.NET Languages and its Construct

In this appendix, we will cover:

- Evolution of the .NET language
- Working with language features
- Advantages of Generics in .NET
- Using iterator blocks in .NET
- Working with LINQ and Lambda expressions
- Using Dynamic objects in .NET
- Compiler as a service

Introduction

As we already know most of the basics of .NET core, and the architecture of the .NET environment, another important part of the framework, that I have rather ignored in the book is Languages. .NET is a repository of more than one language. During the initial release of .NET Framework, Microsoft released a number of languages, each of which were quite different in their syntaxes and targeting some set of programming. Languages are tools for an application developer. Supporting so many languages made it more versatile and widely acceptable. Notably, the most important languages released during its inception is C# and VB.NET, where C# is targeting people who are confident on C++ programming style and VB.NET is targeting towards Microsoft's existing customers who works on VB. But the important thing is both of these languages or any other languages targeting .NET Framework share the same base class library and can easily interoperate between each other.

The languages are actually provided with compilers (CSC for C# and VBC for VB.NET) that produce the intermediate language. As each of these languages are the same when converted to **Common Intermediate Language** (**CIL**), the two assemblies from different languages can easily communicate and be called from each other.

The benefits of .NET Framework languages are as follows:

- A large number of languages targeting the same framework
- Interoperability of application code built under different languages
- Language development can spread outside the Microsoft domain, as the Common Language Specification is standardized under ECMA
- The learning curve is adaptable, as you can choose the language that you are comfortable with
- All languages gets updated when a new release of Framework is updated
- Easier installation of applications as assemblies are not machine dependent code, and do not need redeployment for every platform, rather it works as XCopy
- No need to install runtime when you need to run .NET applications, as the runtime is automatically shipped with Windows

The benefits of .NET Framework are enormous. .NET framework is built into a large number of assemblies, each of which is logically separated into namespaces.

A namespace is a logical separation of one type with another. A namespace is created for each type, and when the type is read the valid path of the entire namespace is required. For example, if a type `MyType` is defined inside a namespace `MyNamespace`, the type can only be accessed using `MyNamespace.MyType`. Inside a namespace, you can define: `namespace`, `class`, `interface`, `delegate`, `struct`, and `enum`.

The namespaces separates one type from the other, for instance `System.Collection` namespace holds all the types that deal with a collection of objects. On the other hand, any type that deals with input/output should not be written inside `System.Collection`, rather should stay inside `System.IO`.

.NET also allows you to specify your own namespace. When a namespace is defined locally, you can easily access the namespace defined in the Framework using the `global::` keyword.

Inside these namespaces the .NET Framework defines a huge number of classes. These classes generally deal with everything that you can think of, and they are also getting more and more enhanced as time progresses. Hence, most of the things that you might require to implement may be already implemented inside the .NET Framework. Using .NET languages you can invoke these types and properties.

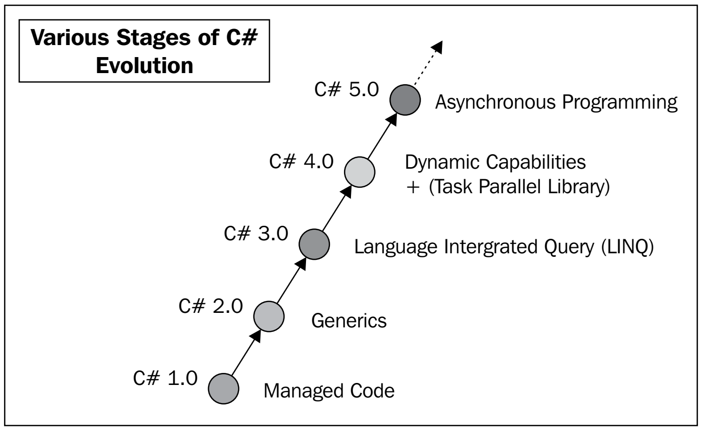
Evolution of the .NET language

As a developer, languages are used to produce applications and use .NET Framework core APIs. As you already know, .NET language is compiled to CIL, hence the use of the compiler is only to convert the high-level language to CIL. There are a number of languages that were introduced from the very birth of the Framework. Each of the languages that are built on top of .NET Framework follow some basic principles. These principles are defined as **Common Language Specification** (**CLS**). CLS defines the standard of the IL which the languages should follow when the compiler is built. The CLS is designed to include the entire basic construct that each language needs to implement to make the language CLS-compliant. The CLS compliance means that the code that is written follows the standard rules and restrictions applied by CLS.

Most of the languages follow a large part of the CLS document, but there are exceptions too. For instance, C++ allows multiple inheritance which is not actually there as a set of standard rules of CLS. Another instance can be C# which allows you to write unsafe code. The type defined in unsafe code is not CLS-compliant. Hence some languages do follow CLSs, but as each language has its own domain of activity and is built for some specific set of applications, this language implementation differs in terms of few exceptions to CLS.

But recently, Microsoft has focused on implementing the missing feature of one language into other, so that these languages can be made almost identical rather than syntactically different. During the phase of development of languages, a large part of language implementation goes in languages C# or VB.NET; these languages are getting richer and richer day by day. In this recipe, we will see how the languages in the .NET Framework have been enhanced with new releases of .NET Framework, and see some basic introduction of the features that comprises the version.

Let us demonstrate the evolution of languages using an image depicting the different stages of the C# language.


Various Stages of C# Evolution
C# 5.0
Asynchronous Programming
C# 4.0
Dynamic Capabilities + (Task Parallel Library)
C# 3.0
Language Intergrated Query (LINQ)
C# 2.0
Generics
C# 1.0
Managed Code

The preceding figure depicts the various stages of evolution of the C# language. C# as a language has changed a lot from its inception. It was introduced with the introduction of .NET Framework, and at that time it was basically a new language that supported managed code. Now let us depict each evolution step one by one.

1. The first release, C# 1.0, was all about building a new language that supported the managed code environment which was introduced with .NET Framework.
2. C# 2.0 introduced Generics. It was a new feature that deferred a type to be passed to another type at runtime in such a way that each type was determined at runtime when the object was created, and yet the type safety was maintained.
3. C# 3.0 introduced a few huge advances, such as WPF, WCF, WWF, and Windows Cardspace.
4. C# 3.5 introduced several functional programming constructs to the language including **Language Integrated Query** (**LINQ**) syntaxes. LINQ is a new language feature that allows you to query object code at runtime.
5. The fourth release of C# 4.0 introduced the Dynamic language feature. The **Dynamic Language Runtime** (**DLR**) capability of the language defered type checking to runtime allowing you to call members to a runtime object, and type checking during compile time was not performed.
6. The fifth release of C# 5.0 introduced the new async feature which allows you to write an asynchronous code block instantly with two contextual keywords in the language called async and await.

As we have introduced all the evolution points of C# language, it would be a good idea to explain each of these evolution steps and get a deep idea about them.

C# 1.0 introduced managed code. Managed code is CLR managed. The managed code execution specifies a contract between the code that is running inside CLR and the code that runs natively. The contract specifies that at any point of execution, the runtime can stop its execution and query information about the CPU instructions, and fetch the state of the objects in memory. **Just in time** (**JIT**) compiles managed code (**IL**) into the native environment during execution, and can guarantee certain things like garbage collection, exception handling, type safety, array bounds, or even index checking. JIT is also free to modify the order of IL execution to optimize its performance.

```
static void Main(string[] args)
{
    Int32 i = 10;
    int j = 10;

    Console.WriteLine(i == j);
```

CLR uses the same framework for every language. In the preceding code, both the values are equal and `int` is an alias of `Int32`. During compilation the language aliases are converted to actual type supported by the Framework.

C# 2.0 introduced Generics. It refers to a technique to defer the declaration of a type which is defined as a template during compilation, and the type is referred when an object is instantiated. Generics ensured reusability with type safety.

```
public class Generic<T>
{
    public T myGenericTypeField;
}

//Call
Generic<int> gint = new Generic<int>();
Generic<string> gstring = new Generic<string>();
```

In the preceding code, the `Generic` type allows you to pass a type as a template. It creates a member of the type `T` inside it. Hence, during its object creation the `gInt` will define the `myGenericTypeField` as integer, and `gstring` as string.

C# 2.0 also introduced anonymous delegates, iterators, and partial classes. We will discuss them in detail later in this appendix.

C# 3.0 introduced LINQ that allows querying runtime application data. The language comes with constructs leveraging existing knowledge of database in application environment.

```
var contacts =
    from c in customers
    where c.State == "WB"
    select c;
```

Here say customers are a collection from which we select each object, and check whether it belongs to `"WB"`. The final contacts will hold all the customers that belong to `"WB"`. `var` specifies the implicitly typed variables. That means it will be replaced by the appropriate type during compilation. Here the type of contacts will be `IEnumerable<Customer>` if customers are a collection of customer type.

C# 3.0 also introduced anonymous types, extension methods, generic delegates, auto implemented properties, and so on. It is discussed in detail in the book.

C# 4.0 introduces dynamic language runtime. Programming languages are generally either statically typed or dynamically typed languages. C# is generally considered as a statically typed language while JavaScript, python, and so on are dynamically typed languages. Dynamic languages do not perform compile time type checks and identify the type of objects at runtime. C# 4.0 allowed dynamic elements to improve interoperability with dynamic languages and frameworks. The C# team introduced a new keyword `dynamic` that deals with dynamically typed objects.

```
static void Main(string[] args)
{
    dynamic mydynamicobj = new string();
    mydynamicobj.MyProperty = 10;


}
```

The preceding code will compile but as the `mydynamicobj` reference holds an instance of string, it cannot call `MyProperty` at runtime. So runtime will produce errors even though compile type checks are totally neglected.

With dynamic features, C# also introduced a number of interesting concepts like generic variance, optional parameters, and so on. We will discuss them later in the appendix.

C# 5.0 introduced a new approach of programming style called asynchronous coding pattern. Asynchronous programming is not new to the language. There were a number of possible ways to define asynchronous code blocks even before the introduction of C# 5.0. But the enhancement actually introduced two new contextual keywords, namely async and await to work with `async` code blocks. The keywords help you to write `async` code blocks in almost similar fashion like the synchronous blocks.

```
public async void MyAsync()
{
    await Task.Delay(1000);
}
```

The preceding code block actually puts a delay on the task. The code runs asynchronously without waiting for the execution of `Task.Delay` to finish. We have discussed more about async in the *Working with async and await patterns* recipe in *Chapter 3*, *Asynchronous Programming in .NET*.

Now let's talk about some other options, or possibly some pieces of general information that are relevant to this task.

Why is Generics important?

Generics is one of the best additions to the .NET language ever released. Generics helps you to pass a type to an existing collection when the object is created. In .NET Framework, the first thing you notice on a valid implementation of Generics is the new set of collection classes. The use of generic collection classes are always recommended. Let us take an example; a collection generally holds a number of elements. In the case of .NET Framework before the inclusion of generics, there are a number of collection objects that store the collection elements as objects rather than a collection that has an associated type attached. Now let us suppose the collection is of type `Int`. Hence to hold a collection of integers, each integer needs to be converted to object type. This conversion from `ValueType` to reference type is sometimes referred to as boxing. Boxing includes additional memory allocation, as the boxed element is transferred to a managed heap rather than storing on the stack. On the other hand, whenever we want to get a value from the collection, we again need to unbox each individual element from the collection. Generics allows you to specify the type for which the collection is meant. So in the case of Generic collection, during the creation of its object we can specify that the collection will hold a number of integers, and it ensures that each elements of the collection are actually integers.

Hence, Generics removes the additional overhead of boxing and unboxing on values and hence increases performance.

.NET recommends to use `List<T>` instead of `ArrayList`, and `Dictionary<T1, T2>` instead of `HashTables`, because the former supports generic implementation while the later doesn't. There's lots of generic implementation going on inside the .NET library, and the old non generic collection classes, even though they still exist, are widely depreciated. We will look more in detail about Generics in coming recipes.

What are extension methods?

Extension method forms special meaning to the compiler. Extension methods allow you to extend an already defined type without recompiling, or modifying the type, or even inheriting the type to another type. It is actually defined as static, but acts as an instance specific method. In other words, even though the code that is running inside the extension method is not declared within the scope of the class, it is still associated with an instance of an object for that particular class. Extension method does not need to be in the same namespace, whereas the actual type does. Hence you can easily add an extension method anywhere outside the namespace and any other static class. The namespace, which when added to the code will automatically add those extension methods, and call appropriately for an existing type. Extension methods can also be thought of as an extension to a pre-existing type.

For instance, let me define a class first:

```
public class MyClass
{
    public int MyMethod()
    {
        Random rand = new Random(100);
        return rand.Next();
    }
}
```

This is a very simple class with one method `MyMethod` that returns an integer value. Now, how to extend the class to add another method, say `MyMethod2`? There are a couple of options available for you, either you need to extend this class to another type `MyClass2` derived from `MyClass` and include that method into it, or use an extension method to add it directly within the type `MyClass` itself.

```
public static class Extensions
{
    public static int MyMethod2(this MyClass obj)
    {
        return 10;
    }
}
```

`MyMethod2` is now an extension method that is included with `MyClass`. You should remember that an Extension method is static with the first argument being of Type in which the method is to be included with a `this` keyword before it. This identifies it as an extension method. Any parameter can be passed normally after the first argument.

By the way, you can even make `MyClass` sealed (not extensible), and yet you have the option to use an extension method to extend the type.

What are partial classes and partial methods?

Partial classes allow you to divide your own class into one or more files, but during runtime the files would be merged to produce a single coherent object. In this way the classes could be made very small and easily maintainable even when the type as a whole remains quite big in size.

The most important part of a partial class is the designer. While you work in Visual Studio, you must have found a portion of code that is generated automatically by the designer. Designers create a class which keeps on refreshing itself when certain modifications are made in it. Partial classes allow you to create an alternate implementation of the same class, so that the code you add to the class does not go out when the designer is refreshed.

