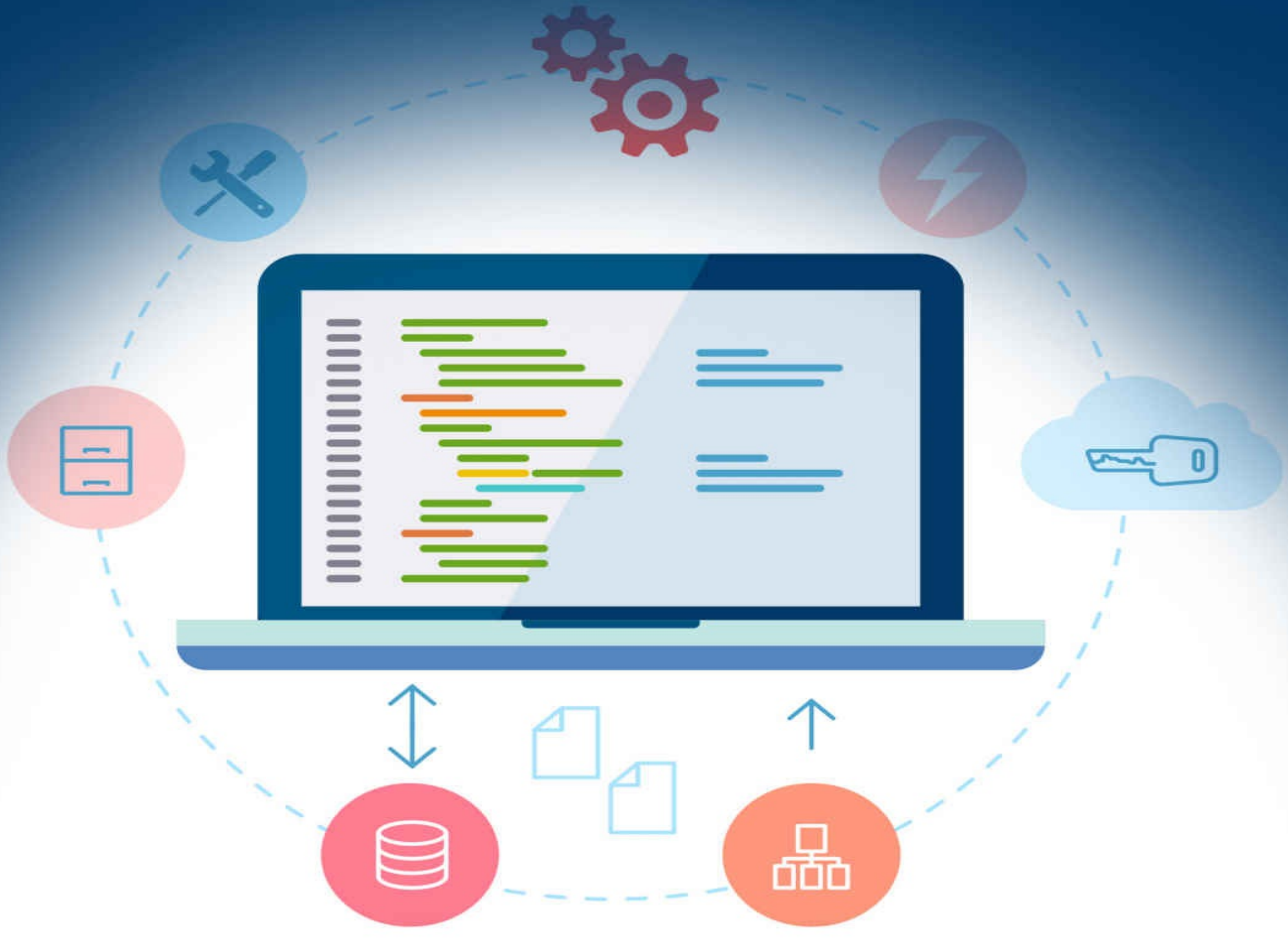
INTERMEDIATE C# PROGRAMMING



TROY DIMES

Intermediate C# Programming

Troy Dimes

Contents

Introduction

Welcome to the Intermediate C# Programming. In this book you will learn intermediate and advance C# programming. If you are reading this book, you should already have some basic familiarity with C# programming including its data types, operators, classes, modifiers, and events. If not, pick up a copy *of C# Programming for Beginners* by visiting:
http://www.deepthoughtpress.com/c-sharp

In the next section, we'll start covering some advanced C# topics. Starting with the all important exception handling, this section will take you through Generics, Expression trees, and will end with LINQ (Language Integrated Query) after covering Null-able and Anonymous types.

Each chapter starts with a thorough summary of the topic and introduces the contents to be covered in the chapter. Sample code snippets with text and examples at end of each chapter (in the form of an exercise) are provided to ensure that you grasp each topic easily and comprehensively.

Chapter 1: Exception Handling

Exceptions are run time errors. That is, they arise during program execution, and are mostly an outcome of poor programming logic, i.e. division by zero, but not always.

While it is easy to detect and handle syntax or compile time errors in your code, since most compilers do that for you, logical errors (or exceptions) may or may not always be easy to foresee and manage. For example, in a simple program that takes user input, if the input type (as specified by programmer) is “int” but a user enters “string,” an exception will occur, which if not already handled might result in an abrupt closure/stoppage of the application.

Exceptions like one above can be thought of beforehand, and are manageable to a large extent. However consider a case where you receive a memory buffer overflow, I/O file error, or database error. Errors like these and many others cannot always be predicted and may result in a system throwing exceptions.

In this chapter, you will see how you can manage exceptions occurring in your applications using the C# exception handling mechanism. This involves:

Contents

But before we see how we can manage exceptions occurring in our code, it’s important that we know a few basic things about them. For instance, Exceptions are thrown by the “System.Exception” class and they are actually good in the sense that they save system failures. Hence, to write efficient code, it’s imperative that we care for and handle exceptions that may occur in our programs ourselves. Let’s now see how.

1. Try/Catch Construct

Working in tandem, a try/catch construct is mostly used to handle exceptions that may occur during run time. Putting our code in a “try { }” block saves our programs from abrupt closure. Inside the try block, if an exception occurs, the system instead of halting program execution, stops at the line where the error has occurred, and looks for a related “catch” block to transfer control by throwing occurred exceptions there. The catch block, as the name indicates, catches that exception, and does as per program instructions in that block.

The Basic Skelton of a try/catch block inside any program is:

```
try
{
   //Program Code that might raise an exception
}
catch
{
   //Exception gets handled inside
}
```

Let’s see an example to better understand how it works.

Example 1:

```
using System;
using System.Collections;

namespace SampleApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         int[] num = new int[2];

         num[0] = 13;
        num[1] = 22;
         num[2] = 42;
         foreach(int x in num)
       Console.WriteLine(x);
         Console.ReadLine();
      }
   }
}
```

If we try to execute the above code, we will receive an “Array out of Bound” exception, since our array size is 2 here while we are trying putting 3 objects in our array. But placing the above code in a try/catch block will result in a different outcome.

Example 2:

```
Int [] num = new int[2];
try
{
   num[0] = 13;
   num[1] = 22;
```

```
    num[2] = 42;

    foreach(int x in num)
        Console.WriteLine(x);
}
catch (Exception e)
{
Console.WriteLine (‘Error occurred: ‘ + e.Message);
}
Console.ReadLine();
        }
    }
}
```

Now, when we execute our code, the try {} block saves our program from throwing exceptions towards us, and instead passes it to our catch block. The catch block receives that exception in object “e” and simply displays the exception message on screen for us.

2. Finally keyword:

The code inside a “finally” block guarantees essential execution, irrespective of whether an exception occurs or not, or what type of exception occurs. Mostly, programmers use this block to print “ending” messages before closure in games or to close file references and garbage objects, as those are no longer required.

Another important thing to note here is that we can use multiple catch segments with the same try block. That is, each catch block can be used as a dedicated basket to catch only a unique type of exception and not all. This is quite useful if we want our program to do different things on each type of exception. To do that on the same code, we get:

Example 3:

```
public void F1()
    { Console.WriteLine(‘Control inside f1’ );}
public void F2()
    { Console.WriteLine(‘Control inside f2’ );}

Int [] num = new int[2];
try
{
    num[0] = 13;
    num[1] = 22;
    num[2] = 42;

    foreach(int x in num)
        Console.WriteLine(x);
```

```
}
catch(IndexOutOfRangeException e)
{
Console.WriteLine('Index out of range found!');
F1();
}
catch(Exception e)
{
Console.WriteLine('Some sort of error occured: ' + e.GetType().ToString();
F2();
}
finally
{
Console.WriteLine('It's the end of our try code block. Time to Sleep!');
}
```

Now, in the code above, if an "array out of bound" exception occurs during execution, the control or the exception is received by our first catch block that prints "Index out of range found!" and then calls F1(). However, if another exception occurs, e.g. buffer overflow for instance, the control will pass to our second catch block that will print out exception type by using the "e.GetType().ToString()" function and will call F2(). In both cases, the control will eventually be passed on to the "finally" segment.

3. Throw Keyword

If we are using the "throw" keyword in our programs, that only means that we are creating and throwing custom exceptions. The idea of creating and throwing exceptions ourselves sounds absurd at first, but it's actually more than handy to begin with. For instance, it's handy when you would like to transfer control if your program fails to open a target file, or is waiting for too long for an I/O resource, etc.

