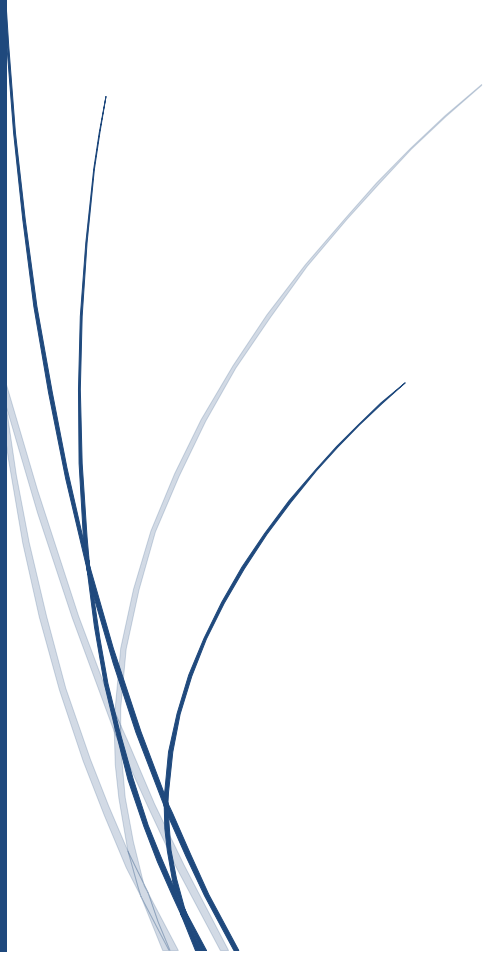
C#Corner

Garnet Valley PA.
Sept 05, 2014

Programing C# 5.0

By Mahesh Chand

Edited by Sam Hobbs
Chief Editor, C# Corner



ABOUT THE AUTHOR

Mahesh Chand founded C# Corner in 1999 as a hobby website to share his code. Today, C# Corner reaches over 3 Million users each month and has become one of the most popular online communities for developers.

Mahesh is a Software Solutions Architect and 9-time Microsoft MVP. Holding a Bachelor's degree in Mathematics and Physics and a Master's degree in Computer Science, Mahesh has written half a dozen books with publishers including Addison-Wesley and APress.

In his day job, Mahesh is a technical architect, startup advisor, and mentor. Some of the companies he has worked with includes Microsoft, J&J, Unisys, Adidas, Juniper, McGraw-Hill, and Exelon.

A Message from the Author

"C# Corner is a community with the main goal to learn, share and educate. You could help grow this community by telling your co-workers and share on your social media Twitter and Facebook accounts "

-Mahesh Chand

Introduction

The focus of this book is the C# language features that were introduced in versions 3.0, 4.0 and 5.0.

This book covers the following topics:

1. Partial classes
2. Automated Implemented Properties
3. Object Initializer
4. Collection Initializer
5. Type Interface
6. Anonymous Types
7. Delegates
8. Generic Delegates
9. Extension Methods
10. Dynamic Programming
11. Named and Optional Parameters
12. Covariance and Contravariance
13. Lambdas

Prerequisites

For this book, you should know C# language programming. This book can be assumed as Part 2 of my previous book that covers C# language versions 1.0, 1.1 and some 2.0 features. If you are new to C# programming, I highly recommend to download my previous book, Programming C# for Beginners which is available to download free on C# Corner Books section.

Partial Classes

In this article, we will understand the concept of partial classes in C#. We will also learn why we need partial classes followed by how to create a partial class and put it to work in a real application.

Introduction

The concept of partial classes was introduced in C# 2.0 and was later extended in C# 3.0.

It is very common for class library developers or architects to define and implement a significant amount of functionality in a single class. For example, a Car class can hold a definition and all of its functions. What if the functionality is too much? Obviously, the file becomes large and it becomes cumbersome to use and update by developers. What if multiple developers in a team want to modify the same class? What if the Car class has functionality separated by its external interface, internal design and the engine and three different developers are working on three different aspects of this class?

"Partial classes improves code maintainability"

The solution to this problem is partial classes. We can break down the Car class into three partial classes. Each partial class can be a different source file (physically). In the case of the Car class, there can be three physical files with even different names, CarExternal.cs, CarInternal.cs and CarEngine.cs, all of which are defined in a Car class. When all these three partial classes are compiled, they are compiled as one Car class. But the advantage here is, each of three developers can work on each of three files without affecting one another.