Partial methods are another new addition to .NET languages that allows you to declare/define the same method more than once. Microsoft introduced the concept of the `partial` method to deal with designer-generated code, and to allow reserving the name of a method that one needed to be implemented later in original classes. In this way, the designer will not produce any warning without implementing the actual method, but will allow future developers to implement or write their own logic on the same method that is defined.

```
partial class MyPartialClass
{
    public void MyNormalMethod()
    {
        Console.WriteLine("Inside first Normal Method");
    }
    partial void MyPartialMethod(); //It is declared and will be
defined later in another partial class
}
```

Let us suppose `MyPartialClass` is generated by the designer. It declares the method `MyPartialMethod`, which I have not defined yet.

```
partial class MyPartialClass
{
    //public void MyNormalMethod()
    //{
    //    Console.WriteLine("Inside second Normal Method");
    //}
    partial void MyPartialMethod()
    {
        Console.WriteLine("Inside second Partial Method");
    }
}
```

In another implementation of `MyPartialClass`, I have defined the method `MyPartialMethod`, that holds the actual implementation of the code. So if I create an object of `MyPartialClass`, it will hold both methods `MyNormalMethod` and `MyPartialMethod`, and whenever it is called it calls appropriately.

Another thing to note is that the implementation of the `partial` method is not mandatory. If you do not define the `partial` method in another `partial` class, it will eventually be eliminated completely in runtime IL.

The limitations of a `partial` method are as follows:

- Partial method is by default private. Hence you can only call a `partial` method from within the partial class.

  ```
  partial class MyPartialClass
  {
      public void MyNormalMethod()
      {
          Console.WriteLine("Inside first Normal Method");
          this.MyPartialMethod();
      }
      partial void MyPartialMethod(); //It is declared and will be
  defined later in another partial class
  }
  ```

  Thus even if you do not declare the method `MyPartialMethod`, you can even call it from within the class. During runtime, CLR will automatically call if it is implemented, or will eliminate the call completely from the system.

- Partial method needs to be declared inside a `partial` class. A `partial` class is only capable of redefining the part later in another file. So, it is essential to mark the class as partial as well when it has a partial method.
- It cannot be marked as extern.
- It must have a void return type, and also cannot use out parameter as argument.

Partial method does not allow us to write as virtual, sealed, new, override, or extern. Even though it is very rarely used in real life, it is still worth knowing.

What are Caller attributes and how they are used?

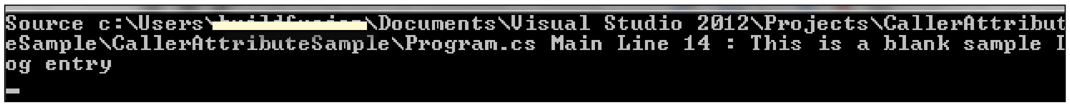
Caller attributes is a new feature introduced with .NET 4.5 which allows you to obtain various information about the program that is currently running. We generally track information about line numbers, source file, and so on either from the stack trace created from exceptions or manually using call stack information available to Visual Studio while debugging the application. But from the actual code or during runtime of the application, this information is somewhat hidden for a long time and even though it exists while running every line of code, it is not available for use in general.

With the introduction of caller attributes, the framework gives an API that allows you to get information, such as `CallerMemberName`, `CallerFilePath`, or `CallerLineNumber` while executing a function. The attributes can be added to the caller function in such a way that when the method is called, the .NET runtime will fill this data with the data that is available to it. Hence, you can use those parameters in our code.

Say for instance you are creating a logger functionality in your code. For doing a log, it is always important to trace this information and write to a file in such a way that whenever tracing the file, you can easily determine the exact line which executed for a particular log entry. For instance, to get the line number, you generally need to manually enter the line number on every line you log, and when you change certain areas of code, you need to again sync all the log entries. But with caller attributes, you do not need to manually keep track of the lines, rather the program gets these values during runtime.

```
static void Main(string[] args)
{
    LogThis("This is a blank sample Log entry");
    Console.ReadLine();
}

static void LogThis(string actualMessage, [CallerMemberName]
string membername = "", [CallerFilePath] string sourceFile = "",
[CallerLineNumber] int sourceLine = 0)
{
    //Log these information
    Console.WriteLine("Source {0} {1} Line {2} : {3}", sourceFile,
membername, sourceLine, actualMessage);
}
```

Here in the preceding code, the `LogThis` method will log the source file, the member name, and line number. Any `CLR` method automatically gets these values for the caller method when the arguments are annotated with these attributes.



```
Source c:\Users\[redacted]\Documents\Visual Studio 2012\Projects\CallerAttribut
eSample\CallerAttributeSample\Program.cs Main Line 14 : This is a blank sample l
og entry
_
```

In the preceding screenshot, the application prints the message with the path of the source file, the exact line number, and the name of the method. These information were available during runtime, but was not exposed before. But with the introduction of Caller attributes, these information are now readily available to any application code. Basically if you look into the internals of the attributes, these attributes are actually replaced by the calls during compilation. Thus during compilation, the parameters are replaced directly with the values.

Working with language features

Classes are the major components of any programming model. .NET comes with a huge set of classes that are categorized into multiple assemblies. These classes are designed and developed using the .NET languages. In this recipe, I will address some of the special implementation of types that form special meaning to the compiler.

When you think of data types of a language, you always think of variables which store values inside. These data types are either mutable, which when you assign to another instance of object copies itself as a value. For instance, most of the basic data types, such as `int`, `float`, `double`, `long`, and so on, are types that are mutable and called as `ValueTypes` in .NET language. Whereas some are actually copied references of the object that have been created and when the object is modified, it will affect all the references that has its pointer referred to. Virtually any type that we define as a class is actually a Reference Type object.

As .NET is purely object-oriented, every object that you create in .NET is actually derived from the parent `System.Object`. Hence if you do not specify anything in the inheritance hierarchy it will put `Object` at its base. When the object that is created is a `ValueType`, the types will be derived from `System.ValueType`. Thus when an object is actually derived from `System.ValueType`, it will be treated by the compiler specially and it will create a mutable instance of the object. Languages restrict any object to be derived from `System.ValueType`, and hence to define a custom type that derives from this type and get special meaning to the compiler we define `struct`.

In this recipe, we are going to cover all the most important types that form a special meaning to the compiler, and give you a brief introduction to each of them and their usage.

Getting ready

To start with the recipe, let us describe some of the basic types that form a special meaning. This will give you a prerequisite on what those types basically are.

- **Class**: A class is a basic entity of a program. When we create a class, it either inherits from the custom class that we specify, or it inherits from the type we specify.
- **Interface**: Interface is a special type that acts as a contract on an object. An interface defines the structure of a type, and when you use this interface for a type, you need to fully qualify each individual definition of the interface implemented within the type.
- **Struct**: A structure is a special type that is automatically derived from `System.ValueType` and restricts inheritance on the type. A `struct` is generally created as mutable and allows to group a set of data elements. It is recommended to create a class type when the size of `struct` is greater than 16 bytes.
- **Delegate**: A delegate is a special type that is derived from `System.Delegate`, and forms a special meaning to the compiler. A delegate is a type that can hold a method. You can assign a method to a delegate when the type of the method matches the type of the delegate, and you can send the delegate anywhere in the code. The delegate can be called using the `Invoke` method, to call the method which it is assigned to.

- **Event**: An event is a special kind of construct in the language that actually forms a wrapper to a delegate. An event has event subscription and actually uses delegate internally to store the methods/event handlers we pass to the event. When an event gets executed, all the methods that have been assigned to the event will be called sequentially.
- **Enum**: Enumerations are special types in .NET that are derived from `System.Enum`. An enum is an enumerated data type which gives names to the values of a domain of data.

Now let's deal with all of these one by one in this recipe and look at the basis of each of them.

How to do it...

Class is the basic unit of programming. In the .NET environment, everything that we write is within a class. .NET being a truly object oriented language you start building your application from a class. Let us create a `WindowsFormsApplication` project and add a class into the project, we'll call it `MyType`.

1. Create a structure and define a few local members in it. Let us name our structure as `MyStruct`.

   ```
   public struct MyStruct
       {
           int IntegerValue { get; set; }
           string Message { get; set; }
       }
   ```

   You should note that `struct` is not a recommended type. Although you can define it, you may want your type to be immutable and also have a size less than 16 bytes long; otherwise, it is better to use a class instead that performs faster than `struct`.

2. Once you have created the structure, let us create an object of the structure within the body of the class. When we specify any type inside the scope of a class, we call that a member object for that particular type.
3. Now let's define a delegate. A delegate as I have already introduced is a special type that can point to a particular method which follows the delegate signature. For instance, I define a delegate as follows:

   ```
   public delegate string CallMeDelegate(int x, int y);
   ```

   This means this is a type that can point to a method which takes two input elements and returns `string`. Let us define a method inside the class to actually implement this functionality.

   ```
   public string CallMe(int x, int y)
           {
   ```

```
            return string.Format("Value of X passed is {0} and the
value of Y passed is {1}", x, y);
        }
```

So the preceding method as defined actually takes the input and returns them as a message.

4. We would call the delegate by using a variable of that type. Let us define a property of the delegate and try to point the same to the method.

   ```
   public CallMeDelegate MethodStore { get; set; }
   ```

5. To this `MethodStore` object we can assign the method `CallMe`. You should note that as this `MethodStore` is public we can also assign a method from the object outside the class. To assign the method to this `MethodStore` we just call:

   ```
   this.MethodStore = this.CallMe;
   ```

   Note that when we are passing the method body to the object of the delegate, we actually don't need to specify the braces and the parameters. It needs to be specified when we actually call the `MethodStore` object. In addition to this, you should also note that a delegate actually maintains a collection of methods inside it, so if you specify more than one method inside a delegate, all those methods will be called at once whenever we call the delegate.

6. In .NET we generally do not expose a property of a delegate object outside the class. .NET implements a new pattern that enables a type to subscribe to a delegate and unsubscribe from it using the event pattern. Even though the event is just a syntactic sugar to the actual underlying delegate, it is better to know exactly how to define an event.

   ```
   private event CallMeDelegate _mymethodcalled;
           public event CallMeDelegate MyMethodCalled
           {
               add
               {
                   this._mymethodcalled += value;
               }
               remove
               {
                   this._mymethodcalled -= value;
               }
           }
   public string OnMyMethodCalled(int x, int y)
   {
     if(this._mymethodcalled != null)
             return this._mymethodcalled(x,y);
     return string.Empty;
   }
   ```

So in the preceding code we have defined a `private event` variable called `_mymethodcalled` and its corresponding event accessor. The event accessor gives you a method stub that will be called whenever the event is subscribed or unsubscribed. Now, you can also see that I have defined a method called `OnMyMethodCalled`. It is important to note that as we have exposed the event, it is not always true that the event is subscribed by the caller. Say for instance, we generally have the mouseUp or mouseDown events on a button, and if we don't need any body for that event we do not specify that to the type. `OnMyMethodCalled` actually determines whether the event handler is assigned or not. If nothing is assigned, the event object will be evaluated to null.

7. Once you define an event, you can now create an object of the type, and subscribe using `+=` sign and unsubscribe using `-=` sign with the appropriate method that we call `EventHandler`.

```
MyStruct structObj;
MyType typeobj = new MyType(structObj);
typeobj.MyMethodCalled += new MyType.CallMeDelegate(typeobj_
MyMethodCalled);
typeobj.MyMethodCalled -= new MyType.CallMeDelegate(typeobj_
MyMethodCalled);
```

   So here the object has been created and `MyMethodCalled` has been subscribed using a method with same signature as the delegate.

8. To define an enum, we use the enum keyword and create few named states.

```
public enum MyEnum
    {
        High, Medium, Low
    }
```

   The enumerated data type defines the states of the object. You can see, we have mentioned three valid states, namely `High`, `Medium`, and `Low`. So upon creating object of the enum, we can only specify any one of these states. It is also important to note that the enumeration type is by default integer if we do not specify it explicitly.

How it works...

As you have already gone through the steps of most of the important features of the .NET language, let us delve deep into the facts which comprise these language features.

To start with, let's talk about interfaces. Interfaces define a contract between two communicating media. In other words, you can strictly specify some rules that identify a type. By the way, it is also important to understand that an Interface is only a type that exists inside an assembly, and not implemented from the base class `System.Object`. Even though in the final implementation of the interface actually derived from `System.Object`, interface does not implement `System.Object` inside IL.

Structures are another important language feature that exist in the .NET language. Any primitive type is defined as a structure. For instance, `int` in C# is actually `System.Int32` which is derived from `System.ValueType` and is a structure. Structures are primitive value types and defined as immutable objects. In our case, we have defined a custom structure that has two members, one is an integer and the other is a string. Structures do not support inheritance.

The most important part of a .NET language is a class. Any logic that you define should remain inside a class. A class supports all the OOP features, such as inheritance, polymorphism, abstraction, encapsulation, and so on.

A delegate is a special type defined in .NET Framework which is derived from `System.Delegate`, or rather `System.MulticastDelegate`. When you define a delegate in your program, it actually produces a type during compilation which can collect multiple methods. The methods that are passed within a delegate are actually stored inside a collection and when we invoke a delegate all the methods that we pass inside it will be called sequentially. C# also supports anonymous delegates.

An event is a wrapper for a delegate. Generally as we know .NET recommends not exposing variables but rather to use properties to get the value of a member, in a similar way delegates are actually wrapped around using an event. If we make an event, a caller to the object can subscribe an event and pass a method that will be stored inside the delegate type. An event is exposed using an event accessor. Even though C# allows you to specify public events, it wraps the event inside an event accessor during compilation.

Finally, the `enum` is an enumerated data type. Enumerated data type allows you to specify named constants for its values. Generally an `enum` is treated as an integer, but we can specify its type explicitly. All are derived from `System.Enum` and bears special meaning to the compiler.

While creating the language C#, the team has modified a few types and added special meaning to them. The one major aspect is `System.ValueType` which is identified by the compiler and restricts inheritance, and makes it immutable. Similarly, delegate is also treated specially. It identifies the `System.Delegate` type and produces special meaning to the compiler. Thus after reading the recipe, you must be clear that these language features are actually specially added to help the programmer to a greater extent. This is why people love this language so much.