Example 4:

```
static void Main()
{
Try{
//your code
   Exception ex = new Exception ('The file was found missing');
   throw e;
}}
catch (Exception e)
{ Console.WriteLine  (e.Message);}
}
```

Here for instance, if our code is looking for a file that it couldn't find it will create an exception object and will pass the control to a catch block using the throw keyword. We can then print out the exception on the console or do whatever our program requires.

Exercise 1

Task:

Create a custom exception if a user tries to divide a number by zero.

Solution

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
namespace SampleApplication
{

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

{
try
{
                int a = 0;
                int b = 10;
                int c = 0;
                c = b / a;
            }
            catch (Exception ex)
            {
                throw (new MyCustomException('You cannot divide a number by 0'));
            }
        }
    }
    public class MyCustomException : System.Exception
    {
            public MyCustomException() : base() {}
            public MyCustomException(string message): base(message)
            {
                MessageBox.Show(message);
            }
    }
}
```

Chapter 2: Lambda Expression and Expression Trees

In this chapter, we are going to study two new related features in the newest version of C# 3.0 and the .NET 3.0 runtime: Lambda expressions and Expression trees. We will learn how to create and use them to enhance and simplify our C# code. The knowledge behind the concept of delegates which we have already studied in *C# Programming for Beginners* will be useful for the understanding of this chapter.

Contents

- **Lambda Expressions.**
- **Expression Trees.**

1. Lambda Expressions

A Lambda Expression is an anonymous function that is used to create delegates or expression tree types. Lambda Expressions give us a fast and easy way to define delegates. A Lambda Expression is like a method without a declaration, i.e. access modifiers. The usage of Lambda Expressions can save you two or three lines of code per call. It speeds up your application development and makes the code maintainable and reusable.

It also allows us to write our methods exactly at the same place where we are going to use them. Lambda Expressions are similar to anonymous methods - just slightly smarter in syntax, but both compile to the same intermediate language.

Lambdas are also used in method-based LINQ queries (as you will see in the chapter on LINQ) as arguments to standard query operator methods such as “Where.”

Lambda Expressions follow the basic signature:

Parameters => Executed code

Example 1:

```
y =>            y*y;
```

In the above example, “y” is the input parameter, and “y*y” is the expression. The Lambda Expression specifies the parameter named ‘”y” and it returns the value of its square.

We can assign this Lambda Expression to a delegate like:

```
delegate int del(int i);
static void Main(string[] args){
del myDelegate = y => y * y;
  int resultVariable= myDelegate(8);
}
```

In the above example, we assign a Lambda Expression “y => y * y” to a delegate “myDelegate” and then call myDelegate with a value “8” and stores the result of this Lambda Expression in an integer variable “resultVariable” which is equal to 64.

Example 2:

```
Class LambdaExpression{
static void Main(string[] args){
List<string> names=new List<string>();
names.Add(‘James’);
names.Add(‘Troy’);
names.Add(‘Harry’);
string resultString=names.Find(name=>name.Equals(‘Troy’));
}
}
```

In the above example, we declare a List of String values “names” then add different names to this list with the built-in function “list.Add”. Next we can find a specific name with the built-in function “list.Find” and pass a Lambda Expression “name=>name.Equals(‘Troy’)” to this built-in function which saves the required result to a string literal named “resultString”.

Example 3:

```
namespace lambdaexample

{

  class QueryOperator

  {

     static void Main(string[] args)

     {
            int[] numbers = { 1, 1, 2, 3, 5, 8, 13, 21, 34 };

            double averageNumber = numbers.Where(num => num % 2 == 1).Average();

            Console.WriteLine(averageNumber);

            Console.ReadLine();

        }

    }

}
```

In the above example, we declare an array of integers “numbers” which holds Fibonacci numbers, and we use the Lambda Expression with a where clause “numbers.Where(num => num % 2 == 1).Average()”. The part “num => num % 2 == 1” of the expression is getting the odd numbers from the list, and then retrieving their average with the built-in function “Average ()” and then saving it to a double type variable “averageNumber”. Lastly, it prints the result on the console by using this statement Console.WriteLine(averageNumber).

2. Expression Trees

A data structure that contains Expressions such as Lambda Expressions is known as an Expression tree. As Expressions are pieces of code, to put it simply, they are a tree structure with pieces of code in them, and can be executed by running the Lambda Expression over a set of data. Generally speaking, expression trees are a kind of Binary tree because by using binary trees you can quickly find the data you need. Expression trees provide us a method to translate the executable code into data. You can use it to transform the C# code. For example, LINQ query expression code can be transformed to operate on another SQL database process.

Example 4:

```
Func <int, int, int> function = (a, b) => a + b;
```

The above statement has three parts:

- A declaration: Func<int,int,int> function
- An equals operator: =
- A Lambda Expression: (a,b) => a+b;

The variable “function” is the executable code which can hold the result obtained from a Lambda Expression ((a,b) => a+b). Now you can call the Lambda Expression like this:

```
int c = function(5, 5);
```

When we call the function, the variable c will be set equal to 5+5, which is 10.

Since expression trees are a form of a data structure you can convert this executable code into a data structure by using a simple syntax of LINQ. You have to add the Linq.Expressions namespace in order to achieve this.

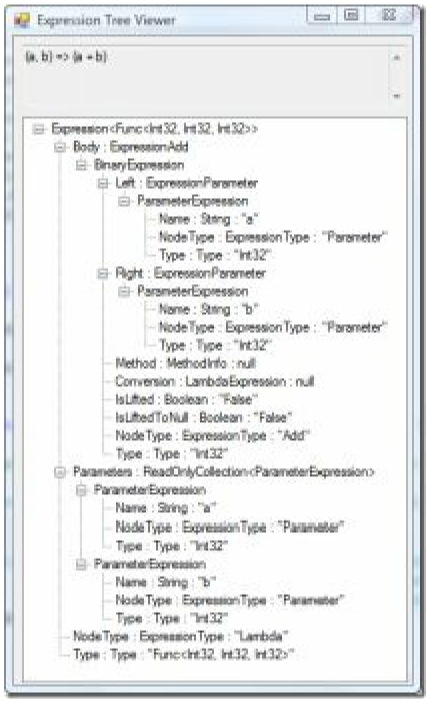
```
using System.Linq.Expressions;
```

Next, create the expression tree like this:

```
Expression<Func<int, int, int>> expression = (a,b) => a + b;
```

Now the above mentioned Lambda Expression is converted into an expression tree such as "Expression<X>". The identifier expression is not executable code; it is a data structure called an expression tree.

In Visual Studio (IDE), you can see the expression tree of the expression statement in a program "ExpressionTreeVisualizer".



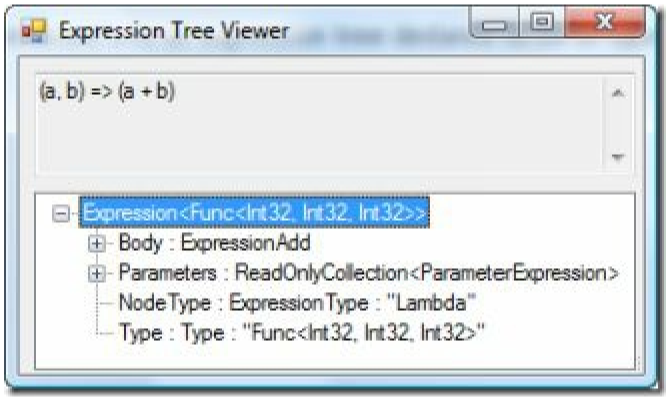
```
(a, b) => (a + b)

Expression<Func<Int32, Int32, Int32>>
  Body : ExpressionAdd
    BinaryExpression
      Left : ExpressionParameter
        ParameterExpression
          Name : String : "a"
          NodeType : ExpressionType : "Parameter"
          Type : Type : "Int32"
      Right : ExpressionParameter
        ParameterExpression
          Name : String : "b"
          NodeType : ExpressionType : "Parameter"
          Type : Type : "Int32"
      Method : MethodInfo : null
      Conversion : LambdaExpression : null
      IsLifted : Boolean : "False"
      IsLiftedToNull : Boolean : "False"
      NodeType : ExpressionType : "Add"
      Type : Type : "Int32"
  Parameters : ReadOnlyCollection<ParameterExpression>
    ParameterExpression
      Name : String : "a"
      NodeType : ExpressionType : "Parameter"
      Type : Type : "Int32"
    ParameterExpression
      Name : String : "b"
      NodeType : ExpressionType : "Parameter"
      Type : Type : "Int32"
  NodeType : ExpressionType : "Lambda"
  Type : Type : "Func<Int32, Int32, Int32>"
```

In the above diagram, you can see the Lambda Expression and its constituent part in TreeView Control.

In the above example, there are four properties in the Expression<x> class:

- Body
- Parameters
- NodeType
- Type

If we collapse the nodes of the tree shown as in the above diagram, the four properties will be clearly visible.