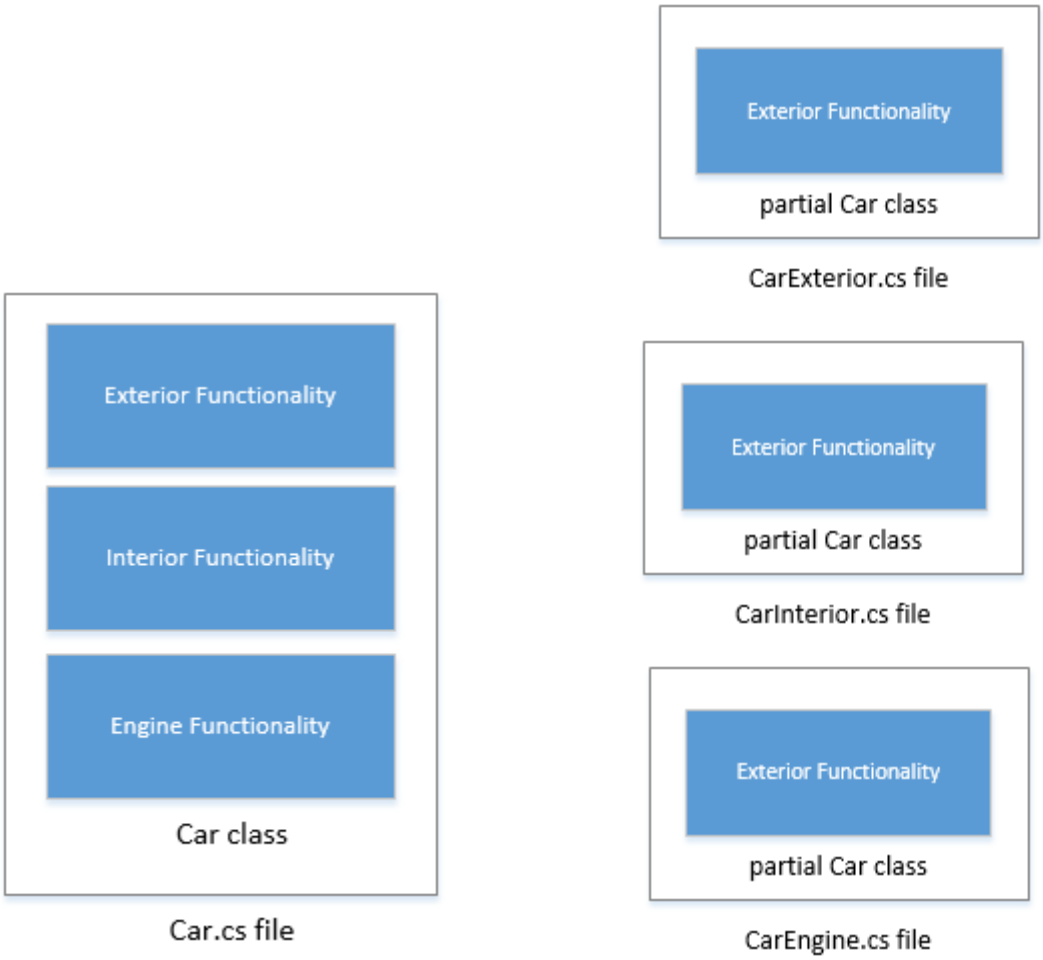


Figure 1

Figure 1 shows two images. The first image shows an example of a typical single class Car that resides in a single Car.cs file. The second image is an example of three partial Car classes that reside in three different .cs files.

In some project types, Visual Studio separates a class into two partial classes to separate the user interface from code behind so the designers and programmers can work on the same class simultaneously.

Not only the definition of a class, but a struct, interface and a method can be implemented as partial and split over multiple source files.

Creating Partial Classes

Let's take a quick look at the Car class in Listing 1.

```
public class Car
{
    // Car Exterior Functionality
    public void BuildTrim() { }
    public void BuildWheels() { }

    // Car Interior Functionality
    public void BuildSeats(){}
    public void BuildDashboard() { }

    // Car Engine
    public void BuildEngine() { }
}
```

Listing 1

The code in Listing 1 defines the Car class that has its functionality broken down into three parts – Exterior, Interior, and Engine.

The partial keyword is used to create a partial class. All the parts must use the partial keyword. All the parts must be available at compile time to form the final type. All the parts must have the same accessibility, such as public, private, and so on.

Now, we are going to break down the Car class into three partial classes to make it simpler. See Listing 2.

```
public partial class Car
{
    // Car Exterior Functionality
    public void BuildTrim() { }
    public void BuildWheels() { }
}


public partial class Car
{
    // Car Interior Functionality
    public void BuildSeats() { }
    public void BuildDashboard() { }

}

public partial class Car
{
    /// Car Engine
    public void BuildEngine() { }
}
```

Listing 2

All partial classes in Listing 2 reside in multiple .cs files.

Partial Methods

A partial class or struct may contain a partial method. Similar to a partial class, a partial method can be used as a definition in one part while another part can be the implementation. If there is no implementation of a partial method, the method and its calls are removed at compile time.

A partial method declaration consists of two parts: the definition and the implementation.