There's more...

Language features are endless. Even though we have identified some of the basic language features of the language, we still left out some of other important elements that need special treatment. Let's jot down them one by one.

What are anonymous types?

Did you know C# 3.0 and above allows you to create anonymous types using Automatic Property Initializers? Yes, you can declare a type directly while you code to hold abstract data. Let's see how you can do this:

```
var x = new { X = 20, Y = new MyObj(), Z = "New String" };
```

Well using this code, the CLR compiler will automatically generate a new concrete class with read-only properties `X`, `Y`, and `Z` having its data types `int`, `MyObj`, and `string` respectively. Hence, in your code you can use this object, and later on if you want to declare an object with the same type you can do so using the same structure.

C# compiler takes care of this code during compilation and produces concrete anonymous types in MSIL.

But while you work with those objects, you should remember that these objects have the following restrictions:

- You cannot declare a type without initializing it
- All the properties they produce will be read-only by default
- You cannot declare methods for these types; C# compiler does not support that
- They directly inherit from object, you cannot inherit the type from your custom interface
- You should use implicitly type `var` while holding the objects, even though you have the option to hold them in dynamic objects introduced in C# 4.0

What are anonymous methods?

Anonymous methods are special chunks of code that can individually act as a complete function without a name. In .Net 2.0 Microsoft introduced a way to define an anonymous method which can be assigned using a delegate, and can be called or invoked directly. Let us look into an example:

```
delegate void Del(int x);

// Instantiate the delegate using an anonymous method.
Del d = delegate(int k) { /* ... */ };
```

So here the function that is assigned to the delegate `d` does not have its corresponding name, but yet it can be called and used in the same way as the named function does. One important thing about anonymous delegate is that you can omit the parameter list from the method. The compiler is smart enough to actually convert the inline closure block into its appropriate signatures.

While defining a delegate you can get local variable into its scope, and can also pass this inline method to another method from where it could be invoked. This is actually the easiest way of defining inversion control into your code.

Anonymous methods are widely used nowadays in the form of Lambda expressions. To define a closure we are now hugely dependent on Lambda expressions that enhance the way of delivering our code and callbacks. We will discuss Lambda expressions in more detail later.

Are the ValueType classes always allocated in a stack?

There is a possible myth that says "The `ValueType` classes are always stored in stack and reference type are allocated in heaps". Many of us know this statement, but never really understand what the meaning of it is. We know that the `ValueType` classes are stored into a stack, but is it true always? The answer is no. A reference type is an encapsulation of the `ValueType` classes which are declared as a member variable of a type, or it is defined on a type that is declared inside this type. So if you think that every member defined within a Type is actually stored in stack, you would just see that every memory that you create is actually created in a stack. The statement misleads us because according to the rule, the `ValueType` classes are only stored into a stack when they are defined inside a scope as local variables. A stack is associated with a thread in such a way that when the thread comes to a method, all the `ValueType` classes are created in the stack and when exiting the method, those values will get cleared.

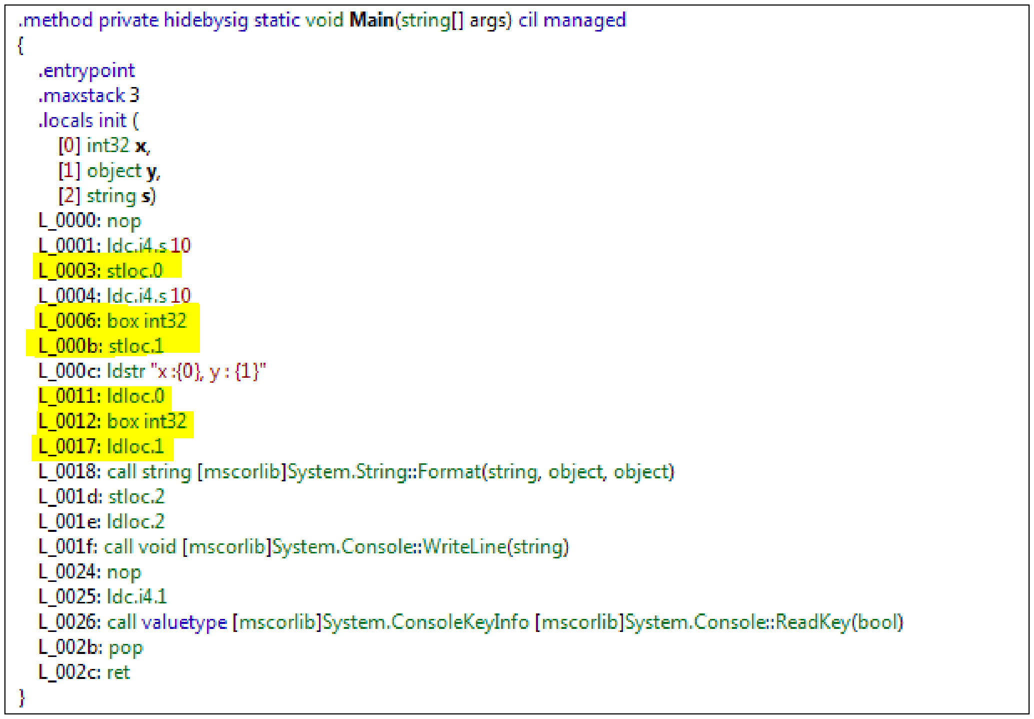
The reference type on the other hand is always created in heap, but the reference pointer pointing to the actual object created in the heap is created on the stack. This is some times called as GC roots.

Let us take an example:

```
int x = 10;

object y = 10;

string s = string.Format("x :{0}, y : {1}", x, y);

Console.WriteLine(s);
Console.ReadKey(true);
```

This is basically a very simple code, where I first created one `ValueType` (`int`) an object `y` which is a reference type and formatted them into a string variable, and loaded the string into the `string` variables.

Now let us see how the IL for the same looks like:



```
.method private hidebysig static void Main(string[] args) cil managed
{
   .entrypoint
   .maxstack 3
   .locals init (
      [0] int32 x,
      [1] object y,
      [2] string s)
   L_0000: nop
   L_0001: ldc.i4.s 10
   L_0003: stloc.0
   L_0004: ldc.i4.s 10
   L_0006: box int32
   L_000b: stloc.1
   L_000c: ldstr "x :{0}, y : {1}"
   L_0011: ldloc.0
   L_0012: box int32
   L_0017: ldloc.1
   L_0018: call string [mscorlib]System.String::Format(string, object, object)
   L_001d: stloc.2
   L_001e: ldloc.2
   L_001f: call void [mscorlib]System.Console::WriteLine(string)
   L_0024: nop
   L_0025: ldc.i4.1
   L_0026: call valuetype [mscorlib]System.ConsoleKeyInfo [mscorlib]System.Console::ReadKey(bool)
   L_002b: pop
   L_002c: ret
}
```

Now here let's take a look at the IL. The line says `.entrypoint`. `EntryPoint` identifies the start of the program. So if you declare your method as `Main` in .NET, `.entrypoint` will automatically be written on it. So it is also a compiler trick which writes `.entrypoint` correctly.

The second line says `.maxstack 3`. Here `maxstack` will indicate that the method allocates three units of stack for the current method. For our code, it will be:

- To store the integer value
- To store reference to the type `y` while the value is stored in heap
- To store the reference to the type `s` while the value is stored in heap

Hence you should remember that even though you create an object or a memory block that is allocated in heap, the reference pointer is still allocated inside your stack, which de-references itself once the code execution ends. When all the references to a memory are de-referenced, it is exposed to the GC for collection.

So the next statement creates three locals for the code block in a stack. Locals indicates the stack allocation.

How to define flags enumeration and description to a value?

`enum` is an enumerated data type. There is often a need to have a bit field in a program, and also you want to specify proper names for the enumerated values. For instance, let us suppose I want an enumeration which defines `FileAttribute`. A file can be encrypted, read-only, hidden, or even compressed. To deal with such a situation, we use bit field operation in such a way that we can define a combination of more than one value. Let us see how to define a flags enumeration:

```
[Flags]
public enum FileAttribute
{
  None = 0,
   ReadOnly = 2,
    Hidden = 4,
  Encrypted = 8,
  Compressed = 16
}
```

Here in the preceding code, the enumeration field is used as flags. We annotate a bit field enumeration. We need to annotate the enumeration declaration using the flags attribute. Again while defining the values we need to specify the values of the attribute options as a power of 2.

Now, let us consider why it is important. As there can be multiple values for individual enumeration objects, we can specify a combination of them in our project.

```
var theFileAttribute = FileAttribute.ReadOnly | FileAttribute.
Encrypted;
```

So in the preceding line the file is both read-only and encrypted. This is a valid statement as the value evaluated is 10.

There is no specific way to get the description of an enumeration. We need to use reflection to get the description of an enumeration. Let us take an example :

```
public enum Standard
{
    [Description("Failed students are returned to their old classes")]
    Fail,
    [Description("Passed students are sent to new class")]
    Pass,
    [Description("Distinctive students will be honored and specially
trained")]
    Distinction
}
```

Each simple enumerated field define its own description. Now sometimes from our code we need the actual description of a value of an enumeration. The enumerations are here marked with a `DescriptionAttribute`. So to fetch the appropriate description for a value we need to first determine the exact description attribute for that value. Let us suppose the value is set as `Standard.Distinction`. To get the description we use the following code:

```
public static string GetEnumDescription(Enum value)
{
    FieldInfo fi = value.GetType().GetField(value.ToString());

    DescriptionAttribute[] attributes =
        (DescriptionAttribute[])fi.GetCustomAttributes(
        typeof(DescriptionAttribute),
        false);
    if (attributes != null &&
        attributes.Length > 0)
        return attributes[0].Description;
    else
        return value.ToString();
}
```

Here in the preceding code we use the enum data type to get the exact `CustomAttribute` that is assigned to the value. The attribute indicates which description is associated with the current value. So if I call `GetEnumDescription((Enum)Standard.Distinction)`, it will get me the string associated with the description.

Even though we needed to use this using reflection, I strongly believe this API should exist in .NET Framework library in near future.

See also

- `http://bit.ly/Language-Features`

Advantages of Generics in .NET

Generics is one of the best addition to .NET. It was not present in the first release of .NET, but after its introduction in .NET 2.0, Microsoft recommends the use all of its generic types rather than its counterpart non-generic implementations. Generics are template objects that support the specification of one or more types during object creation, but holding the reusability, type safety, and efficiency of the code. Generics are mostly common in collections, but one is free to use the flexibility of Generics on any type. Hence it is important to note that in current situations, it is recommended to avoid collections like `ArrayList`, `HashTable`, rather, it is recommended to use `List<T>`, `Dictionary<Key, Value>` instead. The old collections hold items as objects, and hence the inherent properties of the type are lost when it is boxed into the `System.Object` type.

On the contrary, generic collection allows you to pass the type as a specific type during its construction, which means that all inherent properties of the object remains intact. In other words, when we add an object, say of type integer, it will be boxed when it gets stored into an `ArrayList`, whereas when we use `List<int>` the whole collection is actually a list of integers. So there will be no boxing to `System.Object` or even hold of the explicit type safety to the collection. That means you cannot add a string to `the List<int>` collection, but `ArrayList` can.

How to do it...

To understand Generics, let us create a Console application and start creating few classes.

1. Start a Console application and create a class. We'll call it `NormalClass`. This class does not implement any generic template.

   ```
   public class NormalClass
       {
           public DateTime FirstMember { get; set; }
       }
   ```

   You can see that `NormalClass` has a member which is of the type `DateTime`. Now the problem with this class is as C# is type safe, we need to explicitly pass the object as `DateTime`.

2. Now let us define an interface which we will use later in the recipe. We'll call it `MyInterface`.

   ```
   public interface MyInterface
    {
           string StringResult { get; set; }
    }
   ```

   The interface looks simple with one property of type `string`.

3. Now let us define a class that implements this interface. We are going to implement this interface with the property so that we can use it later.

   ```
   public class StringImplementer : MyInterface
   {
     public string StringResult { get; set; }
   }
   ```

4. Finally, let us define an interface for our generic implementation. As Generics supports the interface types as well, we can define an interface like this:

   ```
   public interface GenericInterface<T, T1, T2> where T : MyInterface
                                    where T1:class
                                    where T2 : struct, new()
       {
       }
   ```

So here the generic interface takes three types `T`, `T1`, and `T2` which you can pass through while creating an object. If you see the preceding code, it clearly states the constraint that the first template object `T` needs to be a type that explicitly implements the interface `MyInterface`. The other types `T1` and `T2` are of object type and `valuetype` respectively. The new constraint defines that the class which implements this interface should contain a default constructor.

You must also note that a constraint must always be an interface or a non-sealed class.

5. Now just to implement the interface, we either need to pass the concrete type or pass an explicit type to be specified by the end user.

```
public class GenericImplementer<T1, T2> : GenericInterface<StringI
mplementer, T1, T2>
{
    public T1 FirstMember { get; set; }
    public T2 SecondMember { get; set; }
    public StringImplementer GetString()
    {
      StringImplementer implementer = new StringImplementer();
      implementer.StringResult = string.Format("The class passed
are {0} and {1}", this.FirstMember, this.SecondMember);
      return implementer;
    }
}
```

   Here in the preceding type declaration, we specify the first type as a concrete type called `StringImplementer`, and the other types `T1` and `T2` will be externally sent when the object is created.

6. Finally from the `program.cs` file of the Console application, let us create an object of the type using the following code:

```
NormalClass nobj = new NormalClass();

GenericImplementer<NormalClass, int> gobj = new
GenericImplementer<NormalClass, int>();
gobj.FirstMember = nobj;
gobj.SecondMember = 30;
var stringresult = gobj.GetString();
Console.WriteLine(stringresult.StringResult);
Console.ReadKey(false);
```

   Here in the example code, we first create an object of type `NormalClass`. You can see that the type is strict and has only a member of `DateTime`. It is not flexible enough to take parameters and mould the type to any type other than `DateTime`.