```
(a, b) => (a + b)
```

```
Expression<Func<Int32, Int32, Int32>>
    Body : ExpressionAdd
    Parameters : ReadOnlyCollection<ParameterExpression>
    NodeType : ExpressionType : "Lambda"
    Type : Type : "Func<Int32, Int32, Int32>"
```

```
BinaryExpression body = (BinaryExpression)expression.Body;
ParameterExpression left = (ParameterExpression)body.Left;
ParameterExpression right = (ParameterExpression)body.Right;
Console.WriteLine(expression.Body);
Console.WriteLine(‘ The left part of the expression: ‘ +
  ‘{0}{4} The NodeType: {1}{4} The right part: {2}{4} The Type: {3}{4}’,
  left.Name, body.NodeType, right.Name, body.Type, Environment.NewLine);
```

From the above lines of code, we can easily explore the expression tree. Let’s see now how the code works. First, you are declaring the variable “body” of type “BinaryExpression” which holds the body of the expression. In this example it is (a+b). Then, we extract the left parameter of the body with (ParameterExpression)body.Left” in a variable “left” of the type “parameterExpression.” In this case it is the variable “a.” Next, we are extracting the right parameter of the body with “(ParameterExpression)body.Right” in a variable “right” of the type “parameterExpression” which in our case is the variable “b”. Next, we are simply printing the body of the expression to the console as well as its NodeType, left & right part, and the type of expression with the help of built-in functions.

Exercise 2

Task 1:

Write a Lambda Expression which calculates the total number of scores greater than 60 in this series (91, 73, 84, 97, 75, 65, 40, and 89).

Solution

```
class SampleLambda
{
    static void Main()
    {
        int[] scores = { 91, 73, 84, 97, 75, 65, 40, 89};

    int highScores = scores.Where(n => n > 60).Count();

        Console.WriteLine('{0} scores are greater than 80', highScores);



    }
}
```

Task 2:

Create the query which retrieves the total scores for First grade Students, Second grade, and so on using Lambda Expression.

Solution

```
private static void StudentsByGrade()
{
    var categ =
    from student in students
    group student by student.grade into studentGroup
    select new { GradeLevel = studentGroup.Key, TotalScore = studentGroup.Sum(s => s.ExamScores.Sum()) };

    foreach (var cat in categ)
    {
        Console.WriteLine ('Key = {0} Sum = {1}', cat.GradeLevel, cat.TotalScore);
    }
}
```

Chapter 3: Generics

In C# Programming for Beginners, we were introduced to arrays. By now, most of us are well familiar with defining and using arrays in our programs. However, the problem with arrays is this: you have to define them "explicitly" before you can use them. That is, their data type, their size, etc, have to be defined before you use them.

Generics, on the other hand, are flexible, moldable, and efficient. With Generics, you don't have to specify the data type of a list, class, or method beforehand. In other words, they allow us to create strongly typed/type-safe data structures without specifying the exact data types at the time of their declaration. That means you can write "type less" classes, functions, and container lists. This is also their biggest advantage, since you can now create one class or method, and can use it over and over, each time with a different data type.

In this chapter, we will discuss Generics in depth.

Contents

1. Generics List <T> Collection

Generic collections (Lists) allow you to create data types that have the functionalities of both Arrays and Array Lists. From their signature and usage, they are just like Arrays. You declare them, initialize their members, and then use them. Let's see how.

Example 1:

```
using System;

namespace MySampleApplication
{
   class Program
   {
      static void Main(string[] args)
      {
Int x=0;
        List<int> myRows = new List <int> ();
          myRows.Add(1);
          myRows.Add(2);
          myRows.Add(3);
        While (x<myRows.Count)
{
    Console.WriteLine('Generics :{0}', myRows[x]);
++x;
}
        }
    }
}
```

Before diving into what the above code does, let’s first have a basic understanding of few things: First let's look at Generic List<int> “myRows”. Notice here that the type signature of a List class is List <T> where T can be any data type. It can be int, string, char, or any custom defined type. A List of type string will only hold strings, a list of custom defined type Bicycle will only occupy Bicycle objects (strictly typed). Hence our List myRows here will only occupy objects of type int.

The other thing to note here is the Add() function. Using myRows.Add(), you can define as many “int” entities in the list as you would like, unlike arrays where we have to specify the size in their definition. Add() however isn’t the only method that we can use with List <T> objects. There are numerous others including Contains, Remove, etc.

While Arrays have their length property that tells us their size, List <T> has the “Count” property that does the same thing.

The output of Example 1 is shown as follows:

Output 1:

```
Generics :{ 1} Generics :{ 2} Generics :{ 3}
```

The code works like this: We have declared a List myRows of type int. Using the Add method, we add three entities of value 1, 2, and 3. Then, using a while loop we print their values on the console using List’s Count Property.

2. Generic Methods

Generic methods, as well as generic classes, allow us to reuse our code in many different ways. Declaring and using a generic method that uses data types as parameters is both easy and fun to work do. While generic methods are mostly added into existing generic classes, they are equally handy where containing classes are not generic or where a method contains parameters that were not initially defined for the generic class parameters. Generic methods follow the basic pattern of “method name (type param syntax)”. Consider the following code, for instance:

Example 2:

```
namespace MySampleApplication
{
public static class MathExp
{
  public static T Max<T>(T first, params T[] values)
      where T : IComparable
   {
      T max = first;
      foreach (T item in values)
      {
          if (item.CompareTo(max) > 0)
          {max = item;  }
      }
      return max;
  }
  public static T Min<T>(T first, params T[] values)
      where T : IComparable
   {
      T mini = first;
        foreach (T item in values)
     {
     if (item.CompareTo(mini) < 0)
            { mini = item; }
        }
      return mini;
   }
}
```

The sample class “MathExp” has two generic methods: “Min <T> and Max <T>”. The functionality of both functions is quite simple. Given a list of values, Min<T> will find and return the minimum value from the list passed as a parameter, while the Max<T> will output the maximum or greatest value among the parameters. The syntax “<T>” simply refers to the fact that we can use both of these methods with any type of data values, i.e. int, strings, etc.

Consider now that we call our Max<T> function with int values first e.g., Console.WriteLine (MathEx.Max<int> (6, 56, 760)) and then with string type values e.g. (Console.WriteLine (MathEx.Max<string> (“Apple,” “Banana,” “Pineapple”)). Our output will be:

Output 2:

```
760
```

```
Pineapple
```

Note that we have specified the type of data values that we are passing in both cases above, i.e. <int> and <string>. It's fine to call generic methods this way, as long as it's for our own clarity. However, the C# compiler does not need that, since it can infer the data types itself while compiling which is also known as "type interfacing." To see "Type interfacing" at work, let's again call the above methods, this time without explicitly specifying their data types by using Console.WriteLine (MathEx.Max (6, 56, 760)) and Console.WriteLine (MathEx.Max ("Apple","Banana","Pineapple")). It should be no surprise that our output will be the same as before.

Output 3:

```
760
Pineapple
```

However, it's important to know here that type interfacing won't work, and in fact will result in a compile type error if we were to call the same function with a call like "MathEx.Max(7.0, 49) " with multiple value types, such as int and float. For type interfacing to work, all the parameter values must obey the defined method signature.

Exercise 3

Task:

Create a generic swap function.

Solution

```
using System;
using System.Collections.Generic;

namespace GenericSwapSolution
{
  class Program
  {
    static void SwapIt<T>(ref T left, ref T right)
    {
      T tmp;
      tmp = left;
      left = right;
      right = tmp;
```

```
        }
        static void Main(string[] args)
        {
          int x, y;
          char a, b;
          x = 20; y = 40;
          a = 'G'; b = 'I';
Console.WriteLine(‘Int values before swap are:’);
Console.WriteLine(‘x = {0}, y = {1}’, x, y);
Console.WriteLine(‘Char values before swap are:’);
  Console.WriteLine(‘a = {0}, b = {1}’, a, b);


          Swap<int> (ref x, ref y);
          Swap<char> (ref a, ref b);


Console.WriteLine(‘Int values after swap are:’);
Console.WriteLine(‘x = {0}, y = {1}’, x, y);
Console.WriteLine(‘Char values after swap are:’);
  Console.WriteLine(‘a = {0}, b = {1}’, a, b);
          Console.ReadKey();
        }
    }
}
```

Chapter 4: Extension Methods

Extension Methods (introduced in C# 3.0) provides programmers with a simple framework to "extend" the functionalities of existing types in their programs. "Static" in nature, Extension Methods gives you the freedom to add already present methods into your types without a need to create new derived types or modifying old ones. They use a "this" keyword in their parameters / parameters lists. Since they are "static" methods, it's essential for you to use them in static classes. However, they are used as if they were instance methods of the extended type.

In this chapter, we will discuss Extension Methods in depth, and will learn:

Contents

- **What Extension Methods are**
- **How to create them**
- **How to applying Extension Methods to existing types**

1. Extension Methods

The OCP, Open close principle, directs programmers to write code and functions in a way that they are open for extension all the time, but dead end on modification. In C#, Extension Methods are a practical case of OCP, which we can use in both custom defined and system defined user types to cater to our requirements.

It's also important for us to know a few basic facts about Extension Methods and how they work. One, they have only access (and hence can only use) the "public" properties of the data type they are extending. Two, their type signature should never be the same as any existing method of that type. Three, to use Extension Methods, their parent type must be in the namespace of the calling application. Four, in the case of a method overloading (methods with same signature), an instance method will be executed (called) instead of the Extended Method (as per the overload resolution mechanism), and lastly, we cannot apply Extension Methods on events, properties, and fields.

Before the addition of these methods, the common practice was to create custom types based on generic/primitive data types. With their addition however, we can now add functionalities to existing classes and functions without tampering with the code type. Let's consider a simple example where we are looking to provide the negative of a variable. The generic, or old way, of doing that is:

Example 1:

```
struct MyExample
{
    int num;

    public MyExample(int val)
    {
        this.num = val;
    }

    public int Negative()
    {
        return -num;
    }
}

static void Main(string[] args)
{


    MyExample i = new MyInt(34);
    Console.WriteLine(i.Negative());
}
```

While the code is quite basic, where we have created a strict “MyExample()” that has a simple function that returns negative of our passed value.

2. Creating Extension Methods

The code above works fine, but technically speaking we are not actually extending the type of our existing type. Instead we have created a new type altogether. Hence, if we are really looking to “extend” the functionality of our existing type, we should be doing something better and more generic than that.

Example 2:

```
static class MySampleExtension
{
    public static int Negative(this int val)
    {
        return -val;
    }
}

static void Main(string[] args)
{
    int i = 53;
    Console.WriteLine(i.Negative());
}
```

While the basic functionality and output of this is similar in that it will output the negative of the passed “int”, how it works is different. See the method declaration here: “public static int Negative()”. We are adding our method “negative” with type “int” and are bounding it to be called only with the int type. We are not creating a new type as in the previous example, and we are only extending functionality of the type “int”. Hence, to create our Extension Methods, we should:

- Have a “Static” class.
- Use a “public static” method with the same name as the class and have an explicit return type.
- Use the parameters type with the “this” keyword.

The “this” keyword is very important since it tells our compiler that we are “extending” this type and are not using it as an expected argument type.

3. Extension Methods on existing types

Using the above created Extension Method, let’s now see how we can call an Extension Method on our existing types. Also note that we used the “this” keyword above, but what about the cases where we want to use additional parameters? We can easily do this too by defining all such parameters with their expected data type after specifying our “extending” type with the “this” keyword. The code below will help us to better understand:

Example 3:

```
static class MySampleExtMethods
{
    public static int Negative(this int val)
    {
        return -val;
    }

    public static int Multiply(this int val, int multi)
    {
        return val * multi;
    }
}

static void Main(string[] args)
{
    int i = 8;
    Console.WriteLine(‘ Using Extension method the input yields: {0}’, i.Multiply(2));
}
```

Note that here we have defined another extension function called “Multiply”. While we are extending it on type “int” as specified by “this in val”, the second argument “int multi” lacks the “this” keyword. This is because we are using the second argument as an “argument,” and extending it with the first type. However, note that we can also pass “strings,” “char,” or “float” values in “arguments” as our needs warrant.

The rest of the code is quite straightforward. We have created two extension methods and are calling the “multiply” method with such an argument of “2,” and are extending it on the value of “I” which is 8. The output of above code will be:

Output 3:

```
16
```

Exercise 4

Task:

Create an Extension Method that checks if a string is an int or not, and if it is, convert the string's numeric value into a corresponding int value. For example, string str=”2345” should be converted into int or return false otherwise.

Solution

```
using System;
using System.Text;


namespace SampleExtensionMethod
{
    public static class SampleClass
    {
public static bool IsInt(this string x)
     {
         float outcome;
         return float.TryParse(x, out outcome);
     }


  public static int IntExt(this string str)
       {
            return Int32.Parse(str);
        }
    }
     class Caller
    {
  static void Main (string [] args)
        {
 string str= ’2345’;
if (str.IsInt())
     { Console.WriteLine(‘Yes It’s an integer’);
int nmb=str.IntExt();
      Console.WriteLine(‘The output using our custom Integer extension method: {0}’, nmb); }
else
     Console.WriteLine(‘No, it’s not an integer.’);


            Console.ReadLine();
         }
     }
}
```

Chapter 5 Nullable Types

C# provides some special types that consist of an additional null value along with the usual possible range of values of that data type. For example, an int32 data type can store a value ranging from -2147483648 to 2147483647 while the nullableint32 is able to store a null value along with its original possible range. Similarly, in the case of a Boolean variable, the Nullable of a Boolean variable is able to store a true, false, or a null value in it.

In this chapter we will study:

Contents

1. Structures of Nullable types in C#

The following table demonstrates the Nullable structure of a primitive data type along with the range of data that each data type can store. Nullable types have an additional value of Null.

Type	**Range**
Nullable Boolean	True or False or Null
Nullable byte	0 to 255 or Null
Nullable decimal	(-7.9 x 1028 to 7.9 x 1028) / 100 to 28 or Null
Nullable double	(+/-)5.0 x 10-324 to (+/-)1.7 x 10308 or Null
Nullable DateTime	Represents an instant in Time or Null

Nullable Int16	-32,768 to +32,767 or Null
Nullable Int32	-2,147,483,648 to 2,147,483,647 or Null
Nullable Int64	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 0r Null
Nullable Single	Single value or Null
Nullable char	U+0000 to U+FFFF or Null

2. Syntax for Nullable types in C#

Nullable types can be declared in two ways. The Syntax for the first way to declare a Nullable type is as follows:

```
System.Nullable<data_type> <variable_name> ;
```

It starts with the System.Nullable keyword, followed by data_type (i.e., int, double) followed by the variable name.

The Syntax for declaring a Nullable type the other way is:

```
< data_type> ? <variable_name> = null;
```

Starting with the data_type (i.e., int, double), it is followed by a question mark and then the name of variable.

Let's now see Nullable types at work.

Example 1:

```
Namespace nullable
{
   Class program
   {
     static void Main ()
   {
     int? a= null;
     int? b=10;
     if(a==null)
    {System.Console.WriteLine(b.Value)}
```

```
      else {System.Console.WriteLine('Undefined');}
      Console.readKey();
      }
   }
}
```

In the example above, we have declared two nullable integers a and b. int? a has a null value and int? b has a value of 10. The if/else construct is quite basic too; if “a” has a null value, the program will print out the value of int? b. Otherwise, it will print “undefined”.

Output 1:

```
10
```

The following example illustrates the Nullable type in action for Boolean and DateTime types.

Example 2:

```
Namespace nullable
{
    Class program
    {
      static void Main ()
    {
      int? a= null;
      int? b=5;
      Double? c=null;
      Double? d=6
      bool? Val= new bool?();
      DateTime? Start= DateTime.today;
      DateTime? End= null;
Console.Writeline('Showing values of Nullables: {0}, {1}, {2}, {3}',a,b,c,d);
Console.Writeline('A Nullable Boolean Variable: {0}',Val);
Console.Writeline(Start);
Console.Writeline('We don't know yet:', End);

Console.readKey();
      }
   }
}
```

In this program, we are using the Nullables of int, double, Boolean, and DateTime. Later, we are simply displaying them on the console. As the program compiles, it shows the values of the variables as:

Output 2:

```
Showing values of Nullables: , 5, , 6
A Nullable Boolean Variable:
6/8/2015 12:00:00 AM
We don’t know yet:
```

3. The HasValue and Value Property

The Nullable type instances have two properties. These are public and read-only properties.

- **HasValue Property:**

The HasValue always returns a Boolean value. It can be true or false. If the type contains an integer or a non-null value, the Hasvalue property is true. If the type doesn’t have a value or it is null, the Hasvalue property is false.

- **Has Property:**

The value is of the same type as the declared type. The Has property has a value if the Hasvalue property is true. If the Hasvalue property is false, the Has property will throw an Exception. See the code below to better understand this:

Example 3:

```
using System;
Namespace nullable
{
   Class program
   {
     static void Main ()
   {
     int? a= null;
     Console.WriteLine(a.HasValue); // HasValue property is false
     Console.WriteLine(a.Value);    // will cause an exception
     Console.readKey();
     }
  }
```

Because our variable “a” has a null value the “HasValue” property will be false. If we try to display the "Value" on the console, we get an exception.

Output 3:

```
False
```

Example 4:

```
using System;
Namespace nullable
{
   Class program
   {
     static void Main ()
   {
     int? a= null;
     Console.WriteLine(a.HasValue); // HasValue property is false

     a=5; //assigning value to variable
     Console.WriteLine(a.HasValue); // hasvalue Property is true because a has non-null value
     Console.WriteLine(a.Value);    // returns value of a
     Console.WriteLine(a);

     Console.readKey();
     }
  }
```

Output 4:

```
False
True
5
5
```

4. The Null Coalescing Operator

C# provides an operator to check the Null values. If it finds a Null value variable, it assigns a value to that variable. It is denoted by double question mark (??). We can use this operator for both Nullable types and reference types. It converts an operand to the type of another value type operand if the implicit conversion is possible. Let’s see how it works:

Example 5:

```
using System;
Namespace nullable
{
   Class program
   {
      static void Main ()
    {
      int? a= null;
      int? b=3;
      int c=a ?? 5;
      System.Console.WriteLine(‘Value of c is: {0}’,c);
      C=b ?? 5;
      System.Console.WriteLine(‘Value of c is: {0}’,c);
      Console.readKey();
      }
   }
}
```

Output 5:

```
Value of c is: 5
Value of c is: 3
```

Exercise 5

Task:

Write a program using a Nullable integer and double values implementing the HasValue property and the Null coalescing properties.

Solution:

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

namespace nullable
{
    class Program
    {
        static void Main(string[] args)
        {
            int? a = null;
            Console.WriteLine(a.HasValue);
                  int? b=3;
                  Double? d=null;
                  Double? e = 4;
int c=a ?? 6;
```

```
  System.Console.WriteLine(‘Value of Int c when assigned to null is: {0}’,c);
  c=b ?? 6;
  System.Console.WriteLine(‘Value of Int c reassigning is: {0}’,c);

  Double f = d ?? 8;
  System.Console.WriteLine(‘Value of Double f when assigned is:{0}’, f);
  f = e ?? 8;
  System.Console.WriteLine(‘Value of Double f reassigning is: {0}’, f);
            a = 1;      //assigning value to variable
            d = 2;   // aasigning value to variable
            Console.WriteLine(a.HasValue);
            Console.WriteLine(d.HasValue);
            Console.WriteLine(a.Value);
            Console.WriteLine(d.Value);
            Console.WriteLine(a);
            Console.WriteLine(d);

            Console.ReadKey();
        }
    }
}
```

Chapter 6: Anonymous Types

C# enables its users to create new data types. Anonymous types are data types that a user can create without defining them. These types are created at the point of instantiation. Anonymous types are both compiler generated and are reference types derived from objects. The compiler defines them itself on the basis of their properties (names, numbers, etc.). It is an important feature used in SQL like LINQ. Anonymous types are useful in LINQ queries because LINQ (language integrated query) is integrated into C#. In C#, the properties created for Anonymous types are read only. This concept was introduced in C# 3.

In this chapter, we will study "Anonymous" types including:

Contents

1. Var Statement:

It is a data type introduced in C# 3.0. The "Var" data type is used to declare the local variables implicitly. Let's have a look at a few valid Var statements.

```
var str='name';
var num='5';
var array=new[]{0,1,2};
```

Now look at how the compiler compiles these statements:

```
var str='name';     // string str='name';
var num='5'         // int num='5';
var array=new[]{0,1,2}; // int array=new[]{0,1,2};
```

Extending basic functionality, let's have a look at a simple code snippet:

Example 1:

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

namespace anonymous
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            var name = ‘Alberta johnson’;
            var number = 15;
            string s = ‘Canada’;
            var s2 = s;
            s2 = null;
            string s3 = null;
            var s4 = s3;
           Console.WriteLine(name);
            Console.WriteLine(number);
            Console.WriteLine(s);
            Console.WriteLine(s2);
            Console.WriteLine(s3);
            Console.WriteLine(s4);
            Console.ReadKey();
        }
    }
}
```

Output 1:

```
Alberta johnson
15
Canada
```

The values of variables var s2, var s3, and var s4 are null. The value of a var variable cannot be Null at compile time but can be Null at run time.

These types' var name and var string are not anonymous types. Actually, we are not declaring the data types of the variable, and the var statement shows that the compiler is itself deciding their data types.

Other Invalid var statements:

```
var a;              // invalid because it need to be initialized
var num=null        // cannot be Null at compile time
var v=‘Lord Belish’
v=15                // an integer value cannot be assigned to a string variable declared implicitly
```

2. Creating and Using Anonymous types in C#

In short, Anonymous types are reference types, and can be declared or created using the var statement using the same syntax as used for regular types. For example, if we need to create an Anonymous type to represent a point, we can do that simply:

```
Var point = new {x=17,y=9};
```

As we have discussed earlier, for var statements initialization is compulsory, and a variable cannot be initialized to null value.

```
Var point = new {x=null,y=9};  //wrong statement, cannot be null at compile time
```

Example 2

```
namespace anonymous
{
    class Program
    {
        static void Main(string[] args)
        {
            var Name = new { FirstName = 'Albert', LastName = 'Camus' };
            Console.WriteLine(Name.Firstname);
            Console.WriteLine(Name.Lastname);
            Console.ReadKey();

        }

    }

}
```

Output 2:

```
Albert
Camus
```

Let's now see how the complier creates an anonymous type.

```
namespace anonymous
{
    class Program
    {
        static void Main(string[] args)
        {
            var Employee = new { EmpName = 'George', EmpAddress = 'Camus' , Empsalary='25,000'};


        }

    }

}
```

At compile time, the compiler will create an anonymous type as follows:

```
namespace anonymous
{
   class Program
   {
private string Name;
private string Address;
int salary;

public string EmpName

{get {return EmpName;}

Set {EmpName=value;}

}

Public string EmpAddress

{

Get{return EmpAddress;}

Set{EmpAddress=value;}
}

Public int Empsalary

{

Get{return Empsalary;}
set{Empsalary=value;}

} }

}
```

We are naming the properties explicitly in all examples. We can do it implicitly if they are set on the basis of a property, or field, or a variable.

Example 3:

```
namespace anonymous
{
   class Program
   {
      static void Main(string[] args)
      {
         int variable = 42;
         var implicitProperties = new { variable, DateTime.Now };
         var explicitProperties = new { variable = variable, Now = DateTime.Now }; //same
                                                                        as above
         Console.WriteLine('Time is '+implicitProperties.Now+'And implicit Variable is ' +
                          implicitProperties.variable);
```

```
        Console.WriteLine('Time is '+explicitProperties.Now+'And Explicit Variable is ' +
                    explicitProperties.variable);


                    Console.ReadKey();

        }

    }

}
```

Output 3:

```
Time is 6/9/2015 4:30:06PM And Implicit variable is 42
Time is 6/9/2015 4:30:06PM And Explicit variable is 42
```

3. Comparing Two Anonymous Instances:

Anonymous types creates overrides of “Equals()” based on the underlying properties, so we can compare two anonymous variables. We can also get their Hash Codes using “GetHashCode()”**.** For example, if we had the following 3 points :

Example 4:

```
namespace anonymous
{
    class Program
    {
        static void Main(string[] args)
        {
            var point1 = new { A = 1, B = 2 };
            var point2 = new { A = 1, B = 2 };
            var point3 = new { b = 2, A = 1 };
    Console.WriteLine(point1.Equals(point2));
        // true, equal anonymous type instances always have same hash code
    Console.WriteLine(point1.GetHashCode() == point2.GetHashCode());
    Console.WriteLine(point2.Equals(point3));
        // quite possibly false
    Console.WriteLine(point2.GetHashCode() == point3.GetHashCode());
            Console.ReadKey();
        }
    }
}
```

Output 4:

```
True
True
False
False
```

Exercise 6

Task:

Write a program of an object collection having properties FirstName, LastName, DOB, and MiddleName. Return Firstname and LastName querying the data using Anonymous types.

Solution:

```
namespace anonymous
{
   class MyData
   {
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public DateTime DOB { get; set; }
   public string MiddleName { get; set; }
      static void Main(string[] args)
      {

         List<MyData> data = new List<MyData>();
   data.Add(new MyData { FirstName = ‘Shelby’, LastName = ‘Frank’, MiddleName = ‘N’, DOB = new
DateTime(1990, 12, 30) });
   data.Add(new MyData { FirstName = ‘sara’, LastName = ‘Simpson’, MiddleName = ‘G’, DOB = new
DateTime(1995, 11, 6) });
   data.Add(new MyData { FirstName = ‘Abigaile’, LastName = ‘jhonson’, MiddleName = ‘G’, DOB = new
DateTime(1993, 10, 8) });
   data.Add(new MyData { FirstName = ‘George’, LastName = ‘Kanes’, MiddleName = ‘P’, DOB = new
DateTime(1983, 6, 15) });
   data.Add(new MyData { FirstName = ‘Alberto’, LastName = ‘Delrio’, MiddleName = ‘K’, DOB = new
DateTime(1988, 7, 20) });

   var anonymousData= from people in data
              select new{people.FirstName, people.LastName};
         foreach(var n in anonymousData){Console.WriteLine(‘Name: ‘ + n.FirstName+ ‘ ‘ + n.LastName);}

   Console.ReadKey();
}

}
```

Chapter 7: LINQ part 1

LINQ is an acronym for “Language Integrated Query.” Those familiar with databases will already know what a “query” is. A query is basically a programmer's way of interacting with a database for manipulating data. That is, using queries, programmers can access database tables in order to insert, edit, retrieve, or delete data. Unlike a traditional mechanism of querying data, LINQ provides C# developers with a completely new way to access and work with multiple types of data including XML files, databases, Lists, and dynamic data.

LINQ functions have two basic units: sequences and elements. A LINQ sequence is a set of items which implements the IEnumerable<T> interface. Each item in the set is called the element. A typical example of a collection that implements the IEnumerable<T> interface is the array collection. Have a look at the following string array:

```
  string[] cities = {"Paris", "Washington", "Moscow", "Athens", "London" };
```

Here, this string type array contains names of different cities. This type of collection is called a local sequence, because all the items in the collection are in the local memory of the system.

Query Operators

Query operators in LINQ are used to take a LINQ sequence as input, transform the sequence, and return the transformed sequence as output. The Enumerable class of the System.Linq namespace contains around 40 query operators. In the following section, we will show the workings of some of the most commonly used LINQ operators.

i- Where operator

The “where” operator is used to filter a sequence based on a particular condition. For instance, if you want to get the name of all the cities in the ”cities” array where length of the city is greater than or equal to 6, you can do so in the following manner.

Example1:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
namespace MyCSharpApplication
```

```
{
    class Program
    {

        public static void Main()
        {

            string[] cities = {"Paris", "Washington", "Moscow","Athens", "London" };

            List<string> citiesendingwiths = (from c in cities
                                              where c.Length >=6
                                              select c).ToList();

            foreach(string c in citiesendingwiths)
            {
                Console.WriteLine(c);
            }

            Console.Read();
        }
    }

}
```

The sequence of a LINQ query is quite similar to that of SQL query. In Example1, we simply fetched all the cities with a length greater than or equal to 6, and then displayed it on the console screen. In this case, all the city names in the cities array will be displayed except “Paris,” which is only 5 characters in lenght.