The code listing in Listing 3 defines a partial method called InitializeCar in a different .cs file and the implementation of the same method is in a different .cs file.

```
public partial class Car
{
    // A partial method definiton
    partial void InitializeCar();

    // Car Exterior Functionality
    public void BuildTrim() { }
    public void BuildWheels() { }
}

public partial class Car
{
    /// Car Engine
    public void BuildEngine() { }

    // A partial method implementaion
    partial void InitializeCar()
    {
        string str = "Car";
        // Put all car initialization here
    }

}
```

Listing 3

Automatic Implemented Properties in C#

Auto-implemented properties were introduced in C# 3.0 to make developer's lives simpler by avoiding default implementations of properties.

"Automatic Implemented properties makes developers lives simpler"

Let's say we create a public property called Model. The default implementation of the property looks as in Listing 4.

```
public partial class Car
{
    private string model;

    public string Model
    {
        get { return model; }
        set { model = value; }
    }
}
```

Listing 4

The same code can be replaced with Listing 5.

```
public string Model { get; set; }
```

Listing 5

Listing 5 is an automatically implemented property example. You must implement both a getter and a setter on a property. The auto-implemented properties code basically tells the compiler that this property is the default property implementation and the compiler writes the code necessary to do exactly the same as what Listing 4 does.

Object Initializer in C#

Object initializers are the easiest and fastest way to assign values of an object's properties and fields. An object can be initialized without explicitly calling a class's constructor.

"Object initializers reduces coding time"

The code snippet in Listing 6 lists an Author class with some public properties.

```
public class Author
{
    public string Name { get; set; }
    public string Book { get; set; }
    public string publisher { get; set;}
    public Int16 Year { get; set; }
    public double Price { get; set; }
}
```

Listing 6

In a typical behavior, we will create the Author class object using the constructor and set its properties values listed in Listing 7.

```
Author mahesh = new Author();
mahesh.Name = "Mahesh Chand";
mahesh.Book = "LINQ Programming";
mahesh.publisher = "APress";
mahesh.Year = 2013;
mahesh.Price = 49.95;
```

Listing 7

By using the object initializers, we can pass the public property's values when we are creating the object without explicitly invoking the Author class constructor. The code snippet in Listing 8 does the trick.

```
Author mahesh = new Author()
{
    Name = "Mahesh Chand",
    Book = "LINQ Programming",
    publisher = "APress",
    Year = 2013,
    Price = 49.95
};
```

Listing 8

This may not sound as useful right now but when you're dealing with complex queries in LINQ, the object initializers make developer's lives much easier. Not only object initializers make it easier to initialize objects but also make code look cleaner and more readable.

Collection Initializer in C#

Collection initializers provide the fastest and easiest way to initialize a collection that implements IEnumerable. In a collection, the Add method is used to add items to a collection.

"Collection initializers reduces coding time"

The code snippet in Listing 9 creates a collection of List<string> and uses the Add method to add items to the collection.

```
List<string> authors = new List<string>();
authors.Add("Mahesh Chand");
authors.Add("Raj Kumar");
```

```
authors.Add("Dinesh Beniwal");
```

Listing 9

We can simply replace the code above with the following code that uses a collection initializer approach and adds items to the collection during the creation of the collection object. See Listing 10.

```
List<string> authors = new List<string>{
    "Mahesh Chand", "Raj Kumar", "Dinesh Beniwal"
};
```

Listing 10

This may not sound as attractive to you right now but when you're dealing with complex queries in LINQ, the object and collection initializers make developer's lives much easier. Not only collection initializers make it easier to initialize collections but also make code look cleaner and readable.

The code snippet in Listing 11 lists an Author class with some public properties.

```
public class Author
{
    public string Name { get; set; }
    public string Book { get; set; }
    public string publisher { get; set;}
    public Int16 Year { get; set; }
    public double Price { get; set; }
}
```

Listing 11

The code snippet in Listing 12 creates a collection of Author objects using collection initializers.

```
List<Author> authors = new List<Author>{
    new Author(){ Name="Mahesh", Book="ADO.NET Programming",
        publisher="Wrox", Year=2007, Price=44.95 },
    new Author(){ Name="Raj", Book="LINQ Cookbook",
        publisher="APress", Year=2010, Price=49.95 },
    new Author(){ Name="Praveen", Book="XML Code",
        publisher="Microsoft Press", Year=2012, Price=34.95 }
};
```

Listing 12

Type Inference in C#

C# is a strongly typed language and the default type declaration is an explicit type. For example, the following code snippet declares and initializes two variables. The first variable, *name* is a string type variable and the second variable, *age* is an integer type variable.

```
string name = "Mahesh Chand";
int age = 25;
```

If I add this line of code below the code above:

```
name = 32;
```

Then the compiler throws the following error:

"Cannot implicitly convert type 'int' to 'string.