On the contrary, `GenericImplementer` can specify what type it wants to specify. Just as you can see in the preceding code we specified `FirstMember` to be of type `NormalClass` and `SecondMember` of type `int`.

How it works...

Generics is a special feature that has been supported by CLR and also by the languages that are implemented on top of it. When you compile C#, the assembly contains parameter placeholders for a specific type. Additionally, the metadata of the generic constraint is also embedded inside the type.

While implementing the types when we pass the type as parameters, it creates a placeholder for the parameter inside the type. These types can specify constraints which are explicitly checked by the compiler itself when the object is being created. The constraint can be either an interface, or a non-static type, or even specified as a class or struct. When we specify a class, it means any type can be passed to the placeholder and when it is struct, it can only take a structure.

Generics parameters can be applied to a class as I have showed you in this recipe; in fact you can pass generic parameters directly inside a method too. For instance, consider the following example:

```
public key CallGenericMethod<key, value>(key k, value v) where value :
class
        {
            if(v!= null)
                return k;

            return default(key);
        }
```

Here in the example while calling the method we need to pass two parameters, where the second one is specified as an object type (we cannot pass a value type as second parameter). Here in the preceding code we first check whether the value that is passed as the second parameter is null or not. If it is null, we return the default value of the type passed.

The default value for a reference type is always null, but for a value type it differs and will be the initial value of it. For instance the default value of an `int` is zero.

There's more...

Even though we have already demonstrated Generics in this recipe, there are a few interesting things that are left behind. Let us discuss them.

What do you mean by covariance and contravariance?

If you literally think of variance, you would say variance is a concept in which if one data varies at a certain time, the other one will vary automatically. And rightly so. In case of .NET, variance comes just in such a way. Let's talk about the need of variance or covariance in .NET.

Before we do, have you ever done something as follows?

```
IA a = new A ();
List<IA> aa = new List<A>(enumerables);
```

The two lists will not compile before .NET 4.0. It will throw `InvalidCastException` as it doesn't know if `List` is actually a list for each enumerable.

Why is this so?

Say I write the following:

```
aa.Add(new B()); //Considering B implements IA
```

This will create problems, as `aa` cannot be cast to `List` anymore. To address such situations .NET 4.0 introduces **covariance and contravariance**.

Let's take another example to clear this:

```
    public interface IPeople
    {
        string GetName();
    }

    public class RuralPeople : IPeople
    {

        #region IPeople Members

        public string GetName()
        {
            return "RuralPeople";
        }

        #endregion
    }

    public class UrbanPeople : IPeople
    {

        #region IPeople Members

        public string GetName()
        {
            return "UrbanPeople";
        }

        #endregion
    }
```

Here the two classes `RuralPeople` and `UrbanPeople` are implemented from `IPeople`. Now, let's define a writer which writes the collection of names of these classes:

```
interface IWriter<T>
    {
        void WriteNames(IEnumerable<T> ts);
    }

 public class Writer<T> : IWriter<T> where T : IPeople
 {
        public void WriteNames(IEnumerable<T> ts)
        {
            foreach (var t in ts)
            {
                Console.WriteLine(t.GetName());
            }
        }
    }
```

Now if I write:

```
List<RuralPeople> ruralist = new List<RuralPeople> { new RuralPeople()
};
IWriter<IPeople> writer = new Writer<IPeople>();
writer.WriteNames(ruralist);
```

This will produce a compiler error in .NET 3.0. This is because of the fact that `IPeople` cannot write names from `List`. In .NET 4.0, we solve this issue by writing the `out` parameter in `IEnumerable`.

```
public interface IEnumerable<out T> : IEnumerable
{
}
```

The `out` keyword in `T` makes `T` covariant, which means you can pass any value to `T` that is derived from `T`. In our case, `IPeople`. Hence, we can now pass `IEnumerable` of `RuralPeople` and `IEnumerable` of `UrbanPeople` easily to `IPeople`, and it will hold its variance completely.

The generic `out` parameter restricts the class to only return Ts to the outside environment. Hence `T` can occur only as an output parameter. Hence you cannot pass `T` to the class from outside. Thus, this ensures that `List` is actually either List or List, and your add method does not allow to pass `IPeople` anymore.

Now if we want to use variance, but still need to create an instance of `T` inside the class, contravariance deals with such situations. You can annotate the generic parameter with `in` keyword, and it will ensure that `T` can be used only in input positions.

```
    interface IWriter<in T>
      {
          void WriteNames(IEnumerable<T> ts);
      }
```

This will make the following lines legal:

```
List<RuralPeople> rurals = new List<RuralPeople> { new RuralPeople()
};
List<UrbanPeople> urbans = new List<UrbanPeople> { new UrbanPeople()
};
IWriter<IPeople> peoples = new Writer<IPeople>();
IWriter<RuralPeople> ruralwriter = peoples;
IWriter<UrbanPeople> urbanwriter = peoples;
ruralwriter.WriteNames(rurals);
urbanwriter.WriteNames(urbans);
```

Hence you can say that `RuralPeople` and `UrbanPeople` is contravariant to `IPeople`. Consequently, you can assign instance of `Writer` of `IPeople` to `IWriter` of `RuralPeople` or `UrbanPeople` easily and it will process both `WriteNames` appropriately.

This is not the end of it. Generic variance is also spread over delegates. In a similar way, you can use Generic covariance and contravariance for delegates too.

See also

- `http://bit.ly/Generics-Basics`

Using iterator blocks in .NET

Iterators are one of the most interesting additions to the .NET features in .NET 2.0. When the word iterator comes into our minds, we often think of `IEnumerable`. But do we actually know what exactly an `IEnumerable` is? Every collection that exists in the .NET environment automatically implements an `IEnumerable` interface, which needs an `IEnumerator` internally to iterate between a sequence.

So .NET iterator is actually based on two most important interfaces, `IEnumerable<T>` and `IEnumerator`. We use generic one as we already know the benefits of generics in .NET.

Getting ready

C# comes with two basic interfaces, namely `IEnumerable`, and `IEnumerator` that represents the base for any collection. `IEnumerable` is an interface that defines a `GetEnumerator` which gets an `IEnumerator`. An `IEnumerator` on the other hand provides a simple iteration over a collection. Using the interface ensures that you could use this collection in `foreach` loop of C# or `for each` in VB.NET. If you look back to MSDN, it says:

> *"IEnumerator is the base interface for all enumerators. Enumerators only allow reading the data in the collection. Enumerators cannot be used to modify the underlying collection.*
>
> *Initially, the enumerator is positioned before the first element in the collection. Reset also brings the enumerator back to this position. At this position, calling Current throws an exception. Therefore, you must call MoveNext to advance the enumerator to the first element of the collection before reading the value of Current."*

Almost all the collections in .NET class library are derived from `IEnumerable`, and hence you can iterate through the collection it internally holds and use it.

`IEnumerable` defines a member called `GetEnumerator`. The `GetEnumerator` returns an `IEnumerator` which allows you to traverse a sequence. The `IEnumerable` and `IEnumerator` patterns are followed by all `IList`, `ICollection`, and so on. So these patterns are very important to understand. Let me define `IEnumerable` and `IEnumerator`.

```
public Interface IEnumerable<out T>
{
  IEnumerator<T> GetEnumerator();
}
public interface IEnumerator<T> : IDisposable
{
        T Current { get; }
        bool MoveNext();
        void Reset();
 }
```

So, `IEnumerable` actually creates an object of `IEnumerator` which allows you to iterate on a sequence. Every time you call an object from `IEnumerable`, `IEnumerator` actually calls `MoveNext` to hold the current object in the `Current` property. This object is then returned from `IEnumerable`. The generic implementation of `IEnumerator` has the property `Current` as template object.

The basic look of the calls to an `IEnumerable` looks like the following:

```
IEnumerator<int> iterator = yourcollection.GetEnumerator();
while (iterator.MoveNext())
{
    int data = iterator.Current;
    // Your custom logic around data is here.
}
Iterator.Dispose();
```

Now iterators in C# even though uses these interfaces are not actually the same if you implement an `IEnumerator` yourself and return it from an implementation of `IEnumerable`. The iterator block that is self-generated by the compiler actually holds a state-machine inside it.

There is also a special interface called `IQueryable` which is used for query providers. The .NET Framework associates any expression tree data execution to a specific query provider which is only supported by `IQueryable` interface that is actually inherited from `IEnumerable`.

How to do it...

In this recipe, we will see what exactly the .NET iterator block is and how to write an iterator.

1. Open the Console application and create a new class.
2. In the class, write the following code:

   ```
   public IEnumerable<int> MyIteratorBlock(int number)
           {
               int i = 0;
               while (i < number)
                   yield return i++;
           }
   ```

3. In the preceding code we have just send a number to the method and the method will loop through and return each number. Now if you look carefully, you see rather than returning the actual value of `i` we did it using the `yield` keyword. Also, even though each time the method executes it returns an integer value we still use `IEnumerable<int>` as its return type. This is confusing right?

   C# 2.0 introduced a keyword `yield` that allows you to create the `IEnumerable` objects automatically. So when we use `yield return i`, the value from the `i` is returned, and also the method is paused with all active states held.

4. We can either use `yield return` or `yield break` to our code that will generate `IEnumerable` automatically. Let us write the code in step 2 a little differently.

   ```
   public IEnumerable<int> MyIteratorBlock(int number, int
   ```

```
breaknumber)
        {
            int i = 0;
            while (i < number)
            {
                if (i == breaknumber)
                    yield break;
                yield return i++;
            }
```

The preceding code is slightly different from the other. It sends one extra parameter such that when `breaknumber` is encountered between the sequence it invokes a `yield break`. The yield break ends the `IEnumerable`.

5. Now if I run both the code using the (10, 7) parameter set, the enumerable will generate 0 to 6, and when I pass (10,12) it generates 0 to 9, as 12 is outside the normal execution path.

How it works...

The preceding code actually executes sequentially and generates the collection from the machine. Now let us step into the code and see how it works exactly. If you write few lines of code, you could see that when you call the method from your code the lines inside the method did not run, rather the lines will run when actually you want to get object from the iterator block.

In fact, the C# iterator internally holds a state machine for each iterator. The state machine is actually a `CompilerGenerated` class which is capable of storing the local variable as properties of the class, the execution point as delegate, and so on. Thus the state machine allows you to pause and resume execution of the block as and when required.

This is a very cool concept. Let me demonstrate the fact with an example:

```
public IEnumerable<int> GetFirst10Nos()
{
    for (int i = 0; i < 10; i++)
        yield return i;
}
```

This is the most simple method which returns the first 10 numbers starting from zero. Now let's see how it looks like after compilation:

```
public IEnumerable<int> GetFirst10Nos()
{
    <GetFirst10Nos>d__0 d__ = new <GetFirst10Nos>d__0(-2);
    d__.<>4__this = this;
    return d__;
}

// Nested Types
[CompilerGenerated]
private sealed class <GetFirst10Nos>d__0 :
  IEnumerable<int>, IEnumerable, IEnumerator<int>, IEnumerator,
IDisposable
{
    // Fields
    private bool $__disposing;
    private bool $__doFinallyBodies;
    private int <>1__state;
    private int <>2__current;
    public Iteratordemo <>4__this;
    private int <>l__initialThreadId;
    public int <i>5__1;

    // Methods
    [DebuggerHidden]
    public <GetFirst10Nos>d__0(int <>1__state)
    {
        this.<>1__state = <>1__state;
        this.<>l__initialThreadId = Thread.CurrentThread.
ManagedThreadId;
    }

    private bool MoveNext()
    {
        bool CS$1$0000;
        try
        {
            this.$__doFinallyBodies = true;
            if (this.<>1__state == 1)
            {
                goto Label_0068;
            }
            if (this.<>1__state == -1)
            {
                return false;
            }
            if (this.$__disposing)
            {
                return false;
            }
            this.<i>5__1 = 0;
            while (this.<i>5__1 < 10)
            {
                this.<>2__current = this.<i>5__1;
                this.<>1__state = 1;
                this.$__doFinallyBodies = false;
                return true;
            Label_0068:
                if (this.$__disposing)
                {
                    return false;
                }
                this.<>1__state = 0;
                this.<i>5__1++;
            }
            this.<>1__state = -1;
            CS$1$0000 = false;
        }
        catch (Exception)
        {
            this.<>1__state = -1;
            throw;
        }
        return CS$1$0000;
    }

    [DebuggerHidden]
    IEnumerator<int> IEnumerable<int>.GetEnumerator()
    {
        if ((Thread.CurrentThread.ManagedThreadId ==
  this.<>l__initialThreadId) && (this.<>1__state == -2))
        {
            this.<>1__state = 0;
            return this;
        }
        Iteratordemo.<GetFirst10Nos>d__0 d__ = new Iteratordemo.<GetFi
rst10Nos>d__0(0);
        d__.<>4__this = this.<>4__this;
        return d__;
    }

    [DebuggerHidden]
    void IDisposable.Dispose()
    {
        this.$__disposing = true;
        this.MoveNext();
        this.<>1__state = -1;
    }

    // Properties
    int IEnumerator<int>.Current
    {
        [DebuggerHidden]
        get
        {
            return this.<>2__current;
        }
    }
}
```

Well, basically the compiler generates a type for holding the state machine for you. The type is generated in such a way that it implements the `IEnumerator`, so that it can produce the iterators and hold the state of the method within itself. Let me explain few methods for you:

- Our method actually creates a nested class `d__0` which holds the state machine, and also implements `IEnumerable` and `IEnumerator`. Once our method is called, it creates a new object of it and returns back the object. As the class implements `IEnumerable`, it doesn't produce any problems. I should remind you, no code from our method has been executed yet.
- Initially, when we use `IEnumerable` in the `foreach` loop, it internally calls `GetEnumerator`. If you look closely, this method checks if the call is made from the current thread or not, and also checks for the state to be `-2`. You can see, while creating the object, it passes the state as `-2`. Hence to conclude, `GetEnumerator` always creates a new object of enumerator if the call is made either for the first time, or through a different thread than the one which owns it. You should note that while creating the object from `GetEnumerator`, the object is initialized to `0`, which states that the enumerator is initialized.
- `MoveNext`, being the important part of the object, actually checks the value of the state to indicate the various stages of the object.
- `0` represents before calling `MoveNext`.
- `-1` at the end of the enumerator, returns false.
- `-2` represents no enumerator is fetched (before calling `GetEnumerator`).
- `1` represents the enumeration in running, sets the value of `this.<>2__current` and returns true.