The LINQ syntax used in Example1 is commonly referred to as query syntax owing to its similarity to the SQL query syntax. However, there is another way to execute LINQ queries using Lambda Expressions. This is known as fluent syntax. Example 2 demonstrates how fluent syntax is used to achieve the same functionality of Example1.

Example 2

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
namespace MyCSharpApplication
{
    class Program
    {

        public static void Main()
        {
            string[] cities = {"Paris", "Washington", "Moscow","Athens", "London" };

            List<string> citiesendingwiths = cities.Where(c => c.Length >= 6).ToList();

            foreach(string c in citiesendingwiths)
            {
                Console.WriteLine(c);
            }

            Console.Read();
        }
    }

}
```

Like SQL, logical operators can also be used along with comparison operators in LINQ. For instance, if you want to retrieve the names of all the cities which have a character length greater than 5 and have “o” in their names, you can use the AND logical operator as follows:

Example3:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
namespace MyCSharpApplication
{
    class Program
    {

        public static void Main()
        {

            string[] cities = {"Paris", "Washington", "Moscow","Athens", "London" };

            List<string> citiesendingwiths = cities.Where(c => c.Length >= 6 && c.Contains("o")).ToList();

            foreach(string c in citiesendingwiths)
            {
                Console.WriteLine(c);
            }

            Console.Read();
        }
    }
}
```

This time, only the cities Washington, Moscow, and London will be displayed on the screen since these are the only cities having “o” in their names and a character length greater than or equal to 6.

2- Select

A Select query is mostly used with objects with multiple members. Select can be used to retrieve a particular member of all the objects in the collection. Select can also be used by any object as a whole. Our next example demonstrates the basic use of the Select query. Have a look at it:

Example4:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
namespace MyCSharpApplication
{

    public class Person
    {

        public int age;
        public string name;

        public Person(int age, string name)
        {
            this.age = age;
            this.name = name;
        }

    }

    class Program
    {

        public static void Main()
        {

            Person p1 = new Person(9, "John");
            Person p2 = new Person(8, "Jack");
            Person p3 = new Person(13, "Mic");
            Person p4 = new Person(15, "Evens");
            Person p5 = new Person(6, "Roddy");
            List<Person> plist = new List<Person>();
            plist.Add(p1);
            plist.Add(p2);
            plist.Add(p3);
            plist.Add(p4);
            plist.Add(p5);

            List<string> personnames = plist.Where(p => p.age <= 10).Select(per => per.name).ToList();

            foreach(string pname in personnames)
            {
                Console.WriteLine(pname);
            }

            Console.Read();
        }
    }

}
```

In Example4, we created a Person class with two member variables: age and name. These variables can be initialized via a constructor. Inside the main method of the program class, we created 5 objects of the Person class and then stored them in the Person list collection named plist. We then executed a LINQ query having the where and select operators. The where operator is used to filter all the persons whose age is less than 10. If we do not use a select operator here, the whole Person object will be retrieved. However, we are only interested in retrieving the names of the persons with an age less than 10. Therefore, we used the Select operator and passed it a Lambda Expression which selects the person name.

LINQ is a very vast subject, and requires a complete book in itself. We will end the discussion of the operators here. In the next two chapters, I will explain the process of connecting LINQ to SQL as well as connecting LINQ to XML.

Exercise 7

Task:

There exists a Car class with the following structure:

```
    public class Car
    {
        public int price;
        public string name;

        public Car(int cprice, string cname)
        {
            this.price = cprice;
            this.name = cname;
        }

    }
```

In the Main method, 5 objects of the Car class have been initialized and stored in a List collection. This is shown below:

```
    Car c1 = new Car(120000, "Honda");
            Car c2 = new Car(90000, "Toyota");
            Car c3 = new Car(130000, "BMW");
            Car c4 = new Car(150000, "Ferrari");
            Car c5 = new Car(30000, "Suzuki");

            List<Car> clist = new List<Car>();
            clist.Add(c1);
            clist.Add(c2);
            clist.Add(c3);
            clist.Add(c4);
            clist.Add(c5);
```

Your task is to Select names of all the cars where the price of the car is greater than or equal to 125,000.

Solution:

```
          List<string> carnames = clist.Where(c => c.price >= 125000).Select(car => car.name).ToList();

            foreach(string cname in carnames)
            {
                Console.WriteLine(cname);
            }
```

Chapter 8: LINQ part 2

What gives LINQ an edge over other technologies is its flexibility to work with multiple types of data. That is, with LINQ, the same query syntax can be used to handle incoming data, independent of the data source type, whereas all other technologies require writing separate queries for each source. SQL queries are required for interaction with the SQL server and Xquery for the XML data type.

In this chapter, we will see the relationship between:

Contents

- **LINQ and SQL**
- **LINQ and Lambda Expressions**

1. LINQ and SQL:

LINQ is a smart alternative for the old mode of accessing SQL data using SQL queries. The term "LINQ to SQL" is often used to describe the relationship through which we can access SQL databases using LINQ. The first step to do that is to map our existing/target SQL database to LINQ.

1. Mapping LINQ to SQL:

Mapping LINQ to SQL refers to .Net recognizing our existing database as Objects (Classes). This is quite easy actually; open Visual Studio-> target project in solution explorer. Next, click Add->New Item. Select "Data" from the "Categories" options, and choose "LINQ to SQL Classes" from the Templates on the left. A ".dbml" file will result with a Graphic User Interface (GUI). This GUI has two parts, one that allows you to drag and drop tables to auto create classes from them and the other part where we can drop stored procedures. Select, drag and drop all essential tables and procedures as needed.

2. Selecting Data:

As soon as we create our ".dbml" file, a corresponding "DataContext" class file is created on its own by the .NET framework that does all the communication with databases. LINQ queries then use objects from these classes to work with databases. Let's see this from the example below:

Example 1:

```
public bool checkValidUser(string Name, string passcode)
{
DBToysDataContext sampleDB = new DBToysDataContext();
var getter = from u in sampleDB.Users
                    where u.Username == Name
                    && u.Password == passcode
                    select u;
return Enumerable.Count(getter) > 0;
}
```

Before diving into the code, it's important to know that when we mapped our "DBToys.dbml" file, a "DBToysDataContext" class file was created automatically. In the code above, we passed two strings "Name" and "Passcode" to our function "checkValidUser" which validates a user entered a name and password against the table "Users" from the database Toys. In the first line inside the function, we instantiated an object "sampleDB" to access our Toys database using the "DBToysDataContext" class file. Visual studio treats "u" from the line "from u in sampleDB.Users" as an object of the "Users" class, referring to our "Users" table from the Toys database. Next, note that we are passing our incoming column/field values as an object of type "var." The Var data type refers to dynamic data. Whatever types of data our LINQ query supplies it can be stored in a variable "getter." Here we are only retrieving and saving the username and password from the "Users" table. Lastly, the function "Enumerable.Count" returns the total number of data rows returned by our LINQ query.

However, there's an alternate way to access the same data without using SQL like syntax in LINQ.

Example 2

```
public bool checkValidUser(string Name, string passcode)
{
     DBToysDataContext sampleDB = new DBToysDataContext();
     List<Users> getter = sampleDB.Users.Where(u => u.Username == Name && u.Password==passcode);
     if(users.Count>0)
     {
          return true;
     }
     return false;
}
```

Here, instead of using the traditional SQL syntax, we are using the "Where" method to directly access data from the "Users" table of the Toys database. The rest of the working process is the same as before, except here we have used the "List" data type to store the values returned from our LINQ query.

Example 3

```
    public User bringUser(string name)
{
    DBToysDataContext sampleDB = new DBToysDataContext();
    User use = sampleDB.Users.Single(u, u.UserName=>name);
    return use;
}
```

For cases where we only want to retrieve and send only a single row (object) using our LINQ query, we can easily do that as well. The function “bringUser” in its only argument accepts a name to be matched against objects in our “Users” table. The method “Single” in the line “sampleDB.Users.Single()” looks for a match against the provided name, and upon success returns only a single row(object) as required.

- **LINQ and Lambda Expressions**

While we have already discussed Lambda Expressions earlier, LINQ queries mostly imply Lambda Expressions in dealing with collections or lists of data to filter list items based on some specific criteria. Let’s see how:

Example 4:

```
IEnumerable <SelectListItem> toys = database.Toys
        .Where(toy => toy.Tag == curTag.ID)
        .Select(toy => new SelectListItem { Value = toy.Name, Text = toy.ID });

