Constructor and Object Initializers

The constructors in C# are the special types of methods of a class that automatically executed whenever we create an instance (object) of that class. The Constructors are responsible for two things. One is the object initialization and the other one is memory allocation. The role of the new keyword is to create the object.

The constructor name should be the same as the class name. It should not contain return type even void also. The constructor should not contain modifiers. As part of the constructor body, we can place return in the constructor but return statement with value is not allowed.

Types of constructors

There are five types of constructors available in C#:

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor
4. Static Constructor
5. Private Constructor

Default Constructor

The Constructor without parameter is called a default constructor. Again the default constructor is classified into two types: System-defined and User-defined

As a programmer, if you are not defined any constructor explicitly in your program, then by default the system will provide one constructor at the time of compilation. That constructor is called a default constructor. The default constructor will assign default values to the data members (non-static variables). System will only provide the default constructor if as a programmer you are not defined any constructor explicitly.

Parameterized Constructor

The developer given constructor with parameters is called as the parameterized constructor in C#. With the help of a Parameterized constructor, we can initialize each instance of the class with different values. That means using parameterized constructor we can store a different set of values into different objects created to the class.

Copy Constructor

The constructor which takes a parameter of the class type is called a copy constructor. This constructor is used to copy one object data into another object. The main purpose of the copy constructor is to initialize a new object (instance) with the values of an existing object (instance).

Static Constructor

It is also possible to create a constructor as static. The static constructor in C# will be invoked only once. There is no matter how many numbers of instances (objects) of the class are created, it is going to be invoked only once and that is during the creation of the first instance (object) of the class. The static constructor is used to initialize the static fields of the class. You can also write some code inside the static constructor which is going to be executed only once. The static data members in C# are created only once even though we created any number of objects.

Points to Remember while creating Static Constructor in C#:

- There can be only one static constructor in a class.
- The static constructor should be without any parameter.
- It can only access the static members of the class.
- There should not be any access modifier in static constructor definition.
- Static constructor will be invoked only once i.e. at the time of first object creation of the class, from 2nd object creation onwards static constructor will not be called.

Private Constructor

In C#, it is also possible to create a constructor as private. When a class contains a private constructor then we cannot create an object for the class outside of the class. So, private constructors are used to creating an object for the class within the same class. Generally, private constructors are used in Remoting concept.

Example:

```
public class Counter
{
    private Counter() { }
    public static int currentCount;
    public static int IncrementCount()
    {
        return ++currentCount;
    }
}
class TestCounter
{
    static void Main()
    {
        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        // Counter aCounter = new Counter();   // Error
        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine("New count: {0}", Counter.currentCount);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
```

```
    }
}
// Output: New count: 101
```

Constructor Overloading

When we define multiple constructors within a class with different parameter types, number and order then it is called as constructor overloading.

Example:

```
using System;
namespace constructorOverloadingDemo
{
      public class Person
      {
         private string last;
         private string first;
         // User defined default constructor
         public Person()
         {
            lastName = "Rangrej";
            firstName = "Dhaval";
         }
         // Parameterized constructor
         public Person(string lastName, string firstName)
         {
            last = lastName;
            first = firstName;
         }
         // Copy constructor
         public Person(Person prevPerson)
         {
            lastName = prevPerson.lastName;
            firstName = prevPerson.firstName;
         }

      }
      public class Adult : Person
      {
         private static int minimumAge;

         public Adult(string lastName, string firstName) : base(lastName,
      firstName)
         { }
         // Static constructor
         static Adult()
         {
            minimumAge = 18;
         }
      }
```

```
        class Program
        {
            static void Main()
            {
                Person defaultPerson = new Person();
                Person p1 = new Person("Shah", "Raj");
                Person p2 = new Person(defaultPerson);
                Person p3 = new Adult("Patel", "Raj");
                // Keep the console window open in debug mode.
                Console.WriteLine($"Default person: {defaultPerson.firstName}
        {defaultPerson.lastName}");
                Console.WriteLine($"Person - 1: {p1.firstName} {p1.lastName}");
                Console.WriteLine($"Copy of default Person: {p2.firstName}
        {p2.lastName}");
                Console.WriteLine($"Adult Person - 3: {p3.firstName} {p3.lastName}
        with minimum age: {Adult.minimumAge}");

                Console.WriteLine("Press any key to exit.");
                Console.ReadKey();
            }
        }
}
// Output:
//     Default person: Dhaval Rangrej
//     Person – 1: Raj Shah
//     Copy of default person: Dhaval Rangrej
//     Adult person – 3: Raj Patel with minimum age: 18
```