- Now as for each request to `MoveNext`, the state is checked and the initial `GoTo` statement moves the control to `Label_0068:` , the object keeps on running our code and starts producing numbers.
- Finally, when the `while` loop fails to satisfy the condition, the state is set to `-1` and the execution terminates.

So, the state machine object is capable of producing numbers and also to pause and resume the method.

What does the member variable of state machine represent?

Locals, parameters, and so on are created as member variables in such a way that local variable `i` is represented as `<i>5__1`. Two Boolean variables hold the state of disposing and finally execution, that is `$__disposing` and `$__doFinallyBodies`. The current value of the object is represented in `<>2__current`, the state in which the object is also represented (even though the state is not given any enumerated names) in `<>1__state`. It also stores object which invokes the iterator, `<>4__this`.

You should note that the variables, methods, and types are generated in such a way that they don't represent a valid C# type, and thus eliminates the occurrence of another type of the same name in the assembly.

There's more...

Iterators open a wide door for us. We can make use of them to implement lot of functionalities. Let us quickly look at a few more things.

Iterators as sequence generators with rules

Iterators acts as a state machine. Hence, it is important to note that we can create a sequence of steps as an enumerable. `IEnumerable` of objects is actually not a collection of values, rather it can be a sequence generator. Let us look into the code:

```
public static IEnumerable<int> FibonacciGenerator(Func<int, bool>
abortCondition)
{
    int a = 0, b = 1;
    int c = 0;
    while (abortCondition(c))
    {
        c = a + b;
        yield return c;
        a = b;
        b = c;
    }
```

This is actually a Fibonacci series generator. We pass a delegate as an argument which determines the exit point of the sequence. If we pass e => true as the Lambda expression to this delegate, we will have an infinite series of Fibonacci:

```
var fen = FibonacciGenerator(e => e <= 100);
            foreach(int i in fen)
                Console.WriteLine(i);
```

The code will print the Fibonacci series until it reaches 100. The state machine maintains its value for each object called from it.

Iterator and finally block

One thing you must remember is that you cannot wrap a `yield` statement inside a `try` block which has a `catch` in it. C# compiler limits you to write a `catch` block because of a `yield` statement. But you can try this with the `finally` block.

Yes, C# compiler generally ensures that the `finally` block will execute at least before going out of the scope of the method. As I showed before, the state machine completely rewrites the block of an iterator into multiple methods. Let's try with `finally` and see what exactly we get inside it.

```
try
{
    int a = 0, b = 1;
    int c = 0;
    while (abortCondition(c))
    {
        c = a + b;
        yield return c;
        a = b;
        b = c;
    }
}
finally
{
    Console.WriteLine("We made it to the finally!");
}
```

Now if we run this code, we will see that the `finally` block will be called only when `yield` finishes. This is a smart approach by the compiler. It creates a method, names it finally, and calls it when the enumeration is exited.

```
private void <>m__Finally6()
    {
        this.<>1__state = -1;
```

```
            Console.WriteLine("We made it to the finally!");
        }
```

The method will be called from the code only when the state represents the exit of the enumeration.

What are collections in .NET and how do they work?

A collection is a group of related objects that are stored in a single plural object. A collection in .NET starts from an array, which is derived from `System.Array` to more usable types, such as `ArrayList, List<T>`, and so on. Unlike arrays, the collection objects have the capability to grow or shrink based on the requirement such that an object can be added or removed from a collection whenever needed.

Even though collections are derived from `IEnumerable`, there are other interfaces too that have been used by the collection types, such as `IList`, `ICollection`, and so on.

The different types of collections are as follows:

- `System.Collections`: These are lists that do not store elements as specifically typed objects. They are general collections that store elements into objects. For instance, `ArrayList`, `Hashtable`, `Queue`, `Stack`, and so on. Depending on whether the type of the object is `ValueType`, these collections are prone to box/unbox elements while adding or removing objects.
- `System.Collections.Generic`: Generic collections are special collections that stores the type of information in addition to the elements. The generic collection is useful when every object in the collection is of the same data type. For example: `Dictionary<TKey,TValue>`, `List<T>`, `Queue<T>`, `Stack<T>`, `SortedList<TKey,TValue>`, and so on.
- `System.Collections.Concurrent`: Concurrent collections are thread safe version of lists. The objects can be added or removed without compromising the thread safety in such a way that multiple threads can access them concurrently. For example, `BlockingCollection<T>`, `ConcurrentDictionary<TKey,TValue>`, `ConcurrentQueue<T>`, `ConcurrentStack<T>`, and so on.
- `ReadonlyCollection`: A read-only collection exposes **read-only** implementation of a collection, which means that when a collection is exposed as read-only, it will not allow the user to modify it. `IReadonlyCollection<T>` is another implementation of `IEnumerable`, but adds the `Count` property so that `ReadonlyCollection` is deterministic. A read-only collection is immutable or frozen and cannot be changed once created. They do not allow addition or removal of elements from it.

The collection classes form an important part of .NET Framework. We generally don't rely much on arrays in .NET languages, and large implementation of collection types really help in building day-to-day applications.

Projections of a collection

In addition to the actual array storage classes, there is a special class called `ArraySegment` that projects on a specific portion of an array. To use it, we pass an array to it and offset of the array index, and the object projects the specific section of the array that has been projected.

```
int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
ArraySegment<int> segmentRange3to8 = new ArraySegment<int>(array, 3,
6);
foreach (int i in segmentRange3to8)
    Console.Write(i);
```

The `ArraySegment` class takes an offset and the count of the elements that it needs to process, and creates a new segment of an array. In case of `ArraySegment`, the actual objects are not copied and no extra memory is used. The preceding code prints from `3` to `8`.

Working with LINQ and Lambda expressions

LINQ is a Microsoft implementation for integrating search queries in .NET languages. We know that it is really easy to find data from SQL objects simply by writing a query, while it's somewhat hectic when we want to do the same thing in a `DataTable` or Lists. Generally, we will have to loop through every element to find the exact match. If there is some aggregation we need to aggregate the values. `LINQ` provides an easy way to write queries that can run with the in memory objects.

Lambda expression is nothing but a syntax of writing an anonymous method directly in the code. With the introduction of Lambda expressions, developers can easily write methods on the fly. .NET introduces generic delegates and also uses them in their base class library to show the world where exactly the Lambda expressions are suited. The BCL-defined general purpose delegates, such as `Action` and `Func` are widely used. You can pass your inline logic in these extension methods to get a new enumerable or some result based on what it is exactly for. For instance, say you have a collection of objects of the `User` type and you want to find the users whose name starts with A. You can directly write:

```
var usersWithA = this.AllUsers.Where(e=> e.Name.StartsWith("A"));
```

Now in the preceding statement the Lambda will be called for every instance of `User` inside the `AllUsers` collection and evaluate whether the name starts with A. If this Lambda returns true, the object will be selected otherwise will be rejected.

LINQ and Lambda expressions are very important in the context of current trend of development. In this recipe, we are going to spend some good time understanding the syntax and insights on how we can use these little gems in a better way in our code.

Getting ready

Before we get started, we need to have a clear idea on what anonymous methods are and how to use them. Let us take a look at how to write anonymous methods in C# 2.0.

```
int i = this.compute(10);
private int compute(int value)
{
   return (value + 2);
}
```

It is a very normal example of a function call. The `compute` function is called when the line `i=this.compute(10)` is called. We can replace the preceding code with inline anonymous function like the following:

```
delegate int DelType(int i);
DelType dd =  delegate(int value)
              {
               return (value +2);
              };
int i = dd(10);
```

In the second case we can see that by declaring a delegate and giving an instance of that delegate type, we can easily manage writing anonymous methods. .NET 3.5 made this concept a little more compact. We can take the use of `=>` operator which was introduced in .NET 3.5. It gives more flexible and easier ways to write expressions. Lambda expressions can be used in the previous case to get the result as follows:

```
delegate int DelType(int i);
DelType d = value => value + 2;
int i = d(10);
```

Thus the delegate type is assigned directly. The meaning of the line value `=>value + 2` is just similar to declaring a function. The first value indicates the arguments that we pass in a function and the one after the `=>` operator is the body of the function. Similarly, if we want we can feed in as many arguments as we want to, as in the following example:

```
// Method with 2 arguments
delegate int DelType(int i, int j);
DelType d = (value1,value2) => value1 + value2;
int i = d(10,20) // Returns 30
```

In this example, we passed in two arguments to the function and returned the result. You may also wonder how I can write a Lambda expression when my function body takes more than one expression. Don't worry, it's simple. Take a look at the following example:

```
// Multiline Function Body
delegate int DelType(int i, int j);
```

```
DelType d = (value1,value2) => {
            value1 = value1 + 2;
            value2 = value2 + 2;
            return value1 + value2;
};
```

Thus we see that writing a Lambda expression is very easy. This comes very handy when working with LINQ, as most of the extension methods that are associated with LINQ takes function delegates as arguments. We can pass simple delegate function body easily to those functions to perform work easily.

Now let us move to LINQ and see how similar it is to write LINQ queries just like the Lambda expressions. To work with this recipe, I am going to create two lists to demonstrate the LINQ operations.

```
List<Employee> employees = new List<Employee>();
List<Employee> employee1 = new List<Employee>();
List<Order> orders = new List<Order>();
//Where Employee and Order are classes.
public class Employee
{
    public string name;
    public int age;
    public override string ToString()
    {
        return this.name;
    }
}
public class Order
{
    public string itemName;
    public string empName;
    public override string ToString()
    {
        return this.itemName;
    }
}
```

Here I have created two classes, one deals with the employees and another deals with orders. Now let us query into these classes and try to understand LINQ better. While writing queries, you will find a number of operators that we can use. The most general operators are:

- Projection operators (`Select`)
- Restriction operators (`Where`)
- Partitioning operators (`Take/Skip`, `TakeWhile/SkipWhile`)

- Join operators (`Join`)
- Concatenation operator (`Concat`)
- Ordering operators (`order by`)
- Grouping operators (`group by into`)

Now let us apply these operators one by one.

How to do it...

1. With the word projection, we mean the `Select` statements. Every LINQ query supports projection. To do this we use the following code:

   ```
   var iNames = from i in employees
                select i.name;

   //Using Lambda Expression
   var iNames = employees.Select<Employee, string>(r => r.name);
   ```

   Here `IEnumerable` of string is returned. In the preceding example, we are also taking advantage of anonymous type declaration that was introduced in .NET 3.5.

2. As every query supports the projection operations, LINQ does as well. You can use the `Where` extension method or the `where` clause in the collection to fetch the filtered collection, as shown in the following example:

   ```
   var filterEnumerable = from emp in employees
                          where emp.age > 50
                          select emp;

    //Using Lambda Expression
   var filterEnumerable = employees.Where<Employee>(emp => emp.age >
   50);
   ```

   So in the preceding code, it selects all the employees who are above 50 years of age.

3. Partitioning is also supported with LINQ. You can partition the data by using the `Take` or `Skip` extension methods. For instance:

   ```
   var filterEnumerable = (from emp in employees
                            select emp).Take<Employee>(2);

    //Using Lambda Expression
   var filterEnumerable = employees.Skip<Employee>(2);
   ```

Here the `filterEnumerable` will take the first two elements from the LINQ statement, while the second one takes all the elements by skipping the first two elements. You can use both `Skip` and `Take` together to select any portion of the enumerable. For instance:

```
var filterEnumerable = employees.Skip<Employee>(2).
Take<Employee>(5);
```

The preceding code will skip the first two elements, but selects the next five elements in the list. The `Take` and `Skip` extension methods on `IEnumerable` is very useful.

4. The `TakeWhile` and `SkipWhile` methods helps in partitioning data based on some Lambda expression passed to them. For instance:

```
var filterEnumerable = (from emp in employees
                        select emp).SkipWhile<Employee>(
                        r => r.name.Length > 4);

 //Using Lambda Expression
var filterEnumerable = employees.Select<Employee, Employee>(
r => {
       return r;
     }).TakeWhile<Employee>(r => r.name.Length > 4);
```

The `TakeWhile` method is just the opposite to the `SkipWhile` method. The `Skipwhile` method skips the elements while evaluating the logic, while the `TakeWhile` method does the reverse.

5. Joining between two statements is another important aspect of LINQ. There is a special join key that allows to join two elements completely with support for `inner`, `right`, `left`, and `outer` joins. Let us look at the basic syntax of joining two collections.

```
var innerjoinedelements  = from emp in employees
                       join ord in orders
                       on emp.name
                       equals ord.empName
                       select emp;

 //Using Lambda Expression
var innerjoinedelements  = employees.Join<Employee, Order, string,
Employee>
                       (orders, e1 => e1.name, o => o.empName, (o,
e2) => o);
```

The preceding syntax joins the `Employee` with `orders` in such a way that both have the employee names. As we select only the common elements, this looks like one `innerjoin` statement.

```
var leftjoinedelement = from e in employees
join ord in orders on e.name equals ord.empName
into joinedorders from orders in joinedorders.DefaultIfEmpty()
select e;
```

The preceding statement represents the `leftjoin` statement. We need to use `DefaultIfEmpty` on the joined object to select the all the other unmatching elements from the other collection. Similarly, you can reverse the join statement to get the `rightjoin` query. You can make use of `DefaultIfEmpty` statement in your joins to generate outer joins too.

6. As we have all the features of query in LINQ, let's take a look how to write the `OrderBy` and `ThenBy` clauses.

```
var orderItems = from emp in employees
                 orderby emp.name, emp.age descending;

  //Using Lambda Expression
var orderItems =employees.OrderBy(i => i.name).ThenByDescending(i
=> i.age);
```

   Here the statement orders by employee name and then by age in descending order.