C# 3.0 introduced a *var* keyword that can be used to declare implicit types. The purpose of implicit types is to store any type in a variable.

The following code snippet declares an implicit variable that stores a string.

```
var name = "Mahesh Chand";
```

Once an implicit type is defined, the compiler knows the type of the variable. You cannot assign another type of value to it. For example, if we write this code:

```
name = 25;
```

The compiler throws the following error:

"Cannot implicitly convert type 'int' to 'string.

In C#, the var keyword tells the compiler to use the type inference to determine the type of a variable. Type inference is heavily used in LINQ queries so any type can be stored in the variable.

The code snippet in Listing 13 shows the Author class.

```
public class Author
{
    public string Name { get; set; }
    public string Book { get; set; }
    public string Publisher { get; set;}
    public Int16 Year { get; set; }
    public double Price { get; set; }
}
```

Listing 13

The code snippet in Listing 14 creates a List of authors and uses LINQ to query the authors list.

```
List<Author> authorsList = new List<Author>{
            new Author(){ Name="Mahesh", Book="ADO.NET Programming",
                Publisher="Wrox", Year=2007, Price=44.95 },
            new Author(){ Name="Raj", Book="LINQ Cookbook",
                Publisher="APress", Year=2010, Price=49.95 },
            new Author(){ Name="Praveen", Book="XML Code",
                Publisher="Microsoft Press", Year=2012, Price=44.95 }
        };

var authorQuery = from author in authorsList
                    where author.Price == 44.95
                    select new { author.Name, author.Publisher };

// In case of an anonymous type, a var must be used
foreach (var author in authorQuery)
{
    Console.WriteLine("Name={0}, Publisher={1}", author.Name, author.Publisher);
}
```

Listing 14

Anonymous Types in C#

Anonymous types are introduced in C# 3.0 with the special purpose of providing an easy way to define objects with a few read-only properties. In a traditional way, we will have to create a class and define read-only properties of the class and that's it. Nothing more. So the C# development team added anonymous types to avoid the necessity of defining the classes. The C# compiler understands when an anonymous type is defined. Each property of the class in inferred by the compiler.

"Anonymous types provides an easy way to define objects"

Anonymous types contain one or more public read-only properties. Members of the class that are not read-only are not valid.

Let's look at the code snippet in Listing 15.

```
var annAuthor = new
{
    Name = "Mahesh Chand",
    Book = "ADO.NET Programming",
```

```
    Publisher = "APress",
    Year = 2003,
    Price = 49.95
};
```

Listing 15

As you can see in the above code snippet, there is no type defined for Name, Book, Publisher, Year, and Price. However strings, integer, and double values are assigned to these variables. This type of declaration in C# is an anonymous type declaration.

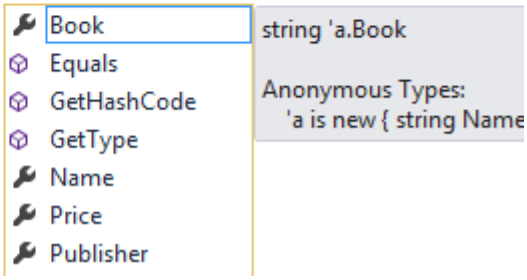
In this code, the left side var is a type inference of a class that has read-only properties with their values assigned in the definition.

So how does this work?

When you declare anonymous types, the compiler automatically creates a class at run-time and knows the internals of the class members and their implicit types.

For example, if you make a call to the "annAuthor" var type defined in your code above, you will see the designer generates the definition and auto-populates the class members as you can see in Listing 16.

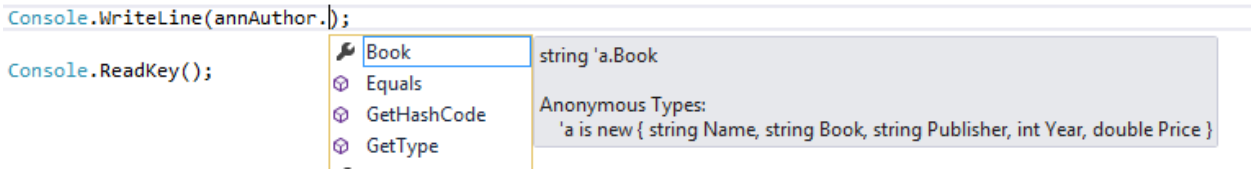
```
static void Main(string[] args)
{
    var annAuthor = new
    {
        Name = "Mahesh Chand",
        Book = "ADO.NET Programming",
        Publisher = "APress",
        Year = 2003,
        Price = 49.95
    };

    Console.WriteLine(annAuthor.);

    Console.ReadKey();
}
```