ViewBag.toySelector = toys;
```

In the code above, we have declared the variable “toys” and are casting it to type “SelectListItem” to save the resultant row from our LINQ query. We have used two methods, “Where” and “Select” to find the target toy that matches our query. The line ‘”toy => toy.Tag == curTag.ID” selects a toy based on a passed tag, whereas the line “toy => new SelectListItem {Value = toy.Name, Text = toy.ID” selects a particular toy based on a passed ID and name. The resultant toy that meets this criterion is saved in the “toys” variable.

Exercise 8

Task:

Using Customer table from DB NorthWind (Visual Studio), apply LINQ to insert/update and delete a customer where the ID is 20.

Solution:

```
namespace MySolutionApplication
{
        public static void InsertCustomer(string id, string name, string city, string phone, string fax)
    {

        NorthWindDataClassesDataContext dco = new
        NorthWindDataClassesDataContext();

        var lookCustomer = ( from c in dc.GetTable<Customer>() where c.CustomerID == id
                    select c).SingleOrDefault();

        if(lookCustomer == null)
        {
           try
           {


              Table<Customer> cus = Accessor.GetCustomerTable();
              Customer cust = new Customer();
              cust.CustomerID = id;
              cust.ContactName = name;
              cust.City = city;
              cust.Phone = phone;
              cust.Fax = fax;
              cus.InsertOnSubmit(cust);
              cus.Context.SubmitChanges();
           }
           catch (Exception ex)
           {
              throw ex;
           }
        }
        else
        {
           try
           {
             lookCustomer.ContactName = name;
              lookCustomer.City = city;
              lookCustomer.Phone = phone;
              lookCustomer.Fax = fax;

              dco.SubmitChanges();
           }
           catch (Exception ex)
           {
              throw ex;
           }
        }
    }
    public static void DeleteCustomer(string ID)
    {
        NorthWindDataClassesDataContext dco = new
        NorthWindDataClassesDataContext();

        var lookCustomer = (from c in dc.GetTable<Customer>()    where c.CustomerID ==id
    select c).SingleOrDefault();

        try
        {
           dco.Customers.DeleteOnSubmit(lookCustomer);
           dco.SubmitChanges();
        }
        catch (Exception ex)
        {
           throw ex;
        }
    }
```

Chapter 9: LINQ part 3

In the previous chapter, we saw how we can use LINQ to work mainly with our SQL databases. In this chapter, however, we will see how we can use LINQ to work with XML data. XML, Extensible Markup Language, is used extensively on the World Wide Web and is much more than just a set of static text based labels. Also called self-describing and self-defining data, XML with its highly standardized and equally customizable tag usage is globally employed in sharing, accessing, and manipulating data.

Hence, in this chapter, we will see how we can use LINQ to:

Contents

1. Retrieve and delete XML data

All XML data files have a root/parent tag (element) that not only encapsulates and defines the type of child records, but also defines their attributes. The subsequent child records (elements) contain actual data that we can manipulate as per our needs. Look at this sample XML code that we are using during rest of this chapter for dealing with LINQ queries.

Sample XML Data

```
<? Xml version=‘1.0’ encoding=‘utf-8’?>
<Toys>
  <Toy ID=‘1’>
    <Name>Barbie</Name>
    <Price>$0.45</Price>
  </Toy>
  <Toy ID=‘2’>
    <Name>Pigeon</Name>
    <Price>$0.40</Price>
  </Toy>
  <Toy ID=‘3’>
    <Name>Crow</Name>
    <Price>$0.55</Price>
  </Toy>
</Toys>
```

Here “Toys” is our root element with multiple child “Toy” elements. Each child toy has an “ID” attribute with “Price” and “Name” inner elements.

Now, let’s see how we can retrieve child toys from this XML data using LINQ queries.

Example 1

```
private string file = ‘SampleData.xml’;

      private void GetData()
      {
         try
         {
            XDocument doc = XDocument.Load(file);
            var comingToys = from toy in doc.Descendants(‘Toy’)
       select new
            {
  ID= Convert.ToInt32(toy.Attribute(‘ID’).Value),
  Name = toy.Element(‘Name’).Value ,
Price = toy.Element(‘Price’).Value
                         };

            foreach (var x in toys)
            {
     Console.WriteLine(‘Toy ID’, x[ID]);
Console.WriteLine(‘Toy Name’, x[Name]);
Console.WriteLine(‘Toy Price’, x[Price]);                    }
         }
         catch (Exception err)
         {
            MessageBox.Show(err.Message);
         }
      }
```

The first line here points to our sample XML data file “SampleData.xml”. Inside the function, we first load our xml file using ‘Load (file)’ function. After loading our target file, we retrieve subsequent child “toy” elements using “Doc.descendants()” function. Next we select each child “toy” and retrieve each toy’s ID attribute as well as its inner elements and pass them onto our dynamic variable “comingToys”. The foreach” loop at the end then displays all the retrieved data.

Output 1:

```
Toy ID 1 Toy Name Barbie Toy Price $0.45
Toy ID 1 Toy Name Pigeon Toy Price $0.40
Toy ID 1 Toy Name Crow Toy Price $0.55
```

Similarly, if let’s say we want to remove a child element from our sample XML data file, we can do that like this.

Example 2

```
private string file = ‘SampleData.xml’;

Private void DeleteData (int id)
      {
          try
          {
              XDocument sampleXML = XDocument.Load(file);
              XElement cToy = sampleXML.Descendants(‘Toy’).Where(c => c.Attribute(‘ID’).Value.Equals(id);
              cToy.Remove();
              sampleXML.Save(file);
          }
          catch (Exception e)
          {
              MessageBox.Show(e.Message);
          }
      }
```

The code here, after loading the target file, traverses the child toys and looks for a match against the passed in “toy ID”. Once it finds that child, it removes that child by calling the “Remove()” function and saves the resulting file.

2. Insert and Update XML data:

Inserting data into existing XML files is not much different from retrieving data from the same file. Essentially, we will need an object of the type “XElement” with the same signature as our existing child/elements in the target XML file. The next step involves inserting that “XElement” object into our data file using an “XDocument” object.

Example 3

```
private string file = ‘SampleData.xml’;

private void InsertData(string name, string price)
       {
           try
           {
  XDocument doc = XDocument.Load(file);
               XElement newToy = new XElement(‘Toy’,  new XElement(‘Name’, name), new XElement(‘Price’,
price));
  Var lastToy = doc.Descendants(‘Toy’).Last();
```

```
    Int newID = Convert.ToInt32 (lastToy.Attribute (‘ID’).Value);
    newToy.SetAttributeValue(‘ID’,++newID);
      doc.Element (‘Toys’).Add (newToy);
          doc.Save (file);
            }
            catch (Exception err)
            {
               MessageBox.Show (err.Message);
            }
        }
```

After creating an object of the type “XDocument”, we have created an “XElement” type object with same signature as our existing “Toy” elements in the line “XElement newToy = new XElement(‘Toy”, new XElement(‘Name’, name), new XElement(‘Price’, price)’". Next, we get the last toy element in var lastToy using the “doc.Descendants(‘Toy’).Last()” function. Once we have that, we get its “ID” value and then increment it while setting that attribute to our new toy element in line “newToy.SetAttributeValue(‘ID’,++newID)”. Finally, we insert our “toy” object using the “Add()” function, and then save our changes to the XML file.

Output 3:

Our sample XML data file will now contain a fourth child. In the file, it will look like this if, for instance, we had passed “Rabbit” as the name and “$0.65” as the price to the above function.

```
<Toy ID=‘4’>
   <Name>Rabbit</Name>
   <Price>$0.65</Price>
 </Toy>
```

In order to make changes to existing data or elements in our XML file, we first need to locate our target element against some specific criteria whose data values we would like to alter. The line “XElement cToy = doc.Descendants(‘Toy’).Where (c=>c.Attribute(‘ID’).Value.Equals(id)” does exactly that for us. It looks for a match against our provided “id” and allows our “XElement” object “cToy” to point to that element. Once we have a reference to that element, we can easily update its inner elements values using “cToy.Element(‘Price’).Value = price”. Lastly, we need to save our XML file (“XDocument” object) to reflect these changes.

Example 4

```
private string file = ‘SampleData.xml’;
private void UpdateData(string name, string price, int id)
```

```
        {
            try
            {
    XDocument doc = XDocument.Load(file);
   XElement cToy = doc.Descendants('Toy').Where (c=>c.Attribute('ID').Value.Equals(id);
  cToy.Element('Name').Value = name;
  cToy.Element('Price').Value = price;
        doc.Save(file);
            }
            catch (Exception err)
            {
                MessageBox.Show(err.Message);
            }
        }
```

Output 4:

Let's say that we have passed these values as arguments to our function above: id=1, name=Crocodile, price=$0.75. After executing the above function, our sample XML data file will no longer have the toy name=Barbie at id=1. Instead it will have a toy name=crocodile in its place.