7. `Groupby` is another important criteria on LINQ. LINQ allows you to group objects to get aggregate of a collection. While using `GroupBy` we actually get an intermediate element which implements `IGrouping<tsource,telement>`. Let us look at the following code:

```
var itemNamesByCategory = from i in _itemList
                          group i by i.Category into g
                   select new { Category = g.Key, Items = g };
```

   This gets all the categories and items grouped by category. Well this grouping seems to be a little tricky for me. Let me help you understand what exactly is happening. Here while we are grouping, we are taking the group into `g` (which is a `IGrouping<tsource,telement>`). The `g` will have a key, which holds the grouped data; you can add multiple grouping statement. If you want to have a `having` clause, it will perform the same functions that the `where` clause does, as in the following example:

```
var filterEnumerable2 = from emp in employees
  where emp.age >65 //Normal Where clause works on all items
  group emp by emp.age into gr where gr.Key > 40
  select new {
            aaa = gr.Key,
            ccc=gr.Count<employee>(),
            ddd=gr.Sum<employee>(r=>r.age),
            bbb = gr
             };
```

Here in the example, I have returned the aggregate functions, such as `sum`, `count`, and so on which you can find from group data.

8. There are other operators like `Union`, `Intersect`, `Except`, and `Distinct` operators too. For instance:

   ```
   var un = (from i in _itemList
             select i.ItemName).Distinct()
             .Union((from o in _orderList
             select o.OrderName).Distinct());
   ```

   It gets distinct items from both the collections, but takes the union of both in the final resultant enumerable. You can use the `Intersect` operator in a similar way just like union to get the intersection of both the collection.

   ```
   var except = (from i in _itemList
                 select i.ItemID).Distinct()
                 .Except((from o in _orderList
                 select o.OrderID).Distinct())
   ```

   The preceding `Except` method takes the first collection and removes all the elements that appear in the second collection.

9. Another important consideration that you need to remember is the use of `let`. While between the query you can always create an intermediate storage to store some values during iteration process. We use the `let` keyword to define values inside the query. For instance:

   ```
   var filterEnumerable = from emp in employees
                          let x = emp.age
                          let y = x * 5
                          select x;
   ```

   In the preceding query, we have actually declared an intermediate variable to store a value. The compiler is smart enough to create an appropriate storage for you as `let` is self-evaluated during compilation just like `var`.

10. LINQ opens the whole gamut of traversal over programmable objects. With the introduction with LINQ few other subsidiary features were also introduced, which includes **DLINQ**, **XLINQ**, **LINQ to Entities**, **LINQ to NHibernate**, and **LINQ to anything**.

How it works...

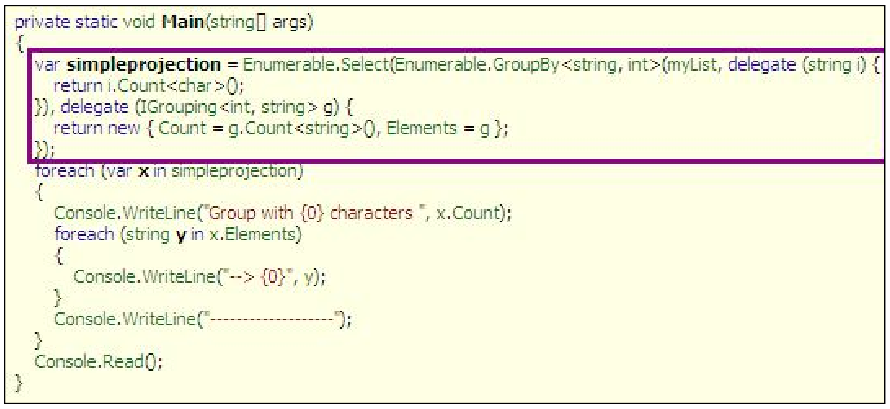
LINQ combines the anonymous methods and extension methods to create query among a type. Practically speaking, if you dive deep into LINQ and Lambda expressions, they are just syntactic sugar to the existing iterators. LINQ is a new way of writing a method that forms an iterator block.

When we say define a LINQ query, the C# compiler generates a number of types based on which it can store intermediate objects. For instance, even though to the end user we can use an anonymous type or an intermediate type using `let/join` or `group` keywords, they are actually under the hood concrete types that exist in the assembly. You should always remember that there is no concept of anonymous delegate or types in CLR, and hence whenever you define a LINQ statement that employs anonymous types, it creates a concrete type and embeds an object in the block that it is compiled to.

It is a combination of delegates and/or types that comprises the whole LINQ statement. If you look into some details of say a `Groupby` statement you will see something as follows:

```
var simpleGroup = from i in Program.myList
                        group i by i.Count() into g
                        select new { Count = g.Count(), Elements = g
};
```

Here we create an `IEnumerable` (as most of the LINQ statement returns an `IEnumerable`) of anonymous types with `g` being an intermediate object for which we generate `Count` and store the grouped data into the objects. If you see the same code in the reflector it would look as shown in the following screenshot:



```
private static void Main(string[] args)
{
  var simpleprojection = Enumerable.Select(Enumerable.GroupBy<string, int>(myList, delegate (string i) {
    return i.Count<char>();
  }), delegate (IGrouping<int, string> g) {
    return new { Count = g.Count<string>(), Elements = g };
  });
  foreach (var x in simpleprojection)
  {
    Console.WriteLine("Group with {0} characters ", x.Count);
    foreach (string y in x.Elements)
    {
      Console.WriteLine("--> {0}", y);
    }
    Console.WriteLine("------------------");
  }
  Console.Read();
}
```

So you can see here in the code that the C# compiler wraps around `Enumerable.GroupBy` inside the `Enumerable.Select`, where the former returns an `IEnumerable` of `IGrouping` members. On the other hand if you see it internally, the enumerable is first traversed to yield `IGrouping` objects. These objects are then iterated again using the iterator block to generate the actual anonymous objects. The `IGrouping` objects are capable to aggregate implementations. It is an interface which is actually implemented by the compiler automatically during compilation.

It is a similar technique just as how `join`, `orderby`, `distinct`, and so on, work internally.

There's more...

LINQ being a very vast topic, there is always an element to speak more about it. Let us look into some other aspects of LINQ that bear special attention.

How to work with LINQ to XML?

Parsing an XML and retrieving information about an XML is quite a big task with the XML DOM objects that existed before. Finding similar types within XML needed it to parse all the nodes and get the required one. But now with the introduction of LINQ, it should support querying of the XML elements and generate an XML document on the fly through code. LINQ to XML is a support of LINQ to the XML document. Before we go further, we need to consider a few types that form the XML body.

- `XDocument`: This determines the whole XML document
- `XElement`: This maps to an individual XML element
- `XDeclaration`: This forms the declaration of an XML
- `XAttribute`: This represents an attribute on an XML element
- `XComment`: This represents a commented out text on an XML
- `XName`: This represents any XML name that has been predefined

Now let us consider the following code:

```
XDocument bookStoreXml =
      new XDocument(
       new XDeclaration("1.0", "utf-8", "yes"),
       new XComment("Bookstore XML Example"),
       new XElement("bookstore", new XElement("genre",
         new XAttribute("name", "Fiction"),
         new XElement("book", new XAttribute("ISBN",
           "10-861003-324"), new XAttribute("Title", "A Tale of Two
Cities"),
          new XAttribute("Price", "19.99"),
          new XElement("chapter", "Abstract...", new XAttribute("num",
"1"),
           new XAttribute("name", "Introduction")),
          new XElement("chapter", "Abstract...", new XAttribute("num",
"2"),
           new XAttribute("name", "Body")
           )   )    )   )
      );
```

Here in the preceding code, you can see that we have built an entire `XDocument` object that represents the XML internally. Each object as stated in the preceding code represents its own properties. You can use the `bookStoreXml.Save` method to actually save the created XML to hard disk or other media.

Similarly, you can also use `XElement.Parse` or `XDocument.Parse` to parse an XML from a string, or `XDocument.Load` or `XElement.Load` to load from a disk file. Now what could you use to search in XML? Let us suppose that in the preceding code, we need to find all the books which have the genre `Fiction`. We write:

```
var element = from e in bookStoreXml.Descendants("genre")
                          let attr = e.Attribute("name")
                          where attr.Value.Equals("Fiction")
                          select e;
```

So here you can see the books will be selected for which the genre is `Fiction` and this `Fiction` is taken from an attribute of an element (name of the element). The `descendantNodes` will find all the nodes that have the element name that we passed.

You can try out more like this to understand this further.

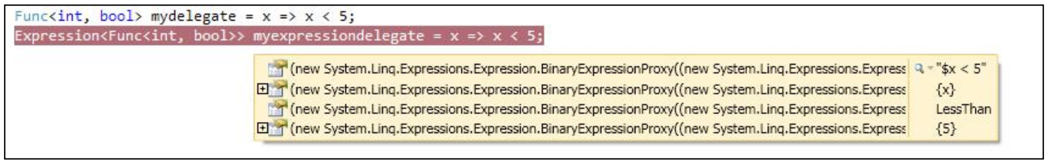
What are expression trees and how can we decompose a Lambda expression into an expression tree?

An expression tree is nothing but the representation of a Lambda expression in terms of .NET objects. `Expression` is the main class that holds any expression in it and it lets us divide the whole body of `Expression` into smaller individual units. For instance:

```
Func<int, bool> mydelegate = x => x < 5;
Expression<Func<int, bool>> myexpressiondelegate = x => x < 5;
```

In the preceding code, the delegate `Func` can take the expression `x=>x<5` which represents an anonymous method that takes an integer argument and returns if the value is less than five.

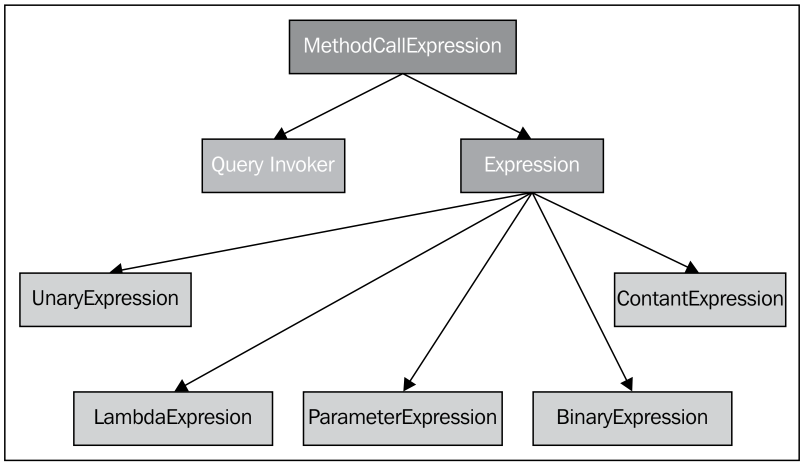
Now to decompose the method into an expression tree, let's wrap the `Func` into an `Expression` body. `Expression` has overloaded the assignment operator to decompose the body of the function.



```
Func<int, bool> mydelegate = x => x < 5;
Expression<Func<int, bool>> myexpressiondelegate = x => x < 5;
```

```
(new System.Linq.Expressions.Expression.BinaryExpressionProxy((new System.Linq.Expressions.Express   "$x < 5"
(new System.Linq.Expressions.Expression.BinaryExpressionProxy((new System.Linq.Expressions.Express   {x}
(new System.Linq.Expressions.Expression.BinaryExpressionProxy((new System.Linq.Expressions.Express   LessThan
(new System.Linq.Expressions.Expression.BinaryExpressionProxy((new System.Linq.Expressions.Express   {5}
```

Now if you see the body of Expression, you can see that there are three parts in the whole expression:



- **ParameterExpression**: This is an external parameter to the expression. Here it is `x`.
- **BinaryExpression**: As the inner expression `x<5` is binary, it produces a BinaryExpression. Each BinaryExpression has two `Expressions` body within it and two properties Left and Right. In our case, the Expression has one ParameterExpression and another ConstantExpression.
- **Left**: This produces the left-hand side of the BinaryExpression. In our case the left-hand side represents the ParameterExpression the same object which is passed as `x`.
- **Right**: Right represents the other side of the expression which in our case is a constant term. So Right represents a ConstantExpression for us.
- **NodeType**: The NodeType gives you an idea of what the BinaryExpression does. There are a lot of binary types available. In our case it is `LessThan`.

Hence the entire decomposition will look as follows:

```
ParameterExpression externalParam = myexpressiondelegate.
Parameters[0];
BinaryExpression bbody = myexpressiondelegate.Body as
BinaryExpression;
ParameterExpression parambodyleft = bbody.Left as ParameterExpression;
ConstantExpression constRight = bbody.Right as ConstantExpression;
ExpressionType type = bbody.NodeType;
```

Each `Expression` has a `ReadonlyCollection` of parameters that are passed within the body of the `Expression`. In our case the `Expression` has one parameter. As you can see I have used `Parameters[0]` to get the parameter that is passed to the `Expression`. The external parameter represents `x` here and its Type is `Int32`.

The `Expression` has a body element which shows you the method body of the expression. In our case the body will hold only the part `x<5`. As it is a **BinaryExpression**, I can easily translate it into a reference of BinaryExpression. So in our case the `body` represents the BinaryExpression.

Finally we parsed the **BinaryExpression** to get the parameter x in the Left and **ConstantExpression** five in the Right property.