Listing 16

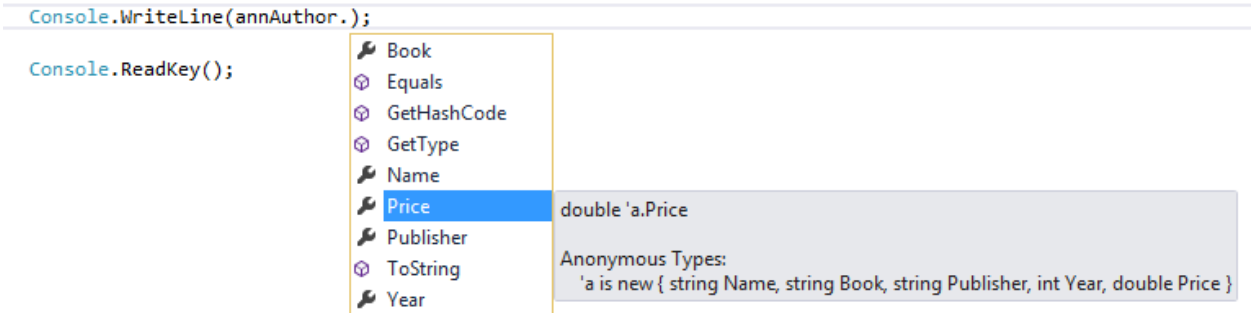
As soon as you call annAuthor, the read-only properties are available and the compiler also generates and explains the anonymous types and their data type.

The image in Listing 17 shows that the compiler knows that "a" is a new with the members Name, Book, Publisher, Year and Price. The compiler also shows that a.Book is a string type.



Listing 17

If you select Price, you will see it is a double type. See Listing 18.



Listing 18

Anonymous types are heavily used in LINQ. It saves developers time by not requiring the writing of anonymous types if they are being used for this simple purpose only.

Delegates

C# has a data type that allows developers to pass methods as parameters of other methods. This data type is called a delegate. A delegate is a reference type.

Delegates are used to encapsulate and pass methods as parameters to other methods. A delegate can encapsulate a named or an anonymous method. You're most likely to use a delegate in events or callbacks.

"Delegates are used to encapsulate and pass methods as parameters to other methods"

The delegate keyword is used to define a delegate type. A delegate type defines a method signature and when you create an instance of a delegate, you can associate it with any method that has the same signature. The method is invoked though the delegate instance.

Let's say we have a method called AddNumbers that adds two numbers. The code snippet in Listing 19 defines the AddNumbers method that returns a sum of the two integer parameters.

```
public static int AddNumbers(int a, int b)
{
    return a + b;
}
```

Listing 19

We want to use this method in a delegate.

First, we need to be sure to define a delegate with the same signature. The following code snippet declares a delegate that has the same signature as the AddNumbers method.

```
public delegate int DelegateInt(int a, int b);
```

Listing 20

Now, we need to define a delegate caller method that uses the above declared delegate as a parameter. The code snippet in Listing 21 defines a method that takes three parameters. The first two parameters are integer values and the third parameter is a delegate of type DelegateInt.

```
public static void DelegateCallerMethod(int a, int b, DelegateInt DelInt)
{
    Console.WriteLine(DelInt(a, b));
}
```

Listing 21

Until now, our delegate has no idea of what it will be doing. All it knows is that it wraps a method that takes two integer values.

Now, to do the actual work, we must create an instance of the delegate by passing a method with similar signature.

The following code snippet creates a DelegateInt instance and passes the AddNumbers method. Now the delegate knows what method to execute when it is called.

```
DelegateInt dm = AddNumbers;
```

The following code is a call to the DelegateCallerMethod that takes a delegate as the third parameter which actually is the AddNumbers method.

```
DelegateCallerMethod(3, 6, dm);
```

You may not find a delegate useful everywhere but it is very important in events and callbacks when you want to force some events to be executed. Delegates also play a major role in LINQ.

The following code lists the complete sample.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DelegatesSample
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create a delegate and assign a method
            DelegateInt dm = AddNumbers;
            DelegateCallerMethod(3, 6, dm);

            Console.ReadKey();

        }

        // Define a delegate. It has same signature as AddNumbers method
        public delegate int DelegateInt(int a, int b);

        // Method that will be used to assign to a delefate
        public static int AddNumbers(int a, int b)
        {
            return a + b;
        }

        // Call delegate by passing delegate
        public static void DelegateCallerMethod(int a, int b, DelegateInt DelInt)
        {
            Console.WriteLine(DelInt(a, b));
        }
    }
}
```

Listing 22

Further readings: Delegates and Events in C# and .NET

Generic Delegates

In the above discussion, we have a delegate called DelegateInt that takes two integer parameters and returns an int type.