Object Initializers

Object initializers let you assign values to any accessible fields or properties of an object at creation time without having to invoke a constructor followed by lines of assignment statements. The object initializer syntax enables you to specify arguments for a constructor or omit the arguments (and parentheses syntax). The following example shows how to use an object initializer with a named type, Cat and how to invoke the parameter less constructor. Note the use of auto-implemented properties in the `Cat` class.

```
public class Cat
{
    // Auto-implemented properties.
    public int Age { get; set; }
    public string Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

```
...
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
...
```

The object initializers' syntax allows you to create an instance, and after that it assigns the newly created object, with its assigned properties, to the variable in the assignment. Starting with C# 6, object initializers can set indexers, in addition to assigning fields and properties. Consider this basic `Matrix` class:

```
public class Matrix
{
    private double[,] storage = new double[3, 3];

    public double this[int row, int column]
    {
        // The embedded array will throw out of range exceptions as appropriate.
        get { return storage[row, column]; }
        set { storage[row, column] = value; }
    }
}

. . .

var identity = new Matrix
{
    [0, 0] = 1.0,
    [0, 1] = 0.0,
    [0, 2] = 0.0,

    [1, 0] = 0.0,
    [1, 1] = 1.0,
    [1, 2] = 0.0,

    [2, 0] = 0.0,
    [2, 1] = 0.0,
    [2, 2] = 1.0,
};

. . .
```

Object Initializers with anonymous types

Although object initializers can be used in any context, they are especially useful in LINQ query expressions. You create anonymous types by using the `new` operator together with an object initializer. Query expressions make frequent use of anonymous types, which can only be initialized by using an object initializer, as shown in the following declaration.

```
var v = new { Amount = 108, Message = "Hello" };
// Rest the mouse pointer over v.Amount and v.Message in the following
// statement to verify that their inferred types are int and string.
Console.WriteLine(v.Amount + v.Message);
```

Note: Local variables can be declared without giving an explicit type. The **var** keyword instructs the compiler to infer the type of the variable from the expression on the right side of the initialization statement. The inferred type may be a built-in type, an anonymous type, a user-defined type, or a type defined in the .NET class library.

Anonymous Types

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first. The type name is generated by the compiler and is not available at the source code level. The type of each property is inferred by the compiler. Anonymous types typically are used in the `select` clause of a LINQ query expression to return a subset of the properties from each object in the source sequence.

Anonymous types contain one or more public read-only properties. No other kinds of class members, such as methods or events, are valid. The expression that is used to initialize a property cannot be null, an anonymous function, or a pointer type.

The most common scenario is to initialize an anonymous type with properties from another type. In the following example, assume that a class exists that is named `Product`. Class `Product` includes `Color` and `Price` properties, together with other properties that you are not interested in. Variable `products` is a collection of `Product` objects. The anonymous type declaration starts with the `new` keyword. The declaration initializes a new type that uses only two properties from `Product`. This causes a smaller amount of data to be returned in the query.

If you do not specify member names in the anonymous type, the compiler gives the anonymous type members the same name as the property being used to initialize them. You must provide a name for a property that is being initialized with an expression, as shown in the previous example. In the following example, the names of the properties of the anonymous type are `Color` and `Price`.

```
var productQuery =
    from prod in products
    select new { prod.Color, prod.Price };

foreach (var v in productQuery)
{
    Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);
}
```