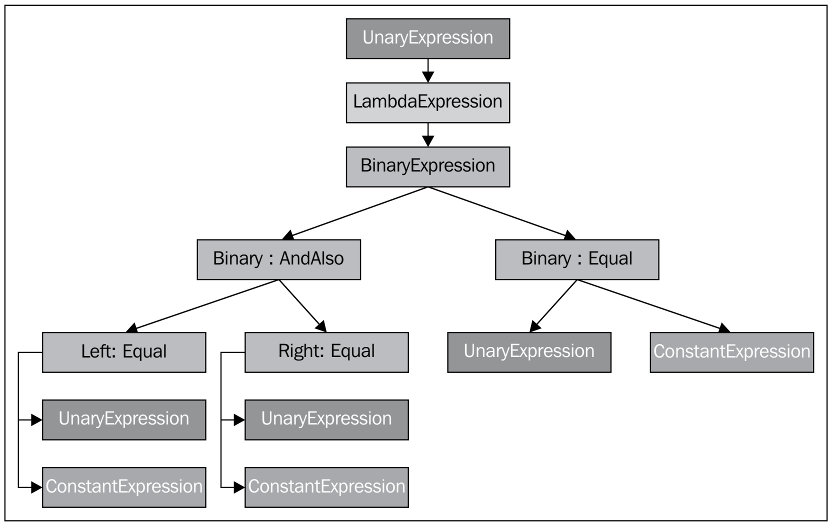
Now if you want to recreate the whole expression from these objects, you can do so using :

```
BlockExpression body = Expression.Block(
Expression.LessThan(parambodyleft, constRight));

Expression<Func<int, bool>> returnexpression = Expression.
Lambda<Func<int, bool>>(body, param1);
Func<int, bool> returnFunc = returnexpression.Compile();

bool returnvalue = returnFunc(30);
```

`BlockExpression` represents the actual body of the `Expression`. Based on the complexity of the code you need to define the Expression block.

Finally the call to `Compile()` generates the IL code dynamically at runtime and basically gives you the example of compiler as service.

Now if you run the code, you will find the `returnvalue` holds *false* for *30* and *true* for *3*.



Expression trees are created based on a number of objects. When you define an `Expression`, you will have internally created a number of `Expression` objects, each of which bears a special meaning to the compiler.

The entire expression is built using the building blocks, such as **BinaryExpression** based on its Type, contains Left and Right properties, which are individual `Expression` attributes themselves. Any logical Expression contains one **UnaryExpression**, a ConstantTerm, and so on.

Why IQueryable exists?

`IQueryable` is a special interface which is used whenever we need an expression to be stored, and at runtime need to be compiled to produce a result. With the use of Expression trees you can store the logic of a method as data and evaluate the same by building the function dynamically on the fly. If you look closely into the `IQueryable` interface or if you look into any of the extension methods `Where`, `Select`, and so on, they are basically storing the anonymous methods into an `Expression` that eventually loads the whole expression tree within it. So any minor change to the Lambda expression will not eventually create a completely new anonymous method, rather it will add up to the expression tree and will be compiled dynamically on demand when the actual evaluation of the statement occurs.

Let's look at the structure of `IQueryable`:

```
public interface IQueryable : IEnumerable
{
    Type ElementType { get; }
    Expression Expression { get; }
    IQueryProvider Provider { get; }
}
```

You can see that `IQueryable` holds an `Expression`. So when an expression is passed to an `IQueryable`, it just holds it in a complete expression tree which it will evaluate using `IQueryProvider` at runtime. `IQueryProvider` actually calls `CreateQuery` to get the `QueryEvaluation` from the `Expression`, and which will eventually be evaluated during runtime.

See also

- `http://bit.ly/Linq-basics`
- `http://bit.ly/LINQ-Internals`

Using dynamic objects in .NET

C# being a strongly typed language, you might wonder how you could make it dynamic. In C# everything that we define will have a strong signature. There are numerous languages available, such as Python, Ruby, and most notably JavaScript which are not strongly typed, and you can change the object anytime whenever you require. C# 3.5 or below doesn't support changing the type dynamically during runtime. But as a developer we generally miss this flexibility of the language. But seriously that shouldn't change the class of the language like what C# is. I mean, by adding up features like dynamic language we shouldn't compromise with C# as a strongly typed language.

Microsoft introduced **dynamic** in C# with C# 4.0. But believe me, C# is not yet a truly a dynamic language. Let us talk about dynamic keywords a bit for the time being.

Getting ready

When I first came to know about dynamic in C# 4.0, I wondered about its requirements. We have an object that can hold any number of objects in it, as it is called as the mother of all objects. Objects can hold anonymous types or even call properties or methods of an anonymous type easily. So what exactly do we require for dynamic? To demonstrate the situation let us take the example of how you can call properties or methods of an object for which you don't know the exact type.

Let us have a class `Person` which is as follows:

```
internal class Person
{
     public string Name { get; set; }
     public int Age { get; set; }
     public double Weight { get; set; }
     public string GreetMe()
     {
         return string.Format("Hello {0}", this.Name);
     }
}
```

Now let our method `GetObject` return an object of the `Person` class as object. So to call properties or methods we would write:

```
    public void ResolveStatic(object obj)
    {
          PropertyInfo pinfo = obj.GetType().GetProperty("Name");

          //To Get from a property Name
          object value = pinfo.GetValue(obj, null);
          string content = value as string;
    Console.WriteLine(content);

           // To Set to Property Name
          pinfo.SetValue(obj, "Abhijit", null);

          //Invoke a method
          MethodInfo minfo = obj.GetType().GetMethod("GreetMe");
          string retMessage = minfo.Invoke(obj, null) as string;

          Console.WriteLine(retMessage);
    }
```

This is really what you expect to write. Does it look complex to you? Yes, it is. Probably if you have more flexibility like static method, or multiple arguments for a method, or even an indexer, your code will look horrendous.

The **dynamic** keyword helps you in this regard. Yes, you can simply call methods and properties in your code without writing the whole reflection methods, and the compiler will do them for you. So doing the same thing with dynamic, the code will look as follows:

```
    public void ResolveDynamic(dynamic obj)
    {
          Console.WriteLine(obj.Name);
          obj.Name = "Abhijit";

          Console.WriteLine(obj.GreetMe());
    }
```

So the code looks the simplest and the same as known types. Actually if you see through the reflector, you will see that the same code is created by the compiler for us. So, the dynamic keyword makes our code look simple for dynamic types and also makes the code easy to understand.

Let us discuss the true feature of dynamic in detail.

How to do it...

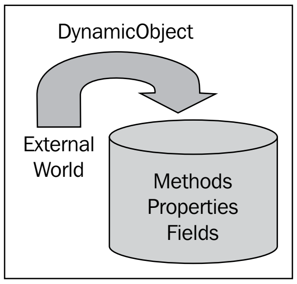
1. Create an object of `ExpandoObject` that exists in .NET 4.0 as a Type.
2. Start typing properties/methods on an object that are assigned in the `dynamic` variable.
3. Call them on the fly, compile it, and run it.
4. When the code gets compiled, you get the output exactly in such a way that the object actually has those properties and methods inside it. Let us take the following example:

   ```
   dynamic d = new ExpandoObject();
   d.Name = "Abhishek Sur";
   d.Age = 26;
   d.Weight = 62.5d;
   d.GreetMe = new Action(delegate()
   {
         Console.WriteLine("Hello {0}", d.Name);
   });
   ```

   Here when we execute the code above the object, `d` can assign the name into its property, the age, weight, and can also call the method `GreetMe`.

How it works...

`ExpandoObject` is a new class added to enhance the dynamic capabilities of C# 4.0. `ExpandoObject` lets you add methods or members to an object dynamically during runtime. If you are wondering how it is possible, or the depth behind creating the `ExpandoObject` class, let's read further to create your own DynamicObject.



To understand how a statically typed language can extend properties and class in runtime, or how this could be made possible, you need to delve deep into the internal structure of `ExpandoObject`.

Actually `ExpandoObject` is a Collection of `KeyValuePair` where the key represents the interface through which the objects are called for, and the value is the object which might be a string, an object, or a method depending on what you pass. When we pass the methods or properties to `ExpandoObject`, it actually adds those as a `KeyValuePair` into its collection, and when the object is invoked it calls up the method or property dynamically.

Now to create your own custom `ExpandoObject`, you need to inherit your class from `IDynamicMetaObjectProvider`. This interface has a method called `GetMetaObject`, which you need to implement to create `DynamicMetaObject` from the LINQ expression. At the first instance, it seemed to me as a trivia, but it is not. You actually need to parse the complex LINQ expression to create your meta object. I must thank Microsoft for giving an implementation of `IDynamicMetaObjectProvider` as `DynamicObject`. So for simplicity, you can inherit your class from `DynamicObject` and get your job done.

Let us implement our class from `DynamicObject`:

```
public class CustomExpando : DynamicObject
 {
   public IDictionary<string, object> Dictionary { get; set; }
   public CustomExpando()
   {
     this.Dictionary = new Dictionary<string, object>();
   }

   public int Count { get { return this.Dictionary.Keys.Count; } }
   public override bool TryGetMember(GetMemberBinder binder, out
object result)
  {
     if (this.Dictionary.ContainsKey(binder.Name))
     {
         result = this.Dictionary[binder.Name];
         return true;
     }
     return base.TryGetMember(binder, out result); //means result =
null and return = false
        }

 public override bool TrySetMember(SetMemberBinder binder, object
value)
        {
            if (!this.Dictionary.ContainsKey(binder.Name))
            {
                this.Dictionary.Add(binder.Name, value);
            }
            else
                this.Dictionary[binder.Name] = value;
            return true;
        }

     public override bool TryInvokeMember(InvokeMemberBinder binder,
    object[] args, out object result)
        {
            if (this.Dictionary.ContainsKey(binder.Name) && this.
    Dictionary[binder.Name] is Delegate)
            {
                Delegate del = this.Dictionary[binder.Name] as
    Delegate;
                result = del.DynamicInvoke(args);
                return true;
            }
            return base.TryInvokeMember(binder, args, out result);
        }

     public override bool TryDeleteMember(DeleteMemberBinder binder)
        {
            if (this.Dictionary.ContainsKey(binder.Name))
            {
                this.Dictionary.Remove(binder.Name);
                return true;
            }

            return base.TryDeleteMember(binder);
        }

      public override IEnumerable<string> GetDynamicMemberNames()
        {
            foreach (string name in this.Dictionary.Keys)
                yield return name;
        }
    }
```

Here we have overridden methods:

- `TryGetMember`: This is called when the `Get` method of a property is called. If returned true, the value in the result will be returned.
- `TrySetMember`: This is called when the Set method of a property is called. If returned true, the value in the result will be set to the member.
- `TryInvokeMember`: This is called when a delegate/method is called. The result will be returned.

We use a dictionary to store the key value collection of the methods and properties. As you can see that `TryGetMember/TrySetMember` and `TryInvokeMember` will be called automatically during runtime, if we can handle these methods properly, we can execute some code to that expanded object.

Unless others are required, you can use these members only to have an `ExpandoObject`. To use this class we use the following:

```
dynamic expandoObject = new CustomExpando();
expandoObject.Name = "Akash";
expandoObject.CallMe = new Func<string, string>(delegate(string name)
{
       expandoObject.Name = name;
      return expandoObject.Name;
});

Console.WriteLine(expandoObject.Name);
Console.WriteLine(expandoObject.CallMe("Hello"));
```

The call to the method will be translated to the collection and the call from the object will call the interface members accordingly. As you have seen, we have actually invoked the appropriate property from within the class, `Console.WriteLines` will be successful. You can remove a member from the list using the following:

```
expandoObject.Dictionary.Remove("Name");
```

This is it. Now you can handle your `ExpandoObject` yourself.

There's more...

Dynamic features of .NET are greatly enhanced by many other important facts.

Difference between var and dynamic

Even though both look the same to a normal developer, they are widely different in terms of actual implementation. `var` is a static type that is inferred during compile time. It depends on the object that has been assigned to it, which infers its type during compile time. Thus, when we assign string to a `var`, it automatically starts behaving like a string variable. `var` just allows the programmer the flexibility to omit the left-hand side initialization of type. `var` is entirely statically defined and if you look at the actual compiled code in the assembly, you will see the actual replacement of the assigned type to its assignee.

Dynamic means that any expression that is going to be executed is bound at execution time, rather than compile time, so it behaves dynamically. The compiler is not going to check a dynamic object for its methods and its bindings. The behavior of the dynamic object is determined at execution time.

The object that is actually declared as dynamic internally implements `IDynamicMetaObjectProvider`. This interface helps in determining and responding to the actual method call during runtime. The call states and binders don't exist at all.

Dynamic and COM Interop

Dynamic feature of the language actually helps a lot with COM Interop services. The C# team specifically dealt with many features and implemented the dynamic part of it inside Office products like Word, Excel, and so on. When working with the Interop services with Excel, the objects are replaced dynamically to make it easier to work with and avoid casting. Let us look at the following code:

```
var excel = new Excel.Application();
excel.Workbook.Add();
excel.Visible = true;
Excel.Range targetRange = excelApp.Range["A1"];
targetRange.Value = "Abhishek";
targetRange.Columns[1].AutoFit();
```

Here the code looks very simple because all the columns are made dynamic instead of previously where we needed to cast the object into appropriate types. The last line in the previous version before dynamic was introduced needs to be written as:

```
(Excel.Range)targetRange.Columns[1, Type.Missing].AutoFit();
```

We needed the cast before as columns returned an object to which we cannot call AutoFit.

See also

- `http://bit.ly/Dynamic-Types`

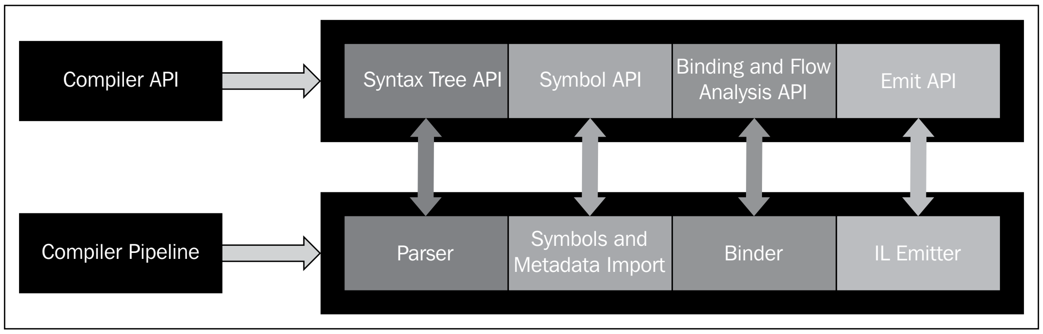
Compiler as service

With the introduction of newer technologies and enhancements in the languages under .NET, the applications are getting more and more powerful to add things that were never easy before. .NET languages are continuously evolving with newer types and technologies that make it much more advanced and useful in most scenarios. Compiler as service is a new concept that has been released very recently with .NET, and is one of the major enhancements to it.