```
<Toy ID='1'>
    <Name>Crocodile</Name>
    <Price>$0.75</Price>
  </Toy>
```

Exercises 9

Task:

From the given XML sample data, use LINQ to create an XML file with the same data values.

XML Data:

```
<?xml version='1.0' encoding='utf-8' ?>
<Students>
   <student id='1' regular='no'>
      <name>Alberto Gustavo</name>
      <adm_date>7/31/1996</adm_date>
   </student>
   <student id='3' regular='yes'>
      <name>Kim Kards</name>
      <adm_date>12/12/1997</adm_date>
   </ student >
   < student id=‘8’ regular=‘no’>
      <name>Carl Mills</name>
      < adm _date>2/6/1998</ adm _date>
   </ student >
   < student id=‘9’ regular=‘yes’>
      <name>Adams</name>
      <adm_date>2/6/1998</ adm _date>
   </ student >
</Students >
```

Solution:

```
public static void CreateStudents()
{
   XDocument doc = new XDocument(
      new XDeclaration(‘1.0’, ‘utf-8’, ‘yes’),
      new XComment(‘Solution XML file’),
      new XElement(‘Students’,
         new XElement(‘student’,
            new XAttribute(‘id’, 1),
            new XAttribute(‘regular’, ‘false’),
               new XElement(‘name’, ‘ Alberto Gustavo’),
               new XElement(‘adm_date’, ‘7/31/1996’)),
         new XElement(‘student’,
            new XAttribute(‘id’, 3),
            new XAttribute(‘regular’, ‘true’),
               new XElement(‘name’, ‘Kim Kards’),
               new XElement(‘adm_date’, ‘12/12/1997’)),
         new XElement(‘student’,
            new XAttribute(‘id’, 8),
            new XAttribute(‘regular’, ‘false’),
               new XElement(‘name’, ‘Carl Mills’),
               new XElement(‘adm_date’, ‘2/6/1998’)),
         new XElement(‘student’,
            new XAttribute(‘id’, 9),
            new XAttribute(‘regular’, ‘false’),
               new XElement(‘name’, ‘Adams’),
               new XElement(‘adm_date’, ‘2/6/1998’))
      )
   );
}
```

Chapter 10: Asynchronous Programming

In this chapter, we are going to study a new feature of C# 5.0 that allows you to write your own asynchronous code. Imagine that you are working on a Windows form application, and you click a button to download an image from the web synchronously. It would take more than 30 seconds to download the image, and during this time your application becomes unresponsive, which from a usability perspective isn't a good thing. Hence, a better way to allow the downloading of the image is to do so asynchronously.

In this chapter, we will understand what "asynchronously" means in C#, and how we can use this feature in our applications.

Contents

- **Asynchronous Programming using async and await**

The problem we discussed above can be easily avoided using two keywords "**async"** and "**await**" in our programs. Let's discuss each of two now:

1. **Async**

If we specify this keyword before a function while declaring it, it becomes an asynchronous function. By using this keyword, you can use the resources provided by the .NET framework to create an asynchronous framework, and the function will be called asynchronously. The syntax of asynchronous methods is like this:

```
public async void MyProcess()
            { }
```

The above declared function is ready to be called asynchronously.

2. **Await**

While the "Async" keyword is used to tell the compiler that the function is asynchronous, the function also needs to have "await" in it. The syntax of await is as follows:

```
public async void MyProcess()
{
// do the asynchronous work
await Task.delay(5);
}
```

The above mentioned method will do the work after a delay of 5 seconds.

Let’s talk about the problem stated at the beginning of the chapter. The following lines of C# code will download the image from the web synchronously.

Example 1

```
private void button_Click(object sender, EventArgs e)
{
    WebClient image = new WebClient();
    byte[] imageData = image.DownloadData(‘http://urlOfTheImage’);
    this.imageView.Image = Image.FromStream(new MemoryStream(imageData));
}
```

Your application will become unresponsive during the execution of this code. While on the other hand, we can easily do it using new keywords provided by C# 5.0 async and await. Let’s see how:

Example 2

```
private async void button_Click(object sender, EventArgs e)
{
    WebClient image = new WebClient();
    byte[] imageData = await image.DownloadDataTaskAsync(‘http://urlOfTheImage’);
    this.imageView.Image = Image.FromStream(new MemoryStream(imageData));
}
```

The above mentioned code looks identical to the one in Example 1, but it is not. There are three differences:

- The addition of the async keyword in the method.
- The call to download the image from the web is preceded by await.
- DownloadData is replaced by its asynchronous counterpart DownloadDataTaskAsync.

The “DownloadData” method of the WebClient class downloads the data synchronously and then after downloading returns the control to the caller which causes the application to become unresponsive. On the other hand, “DownloadDataTaskAsync” returns immediately and downloads the data asynchronously. The await keyword is the most interesting part as it releases the UI thread unless the download is complete. Whenever the code encounters the await keyword, the function returns, and when the specified operation completes, the function resumes. It continues executing from where it has stopped.

NOTE: Every asynchronous method can return three types of values.

- Void: return nothing.
- Task: It will perform one operation.
- Task<T>: Will return a task with a T type parameter.

Task: A Task returns no value (it is void). A Task<int> returns an element of the type int. This is a generic type .

Note: An async method willrun synchronously if it does not contain the await keyword .

Example 3:

```
using System;
using System.IO;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {

            Task task = new Task(ProcessDataAsync);
            task.Start();
            task.Wait();
            Console.ReadLine();
    }

    static async void ProcessDataAsync()
    {

            Task<int> task = HandleFileAsync(‘C:\\enable1.txt’);


            Console.WriteLine(‘Please wait patiently ‘ +
                ‘while I do something important.’);


            int x = await task;
            Console.WriteLine(‘Count: ‘ + x);
    }

    static async Task<int> HandleFileAsync(string file)
    {
            Console.WriteLine(‘HandleFile enter’);
            int count = 0;


            using (StreamReader reader = new StreamReader(file))
            {
                string v = await reader.ReadToEndAsync();


                count += v.Length;


                for (int i = 0; i < 10000; i++)
                {
                        int x = v.GetHashCode();
                        if (x == 0)
                        {
                            count--;
                        }
                }
            }
            Console.WriteLine('HandleFile exit');
            return count;
    }
}
```

Output initial 3:

```
HandleFile enter
Please wait patiently while I do something important.
```

Output final 3:

```
HandleFile enter
Please wait patiently while I do something important.
HandleFile exit
Count: 1916146
```

In the above example, after the main method we create an instance of Task with the ProcessDataAsync method passed as an argument. Then in the next line we start this task with "task.Start()" and then wait for it to finish with the "task.Wait()" method. The method "ProcessDataAsync" is an asynchronous method as the "async" in the method signature tells us. Hence the keyword await is mandatory here. The first line inside method, "Task<int> task = HandleFileAsync('C:\\enable1.txt')" calls another method "HandleFileAsync". As this is an asynchronous method, control returns here before the "HandleFileAsync" method returns as we discussed in the above example. Meanwhile as "HandleFileAsync" method is performing its task we display a message on the screen "HandleFile enter" & "Please wait patiently while I do something important." Here, the first line is from the method "HandleFileAsync." This happens because when the method is called, it is gets printed to the screen and then control returns back to the method “ProcessDataAsync”. Next the second line is printed to the screen. Now the next line “await task” of the method “ProcessDataAsync” tells it to wait for the HandleFile task to complete, and then assigns the total computed result to the variable “x” and finally prints it on the screen.

Now let’s talk about the next method “HandleFileAsync,” which is also an asynchronous method as the async in the method signature tells us, and hence has an await keyword as well. After the first line prints on the screen, we initialize a dummy integer variable “count” and set it equal to “0” as we passed the location of a file in the argument when we call this method which is “C:\\enable1.txt”. Now in order to read the data from this file, we initialize a reader of the type “StreamReader” and pass it the location of our file. To read this file, we use the asynchronous built-in method “reader.ReadToEndAsync();” and tell it to wait until it finishes reading the file with the “await” keyword. Then we assign the result to a string type variable “v”. Finally, we add the total length of the string to the dummy variable “count”. Then we put in some dummy code to count this value. This dummy code is just for your understanding, and at the end when this method returns, a simple line “HandleFile exit” prints on the screen. The dummy value also gets printed.

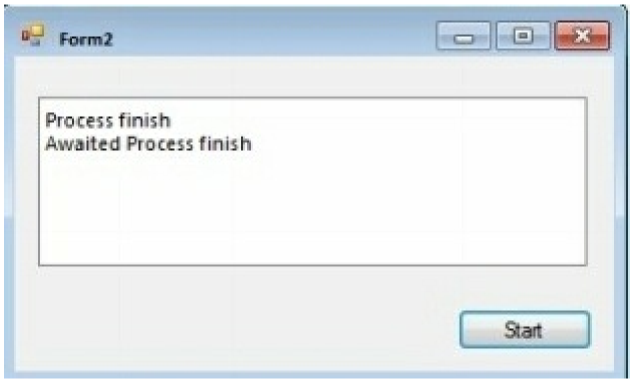
Exercise 10

Task:

Write code for a Windows application which creates an asynchronous function that will wait for two minutes.

Solution

```
using System;
using System.ComponentModel;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace WindowsFormsApplication
{
    public partial class Form2 : Form
    {
        public Form2()
        {
            InitializeComponent();
        }

        public static Task awaitedProcess()
        {
            return Task.Run(() =>
            {
                System.Threading.Thread.Sleep(2000);
            });
        }

        public async void myProcess ()
        {
            await awaitedProcess();
            this.listBox.Items.Add(‘Awaited Process finish’);
        }

        private void Form2_Load(object sender, EventArgs e)
        {
        }

        private async void button1_Click(object sender, EventArgs e)
        {
             myProcess();
             this.listBox.Items.Add(‘Process finish’);
        }
    }
}
```



```
Process finish
Awaited Process finish
```