methods allow you to inject additional methods without modifying, deriving or recompiling the original class, struct or interface. Extension methods can be added to your own custom class, .NET framework classes, or third party classes or interfaces.

In the following example, IsGreaterThan() is an extension method for int type, which returns true if the value of the int variable is greater than the supplied integer parameter.
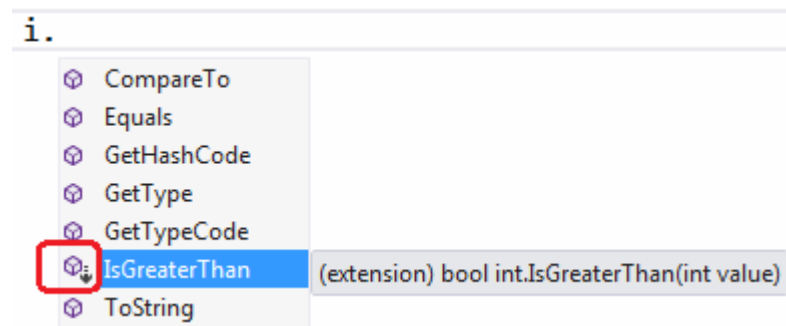
Example: Extension Method

```
int i = 10;

bool result = i.IsGreaterThan(100); //returns false
```

The `IsGreaterThan()` method is not a method of int data type (Int32 struct). It is an extension method written by the programmer for the int data type. The IsGreaterThan() extension method will be available throughout the application by including the namespace in which it has been defined.

The extension methods have a special symbol in intellisense of the visual studio, so that you can easily differentiate between class methods and extension methods.

Extension Method Symbol in visual studio intellisense

Now let's see how to write an extension method.

An extension method is actually a special kind of static method defined in a static class. To define an extension method, first of all, define a static class.

For example, we have created an `IntExtensions` class under the `ExtensionMethods` namespace in the following example. The IntExtensions class will contain all the extension methods applicable to int data type. (You may use any name for namespace and class.)

Example: Create a Class for Extension Methods

```
namespace ExtensionMethods
{
    public static class IntExtensions
    {

    }
}
```

Now, define a static method as an extension method where the first parameter of the extension method specifies the type on which the extension method is applicable. We are going to use this extension method on int type. So the first parameter must be int preceded with the ***this*** modifier.

For example, the `IsGreaterThan()` method operates on int, so the first parameter would be, `this int i`.

Example: Define an Extension Method

```
namespace ExtensionMethods
{
    public static class IntExtensions
     {
        public static bool IsGreaterThan(this int i, int value)
        {
            return i > value;
        }
    }
}
```

Now, you can include the ExtensionMethods namespace wherever you want to use this extension method.

Example: Extension method

```
using ExtensionMethods;

class Program
{
    static void Main(string[] args)
    {
        int i = 10;

        bool result = i.IsGreaterThan(100);

        Console.WriteLine(result);
    }
}
```
Output:

false

 Note:

The only difference between a regular static method and an extension method is that the first parameter of the extension method specifies the type that it is going to operator on, preceded by the this keyword.

Points to Remember :

1. Extension methods are additional custom methods which were originally not included with the class.
2. Extension methods can be added to custom, .NET Framework or third party classes, structs or interfaces.

3. The first parameter of the extension method must be of the type for which the extension method is applibable, preceded by the this keyword.
4. Extension methods can be used anywhere in the application by including the namespace of the extension method.