Most of the languages that we regularly deal with are often thought of as black boxes that take source code as input and produce binaries as output. The internal logic on how the entire process of compilation works is still not exposed to the application programmers. Compiler as a service is a new enhancement made to the compiler which allows the application programmer to query the compiler to get information at various stages of compilation. In this recipe I am going to cover a basic introduction to compiler as service and get you through the capabilities that it posses.

Getting ready

To start with this recipe, let us consider the things that a compiler does after we kick off the build process. The first thing that the compiler does is parse the code that we pass to the compiler. There are specific parsers available for each compiler that parse and understands the code. The next thing it does is that it identifies symbols and metadata to import external files or assemblies. The most important process is done by the Binder, which binds symbols with actual calls. And finally it emits the IL into an assembly file. `Roslyn project`, or more specifically the compiler as a service, exposes these features of the compiler as a service and give you an API to actually get information out of these compilation tasks.



In the preceding figure, if you look at the bottom of the image, it tells you about the Compiler Pipeline block that I just stated. In this section, the Parser block is available through Syntax Tree API, the Symbol API exposes the symbols, Binding and FlowAnalysis API exposes the Binder, and finally Emit API exposes the IL Emitter.

How to do it...

In this module, we are going to brief on each of these sections one-by-one and also look at how to get benefit out of these.

1. Start a Windows Application, and paste a textbox and a button. Say the textbox represents the source code and the button represents the command to parse the text.

```
SyntaxTree tree = SyntaxTree.ParseCompilationUnit(this.textBox1.
Text);
AssemblyFileReference mscorlib = new AssemblyFileReference(typeof(
object).Assembly.Location);
 Compilation compilation = Compilation.Create("Random");
compilation.AddReferences(mscorlib);
compilation.AddSyntaxTrees(tree);
```

The preceding code represents the full syntax tree of the compilation. The first line takes the whole program that needs to build a compilation unit. `ParseCompilationUnit` parses the source code into a `SyntaxTree`. By the way, we added the `mscorlib.dll` to the compilation and created a compilation.

2. We add a `treeView` and a `PropertyGrid` to the form, and on `TreeView.NodeMouseClick` we select the object in the `PropertyGrid`.
3. The root of the compilation unit is determined by using the `GetRoot` method of the `CompilationUnit`.
4. From the root, we can parse collections like `Usings` to get the information about the Using added to the compilation unit or members to get types.

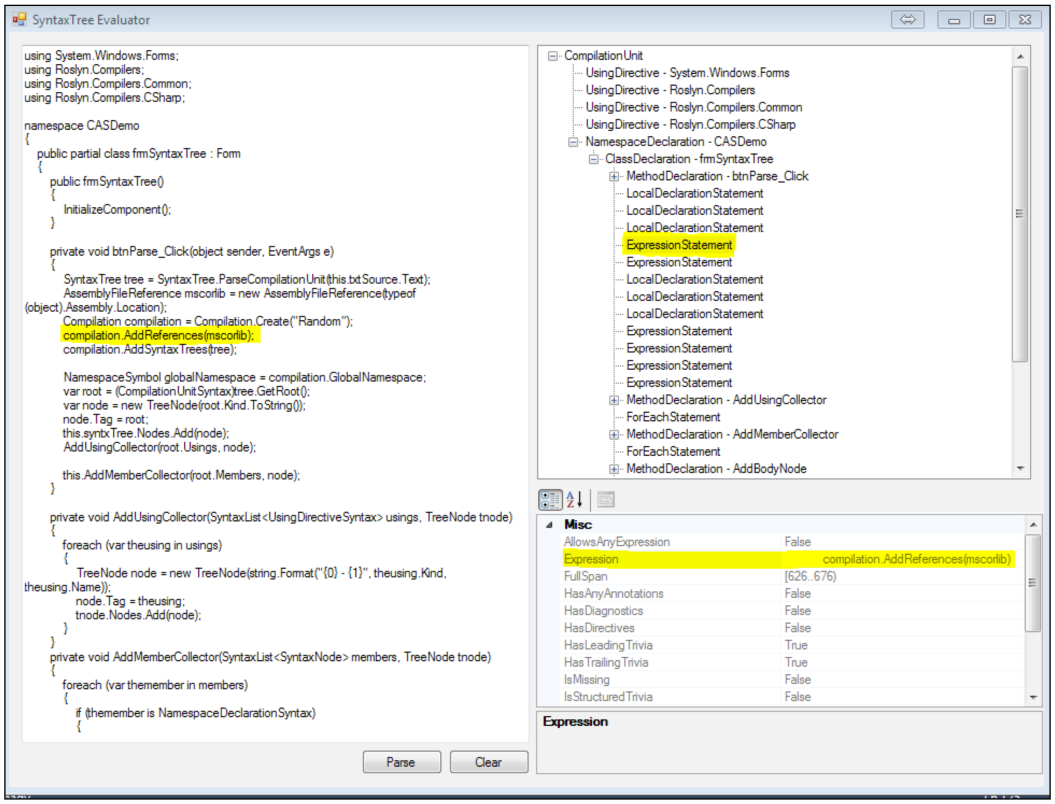
```
var root = (CompilationUnitSyntax)tree.GetRoot();

foreach (var theusing in root.Usings)

{

     TreeNode node = new TreeNode(string.Format("{0} - {1}",
theusing.Kind, theusing.Name));

     node.Tag = theusing;

     tnode.Nodes.Add(node);

}
```

In the preceding code, we got the `CompilationUnitSyntax` from the root and added the `Usings` associated with the code into the `TreeView`. It is important to note that we added the actual object to the tag of each node, so that we can select the object from its tag to the `PropertyGrid`. The `Usings` has a kind and a name, which determines the `SyntaxType` and the text of Using respectively. The actual Type of the `usings` is `UsingDirectiveSyntax`.

5. Similarly, we can also use `root.Members` to get the collection of `MemberDeclarationSyntax`. This member is possibly a collection of `NamespaceDeclarationSyntax` or a `TypeDeclarationSyntax`. Each of them has its own members. In the case of `NamespaceDeclarationSyntax` it holds a collection of `TypeDeclarationSyntax` and the `TypeDeclarationSyntax` holds a collection of `MethodDeclarationSyntax`.
6. We add each element to the tree to get the complete syntax tree hierarchy of the code.

How it works...

The following screenshot shows the entire hierarchy of the program:



```
using System.Windows.Forms;
using Roslyn.Compilers;
using Roslyn.Compilers.Common;
using Roslyn.Compilers.CSharp;

namespace CASDemo
{
   public partial class frmSyntaxTree : Form
   {
      public frmSyntaxTree()
      {
         InitializeComponent();
      }

      private void btnParse_Click(object sender, EventArgs e)
      {
         SyntaxTree tree = SyntaxTree.ParseCompilationUnit(this.txtSource.Text);
         AssemblyFileReference mscorlib = new AssemblyFileReference(typeof
(object).Assembly.Location);
         Compilation compilation = Compilation.Create("Random");
         compilation.AddReferences(mscorlib);
         compilation.AddSyntaxTrees(tree);

         NamespaceSymbol globalNamespace = compilation.GlobalNamespace;
         var root = (CompilationUnitSyntax)tree.GetRoot();
         var node = new TreeNode(root.Kind.ToString());
         node.Tag = root;
         this.syntxTree.Nodes.Add(node);
         AddUsingCollector(root.Usings, node);

         this.AddMemberCollector(root.Members, node);
      }

      private void AddUsingCollector(SyntaxList<UsingDirectiveSyntax> usings, TreeNode tnode)
      {
         foreach (var theusing in usings)
         {
            TreeNode node = new TreeNode(string.Format("{0} - {1}", theusing.Kind,
theusing.Name));
            node.Tag = theusing;
            tnode.Nodes.Add(node);
         }
      }
      private void AddMemberCollector(SyntaxList<SyntaxNode> members, TreeNode tnode)
      {
         foreach (var themember in members)
         {
            if (themember is NamespaceDeclarationSyntax)
            {
```

```
CompilationUnit
   UsingDirective - System.Windows.Forms
   UsingDirective - Roslyn.Compilers
   UsingDirective - Roslyn.Compilers.Common
   UsingDirective - Roslyn.Compilers.CSharp
   NamespaceDeclaration - CASDemo
      ClassDeclaration - frmSyntaxTree
         MethodDeclaration - btnParse_Click
         LocalDeclarationStatement
         LocalDeclarationStatement
         LocalDeclarationStatement
         ExpressionStatement
         ExpressionStatement
         LocalDeclarationStatement
         LocalDeclarationStatement
         LocalDeclarationStatement
         ExpressionStatement
         ExpressionStatement
         ExpressionStatement
         ExpressionStatement
         MethodDeclaration - AddUsingCollector
         ForEachStatement
         MethodDeclaration - AddMemberCollector
         ForEachStatement
         MethodDeclaration - AddBodyNode
```

Misc

AllowsAnyExpression	False
Expression	compilation.AddReferences(mscorlib)
FullSpan	[626..676)
HasAnyAnnotations	False
HasDiagnostics	False
HasDirectives	False
HasLeadingTrivia	True
HasTrailingTrivia	True
IsMissing	False
IsStructuredTrivia	False

Expression

If you navigate over the syntax tree, you can determine each and every individual statement for the whole compilation unit. When you create a `compilationUnit` the `SyntaxTree` parser creates individual objects that resemble the whole program. Each and everything from the start of the program to the end of the program, tokens, expressions, or return values, are tracked within its syntax tree. The `SyntaxTree` object has a universal property called `Kind` which determines the exact type of the syntax. For instance, the `ExpressionStatement` is a value of the enumeration `SyntaxKind`. The `SyntaxKind` holds virtually every kind of syntaxes that the C# compiler supports.

You can create a query over the syntax tree after you built the `compilationUnit`. Remember, building a `CompilationUnit` does not mean building an assembly. It produces the objects on the fly in memory, so it is safe to build more than one `compilationUnit` for comparison.

To query in a syntax tree, we can define a `semanticModel` out of the tree. Let us suppose we want to query the symbol associated with the `First` method declared in the compilation tree:

```
SemanticModel semanticModel = compilation.GetSemanticModel(tree);
MethodDeclarationSyntax methodDecl = tree.GetRoot()
            .DescendantNodes()
            .OfType<MethodDeclarationSyntax>()
            .First();
Symbol methodSymbol = semanticModel.GetDeclaredSymbol(methodDecl);
```

The query gets the symbol that resembles the `First` method on the type. As you can see, the `DescendantNodes` returns all the nodes on the `SyntaxTree` from the root, we can easily get the exact `SyntaxElement` from the tree using LINQ.

```
InvocationExpressionSyntax invocation = methodDecl.DescendantNodes()
              .OfType<InvocationExpressionSyntax>()
              .First();
SymbolInfo info = semanticModel.GetSymbolInfo(invocation);
Symbol invokedSymbol = info.Symbol;
```

For instance in the preceding code, we select the `First` expression from the method. The `Symbol` result represents the whole information about the expression.

Similar to the one shown in the preceding code, we can query other types using appropriate symbols in the code.

There's more...

By the way, compiler as a service exposes lot of additional APIs in addition to Syntax or Semantic API. Let's discuss some more interesting things that you can do with Roslyn.

How to add C# scripting Support?

Roslyn APIs allow you to expose C# scripting APIs and compile expressions dynamically at runtime. You can define and interact with the program directly while you are running your code. You can use the C# interactive window to actually get a flavor of the same from the IDE.

To add scripting support on an application we need to:

1. Use Roslyn scripting APIs to create C# script engine
2. Create a session that holds all the execution context of the Interactive window
3. Execute each C# code supplied in the host specified scope

4. Push function definitions as command implementations and add to the command collection
5. Add appropriate references beforehand to execute library functions

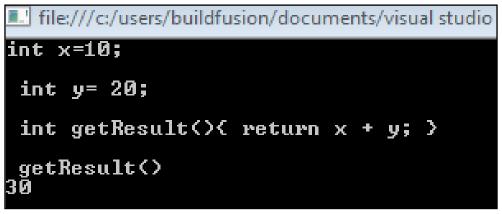
Let us consider the implementation of the most basic interactive host:

```
public class InteractiveHost
{
    ScriptEngine engine = null;
    Session session = null;

    public InteractiveHost()
    {
        engine = new ScriptEngine(new[] {"System", this.GetType().
Assembly.Location});
        session = Session.Create();
    }

    public object ExecuteCode(string code)
    {
        return this.engine.Execute(code, this.session);
    }
}
```

The first thing that I did here was create `ScriptEngine`. The `ScriptEngine` actually creates an engine where the C# code will be executed. We have passed a few namespaces that we need to add in to start with. One of them is `System` and also added the reference of this assembly. The other thing that you need to notice is the `Session`. When we create a new `Session`, all the existing execution is made under that `Session`. So when we execute an expression within a `Session`, all the previous executions will be available to it.

```
InteractiveHost host = new InteractiveHost();

bool isExecute = true;
while (isExecute)
{
    string command = Console.ReadLine();
    Console.WriteLine(host.ExecuteCode(command));

    var rkey = Console.ReadKey();
    isExecute = rkey.Key != ConsoleKey.Escape;
}
```

If I run this code, I will get the following output:



```
int x=10;

 int y= 20;

 int getResult(){ return x + y; }

 getResult()
30
```

Now from the Console if I execute commands like the one shown in the preceding screenshot, we could create integer variables `x` and `y` directly from the engine, and then create a method `getResult` which can again be executed on the fly to get the sum of `x` and `y` (*30*). This is the simplest implementation of the Interactive window.