```
public delegate int DelegateInt(int a, int b);
```

The DelegateInt works with only methods that have two integer parameters. What if we want to create a delegate that will work with any type of two parameters and return any type? In the above case, it will not work. This is where generics are useful and generics play a major role in LINQ.

The following code snippet declares a generic delegate.

```
public delegate string GenericDelegateNumber<T1, T2>(T1 a, T2 b);
```

The code snippet in Listing 23 defines two methods that we will use when creating instances of generic delegates.

```
public static string AddDoubles(double a, double b)
{
    return (a + b).ToString();
}

public static string AddInt(int a, int b)
{
    return (a + b).ToString();
}
```

Listing 23

The code snippet in Listing 24 creates two delegate instances where first one uses integers and the second delegate uses double parameter values.

```
GenericDelegateNumber<int, int> gdInt = new GenericDelegateNumber<int, int>(AddInt);
Console.WriteLine(gdInt(3, 6));


GenericDelegateNumber<double, double> gdDouble = new GenericDelegateNumber<double,
double>(AddDoubles);
Console.WriteLine(gdDouble(3.2, 6.9));
```

Listing 24

Listing 25 is the complete code sample.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace GenericDelegateSample
{

    class Program
    {
        static void Main(string[] args)
```

```
        {
            GenericDelegateNumber<int, int> gdInt = new GenericDelegateNumber<int,
int>(AddInt);
            Console.WriteLine(gdInt(3, 6));


            GenericDelegateNumber<double, double> gdDouble = new
GenericDelegateNumber<double, double>(AddDoubles);
            Console.WriteLine(gdDouble(3.2, 6.9));

            Console.ReadKey();
        }

        // Generic Delegate takes generic types and returns a string
        public delegate string GenericDelegateNumber<T1, T2>(T1 a, T2 b);

        public static string AddDoubles(double a, double b)
        {
            return (a + b).ToString();
        }

        public static string AddInt(int a, int b)
        {
            return (a + b).ToString();
        }
    }
}
```

Listing 25

Extension Methods in C# Simplified

Developers are always looking for creative ways to extend existing libraries and classes that were developed by other developers. Let's say, in one of my projects, I use a class library that was developed by some developer that I have no idea about and obviously, I do not have access to the original code. Now, I want to extend the functionality of the class library (class or struct) by adding some methods. The straight-forward OOPs method is to inherit a class from an existing class and extend its methods. What if we just want to extend an existing type by just a method? The inheritance can be used but it has some limitations. The first problem is structs don't fall in this category. Secondly, inheritance is not an effective way to extend an existing class by just a method or two. The C# compiler does much more internally when a class is being inherited. What about if a class is a sealed class? Many developers won't let you inherit a type from their types and they define their classes as sealed classes.

An alternative approach is to define extension methods. Using extension methods, we can add new methods to existing types and use them through the type instances.

18 "Extension methods extends the functionality of existing types"

Extension methods featured in C# gives developers power to add new methods to existing types without inheriting the type, recompiling, or modifying it.

Let's take an example.

Let's say, we have a class called AuthorElite defined in the Listing 26 code snippet.

```
public class AuthorElite
{
    public ArrayList AllEliteAuthors()
    {
        return new ArrayList()
            {"Mahesh Chand", "Praveen Kumar", "Raj Kumar"};
    }
}
```

Listing 26

Now, this class can be in the same or a different assembly. As long as you have added a reference to the assembly, you're good to go.

The way to extend the class above is by creating a new static class and defining a new static method within the new class. The new static method takes one "this" parameter of the above class type.

The code snippet in Listing 27 creates a static class, defines a new static method and passes the AuthorElite class as a parameter.

```
public static class ExtendedAuthorElite

{
    public static ArrayList NewAuthors(this AuthorElite elite)
    {
        return new ArrayList() { "Author One", "Author Two", "Author Three" };
    }
}
```

Listing 27

Now, let's see how we can use the extended method. The code snippet in Listing 28 creates an instance of the AuthorElite type and as you can see, you have access to both the methods that were defined in the AuthorElite and the ExtendedAuthorElite classes.

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

```
{
    AuthorElite ae = new AuthorElite();
    ae.AllEliteAuthors();
    ae.NewAuthors();
}
```

Listing 28

The code snippet in Listing 29 is the complete example of the example discussed above.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Collections;

namespace ExtensionMethodsSample
{
    class Program
    {
        static void Main(string[] args)
        {
            AuthorElite ae = new AuthorElite();
            ae.AllEliteAuthors();
            ae.NewAuthors();
        }
    }

    public class AuthorElite
    {
        public ArrayList AllEliteAuthors()
        {
            return new ArrayList()
                {"Mahesh Chand", "Praveen Kumar", "Raj Kumar"};
        }
    }


    // Extended class
    public static class ExtendedAuthorElite
    {
        public static ArrayList NewAuthors(this AuthorElite elite)
        {
            return new ArrayList() { "Author One", "Author Two", "Author Three" };
        }
    }

