Implicitly Typed Local Variables

Local variables can be given an inferred "type" of **var** instead of 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, a type defined in the .NET Framework class library, or any expression.

It is important to understand that the **var** keyword does not mean “Variant” and does not indicate that the variable is loosely typed, or late-bound. It just means that the compiler determines and assigns the most appropriate type.

The var keyword may be used in the following contexts:

- On local variables (variables declared at method scope)
- In a for initialization statement.
  For(var x=1;x<=10;x++)
- In a foreach initialization statement.
  Foreach(var x in A)

The following restrictions apply to implicitly-typed variable declarations:

- var can only be used when a local variable is declared and initialized in the same statement; the variable cannot be initialized to null.
- var cannot be used on fields at class scope.
- Variables declared by using var cannot be used in the initialization expression. In other words, var v = v++; produces a compile-time error.
- Multiple implicitly-typed variables cannot be initialized in the same statement.

The only scenario in which a local variable must be implicitly typed with var is in the initialization of anonymous types. Type of the variable declared with the keyword **var** will be determined by the value assigned to it.

Example : The following example creates three local variables with **var** keyword and their type is printed with gettype() method.

```
namespace AnonymousTypes
{
    class ImpliciteTypes
    {
        static void Main()
        {
            var A = 10;
            var D = 234.567;
            var S = "Naresh";
            Console.WriteLine("Type Of A Is {0}", A.GetType());
            Console.WriteLine("Type Of D Is {0}", D.GetType());
            Console.WriteLine("Type Of S Is {0}", S.GetType());
        }
    }
}
```

You can create an implicitly-typed array in which the type of the array instance is inferred from the elements specified in the array initializer. The rules for any implicitly-typed variable also apply to implicitly-typed arrays. Implicitly-typed arrays are usually used in query expressions together with anonymous types and object and collection initializers.

Example : The following example creates an implicitly typed array

```
namespace AnonymousTypes
{
    class ImplicitArray
    {
        static void Main()
        {
            var A = new[] { 10, 20, 30, 40, 50 };
        var S = new[] { "Microsoft", "Sun", "Adobe", "Oracle", "Ibm" };
     var B = new[,] { { 10, 20, 30 }, { 40, 50, 60 }, { 70, 80, 90 } };
            var JA = new[]
            {
                new[]{10,20,30},
                new[]{40,50,60,70},
                new[]{80,90,100,110,120}
            };
            foreach (var n in A)
            {
                Console.Write("{0}\t", n);
            }
            Console.WriteLine();
            Console.WriteLine();
            foreach (var n in S)
```

```
                {
                    Console.WriteLine(n);
                }
                Console.WriteLine();
                for (int i = 0; i < B.GetLength(0); i++)
                {
                    for (int j = 0; j < B.GetLength(1); j++)
                    {
                        Console.Write("{0}\t", B[i, j]);
                    }
                    Console.WriteLine();
                }
                Console.WriteLine();
                Console.WriteLine();
                for (int i = 0; i < 3; i++)
                {
                    for (int j = 0; j < JA[i].GetLength(0); j++)
                    {
                        Console.Write("{0}\t", JA[i][j]);
                    }
                    Console.WriteLine();
                }
            }
        }
    }
```

Object Initializer

Object initializer feature of .net 3.5 let you assign values to any accessible fields or properties of an object at creation time without having to explicitly invoke a constructor.

Example : The following example creates a class with the name Student with two properties Sid And Sname that are initialized without creating a constructor and by using object initializer feature of .net 3.5.

```
namespace AnonymousTypes
{
    class Student
    {
        int _Sid;
        string _Sname;
        public int Sid
        {
            get { return _Sid; }
            set { _Sid = value; }
        }
        public string Sname
        {
            get { return _Sname; }
            set { _Sname = value; }
        }
        public void Print()
        {
            Console.WriteLine("Sid : {0}\tSname : {1}", Sid, Sname);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Student S = new Student { Sid = 1001, Sname = "SAI" };
            S.Print();
        }
    }
                                  }
```

Anonymous Types

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to first explicitly define a type. The type name is generated by the compiler and is not available at the source code level. The type of the properties is inferred by the compiler. Anonymous types are created by using the **new** operator with an **object initializer**. Anonymous types are class types that consist of one or more public read-only properties. No other kinds of class members such as methods or events are allowed.

Example : The following example creates an anonymous type with three properties empid, ename and job.

```
namespace AnonymousTypes
{
    class AnonymType
    {
        static void Main()
        {
         var E = new { Empid = 1001, Ename = "SAI", Job = "ANALYST" };
         Console.WriteLine("{0}\t{1}\t{2}", E.Empid, E.Ename, E.Job);
        }
    }
}
```

Anonymous types are reference types that derive directly from object. The compiler gives them a name although your application cannot access it. From the perspective of the common language runtime, an anonymous type is no different from any other reference type.

If two or more anonymous types have the same number and type of properties in the same order, the compiler treats them as the same type and they share the same compiler-generated type information.