}
```

Listing 29

Dynamic Programming in C#

20

C# is a strongly-typed language. What does that mean? In the C# language, every variable and constant has a pre-defined type. But what if you are not sure of the type of the variable? This is where dynamic programming is useful that was introduced in C# 4.0.

The dynamic keyword is used to declare dynamic types. The dynamic types tell the compiler that the object is defined as dynamic and to skip type-checking at compile time. Instead leave it to the runtime. All syntaxes are checked and errors are thrown at run-time. The dynamic programming was introduced to provide a cross reference between .NET and non .NET APIs such as COM APIs, IronPython, and HTML Document Object Model (DOM).

The code snippet in Listing 29 declares a dynamic field, a dynamic property, a dynamic method and a dynamic variable.

```
// Define a dynamic field.
dynamic name;

// Define a dynamic property.
dynamic NameProperty { get; set; }

// Define a dynamic method with a dynamic parameter type.
public dynamic FullNameMethod(dynamic d)
{
    name = "Mahesh";
    dynamic firstname = name;
    string lastname = "Chand";
    return firstname + lastname;
}
```

Listing 29

As you may have noticed from the code above, a dynamic variable can store any type of value. A method can also return a dynamic type and also can accept dynamic parameters.

Here is an example where a dynamic variable can store a string, an integer, a double, and a Boolean value. See Listing 30.

```
dynamic dyno = "Mahesh Chand";
Console.WriteLine(dyno);
dyno = 38;
Console.WriteLine(dyno);
dyno = 44.95;
System.Console.WriteLine(dyno);
dyno = true;
System.Console.WriteLine(dyno);
```

Listing 30

Dynamic types can also be converted explicitly to other types and vice versa. The code snippet in Listing 31 converts a string, an integer, and a double type into dynamic types by casting them with the dynamic keyword.

```
dynamic d;
string name = "Mahesh Chand";
d = (dynamic)name;
Console.WriteLine(d);

int age = 38;
d = (dynamic)age;
Console.WriteLine(d);

double price = 44.95d;
d = (dynamic)price;
Console.WriteLine(d);
```

Listing 31

Dynamic can be used with classes too. The following example defines an Author class with a dynamic method `FullNameMethod`. The Main method creates a dynamic type of the Author class and calls its dynamic method by passing a dynamic parameter. See Listing 32.

```
namespace DynamicProgSample
{
    class Program
    {
        static void Main(string[] args)
        {
            dynamic lastName = "Chand";
            dynamic dynAuthor = new Author();
            Console.WriteLine(dynAuthor.FullNameMethod(lastName));

            Console.ReadKey();
        }
    }

    public class Author
    {
        // Define a dynamic field.
        dynamic name;
        // Define a dynamic property.
        dynamic NameProperty { get; set; }
        // Define a dynamic method with a dynamic parameter type.
        public dynamic FullNameMethod(dynamic d)
        {
            name = "Mahesh";
            dynamic firstname = name;
            return firstname + " " + d;
        }
```

```
    }
}
```

Listing 32

Dynamic Language Runtime
http://msdn.microsoft.com/en-us/library/dd233052%28VS.100%29.aspx

Named and Optional Parameters in C#

Named and optional arguments were introduced in C# 4.0 that allow developers to have an option to create named and optional parameters in a method.

Listing 33 lists the AuthorRank class that defines and implements the CalculateRank method. The CalculateRank method takes three arguments of type short, double, and Boolean respectively.

```
public class AuthorRank
{
    public double CalculateRank(short contributions, double loyalty, bool include)
    {
        if (include)
            return (contributions * 0.12 + loyalty * 0.9);
        return contributions * 0.12;
    }
}
```

Listing 33

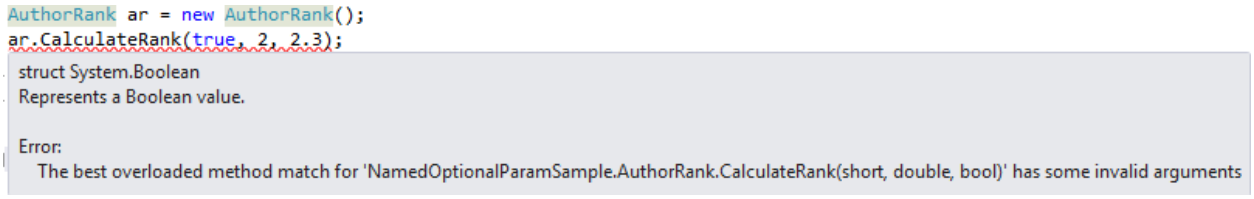
The code snippet in Listing 34 calls the CalculateRank method where the first argument is a short value, the second argument is a double value and the third argument is a Boolean value.

```
AuthorRank ar = new AuthorRank();
ar.CalculateRank(2, 2.3, true);
```

Listing 34

Named Arguments

So what happens if we do not pass the argument values in the same order as they are defined? For example, if we change the order of the values of parameters, we will get an error message from the compiler.



Thank God, we have the Visual Studio IDE Intellisense and the compiler that tell us the names and types of the parameters are incorrect. You may thank reflection feature that reads an API (DLL) and lists us its methods, their parameter names and their types.

What if you don't have Visual Studio and am writing code in some editor that does not have the Intellisense? What if we are calling an API such as a COM API that does not support reflection? What if we know parameter names only, without their order?

This is where you may find the named arguments or named parameters feature useful.

Named arguments allow us to call a method without knowing the orders of the parameters.

The named arguments of a method are defined using the parameter name followed by a colon and the parameter value. Once you define a method argument as named, no matter what order it is in, the compiler knows what it is and where to use the parameter value in what order.

The following code snippet uses named arguments to call the CaclulateRank method that is listed in Listing 35.

```
AuthorRank ar = new AuthorRank();
ar.CalculateRank(include:true, contributions:2, loyalty:2.3);
```

Listing 35

As you can see from the above code, the order of the method parameter is not the same as in the definition.

Optional Arguments

Optional arguments allow developers to write methods that have optional parameters. Each optional parameter has a default value and the caller code does not pass a value of the optional parameter, the default value is used. Optional parameters are defined at the end of the parameter list, after any required parameters. If the caller provides an argument for any one of a

succession of optional parameters, it must provide arguments for all preceding optional parameters.

Let's take a look at code in Listing 36.

```
public class AuthorRank
{
    public double OptionalParameterMethod(short contributions, double loyalty = 3.1, bool
include = true)
    {
        if (include)
            return (contributions * 0.12 + loyalty * 0.9);
        return contributions * 0.12;
    }

}
```

Listing 36

In Listing 36, the OptionalParameterMethod the last two parameters are optional and the caller code does not need to pass a value for these parameters.

The code snippet in Listing 37 calls OptionalParameterMethod in two different ways. The first call does not specify any value of the optional parameters. The second call on the other hand passes the optional parameters values.

```
AuthorRank ar = new AuthorRank();
Console.WriteLine(ar.OptionalParameterMethod(3));
Console.WriteLine(ar.OptionalParameterMethod(3, 4.4, true));
```

Listing 37

Covariance and Contravariance in C#

Covariance and contravariance features were introduced in C# 4.0. Covariance and contravariance enable implicit reference conversion for array types, delegate types, and generic type arguments. Covariance preserves assignment compatibility and contravariance reverses it.

For example, in case of generics, Covariance allows casting of generic types to base types, for example, IEnumerable<A> will be implicitly convertible an IEnumerable<B> if A can implicitly be converted to B and Contravariance reverses it. See Listing 38.

```
// List of string
IList<string> arrNames = new List<string>();
// Convert it to IEnumerable collection of objects
IEnumerable<object> objects = arrNames;
```

Listing 38

Lambdas

25

Delegates are an integral part of LINQ. However, the process of declaring and implementing delegates is a few steps. In C# space, the Lambda expressions are a concise way to define methods.

Let's create a method as in the following called Mul, that takes two integer type parameters, multiplies the two input parameters and returns the multiplied value. See Listing 39.

```
int Mul(int a, int b)
{
    return a * b;
}
```

Listing 39

Now to use the Mul method, we can simply call it in the following way:

```
int result = Mul(5, 3);
```

Now, what happens when you need to use that often? We can do that by simply using lambda expressions. The code snippet in Listing 40 is a lambda expression that replaces the preceding code.

```
delegate int del(int i, int j);
del mul = (x, y) => x * y;
int result2 = mul(5, 3);
```

Listing 40

The preceding comparison does not do justice to the lambdas but you get the idea. LINQ relies heavily on and uses delegates and that is where you will find lambdas saving you time.

Lamda Expression

The operator "=>" is called the lambda operator. The lambda operator => is read "goes to". A lambda expression has a left side and a right side. The following is a definition of a lambda expression.

A => B

Is read as A goes to B.

In the above expression, the left side (A) is the input parameters and the right side (B) is the code execution block.

(Input Parameters) => Code execution block;

Let's take the example of the preceding code:

```
(x, y) => x * y;
```

It is read as "x and y goes to x into y".

In the preceding lambda expression, the left side (x, y) are the input parameters and the right side (x * y) is the code execution block. The preceding lambda takes two parameters and multiplies them, and returns the multiplied value.

Summary

This book is an introduction to the key features of C# versions 3.0, 4.0, and 5.0. The detailed examples and tutorials may be found on C# Corner (www.c-sharpcorner.com).