VTC Academy
Thinking Ahead, Beyond Boundaries

Object Oriented Programming

Advanced C# Programming

Session 12

Objectives

- Nullable types
- Regular expressions
- Anonymous method
- LINQ
- Unsafe code

Nullable Types

- C# provides a special data types, the **nullable** types, to which you can assign normal range of values as well as null values.
- For example, you can store any value from -2,147,483,648 to 2,147,483,647 or null in a Nullable<Int32> variable. Similarly, you can assign true, false, or null in a Nullable<bool> variable.
- Syntax for declaring a **nullable** type is as follows:

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

Nullable Types Example

```
using System;
namespace NullableTypesDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int? num1 = null;
            int? num2 = 45;
            double? num3 = new double?();
            double? num4 = 3.14157;

            bool? boolval = new bool?();

            // display the values

            Console.WriteLine("Nullables at Show: {0}, {1}, {2}, {3}",
                                                  num1, num2, num3, num4);
            Console.WriteLine("A Nullable boolean value: {0}", boolval);
            Console.ReadLine();
        }
    }
}
```

Null Coalescing Operator (??)

- The null coalescing operator is used with the nullable value types and reference types. It is used for converting an operand to the type of another nullable (or not) value type operand, where an implicit conversion is possible.
- If the value of the first operand is null, then the operator returns the value of the second operand, otherwise it returns the value of the first operand.

Null Coalescing Operator (??)

```
using System;
namespace CoalescingOperatorDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         double? num1 = null;
         double? num2 = 3.14157;
         double num3;
         num3 = num1 ?? 5.34;
         Console.WriteLine(" Value of num3: {0}", num3);
         num3 = num2 ?? 5.34;
         Console.WriteLine(" Value of num3: {0}", num3);
         Console.ReadLine();
      }
   }
}
```

Regular Expressions

- A **regular expression** is a pattern that could be matched against an input text
- The .Net framework provides a regular expression engine that allows such matching.
- A pattern consists of one or more character literals, operators, or constructs
- Regex class is used for representing a regular expression

Regex Class

- Regex class is used for representing a regular expression
- Regex class represents the .NET Framework's regular expression engine.
- Regex class can be used to quickly parse large amounts of text to find specific character patterns; to extract, edit, replace, or delete text substrings; or to add the extracted strings to a collection to generate a report.

Methods of Regex Class

No.	Methods
1	**public bool IsMatch(string input)** Indicates whether the regular expression specified in the Regex constructor finds a match in a specified input string.
2	**public bool IsMatch(string input, int startat)** Indicates whether the regular expression specified in the Regex constructor finds a match in the specified input string, beginning at the specified starting position in the string.
3	**public static bool IsMatch(string input, string pattern)** Indicates whether the specified regular expression finds a match in the specified input string.
5	**public string Replace(string input, string replacement)** In a specified input string, replaces all strings that match a regular expression pattern with a specified replacement string.
6	**public string[] Split(string input)** Splits an input string into an array of substrings at the positions defined by a regular expression pattern specified in the Regex constructor.

Regex Class Example 1

Example matches words that start with 'S':

```
using System;
using System.Text.RegularExpressions;
namespace RegExDemo {
   class Program {
      private static void showMatch(string text, string expr)
      {
         Console.WriteLine("The Expression: " + expr);
         MatchCollection mc = Regex.Matches(text, expr);
         foreach (Match m in mc)
         {
            Console.WriteLine(m);
         }
      }
      static void Main(string[] args)
      {
         string str = "A Thousand Splendid Suns";
         Console.WriteLine("Matching words that start with 'S': ");
         showMatch(str, @"\bS\S*");
         Console.ReadKey();
      }
   }
}
```

Regex Class Example 2

Example matches words that start with 'm' and ends with 'e':

```
using System;
using System.Text.RegularExpressions;
namespace RegExApplication {
   class Program {
      private static void showMatch(string text, string expr)
      {
         Console.WriteLine("The Expression: " + expr);
         MatchCollection mc = Regex.Matches(text, expr);
         foreach (Match m in mc)
         {
            Console.WriteLine(m);
         }
      }
      static void Main(string[] args)
      {
         string str = "make maze and manage to measure it";
         Console.WriteLine("Matching words start with 'm' and ends with
                                                                  'e':");
         showMatch(str, @"\bm\S*e\b");
         Console.ReadKey();
      }
   }
}
```

Anonymous Methods

- **Anonymous methods** provide a technique to pass a code block as a delegate parameter. Anonymous methods are the methods without a name, just the body.
- You need not specify the return type in an anonymous method; it is inferred from the return statement inside the method body.

Writing an Anonymous Method

- Anonymous methods are declared with the creation of the delegate instance, with a **delegate** keyword.
- For example:

```
delegate void NumberChanger(int n);
...
NumberChanger nc = delegate(int x)
{
   Console.WriteLine("Anonymous Method: {0}", x);
};
```

- The code block Console.WriteLine("Anonymous Method: {0}", x); is the body of the anonymous method.
- The delegate could be called both with anonymous methods as well as named methods in the same way, i.e., by passing the method parameters to the delegate object.

Anonymous Method Example

```
using System;
delegate void NumberChanger(int n);
namespace AnonymousMethodDemo {
   class Program {
      static int num = 10;
      public static void AddNum(int p) {
         num += p;
         Console.WriteLine("Named Method: {0}", num);
      }
      public static void MultNum(int q) {
         num *= q;
         Console.WriteLine("Named Method: {0}", num);
      }
      public static int getNum() {
         return num;
      }
      static void Main(string[] args) {
         // create delegate instances using anonymous method
         NumberChanger nc = delegate(int x)
         {
            Console.WriteLine("Anonymous Method: {0}", x);
         };
         // calling the delegate using the anonymous method
         nc(10);
         // instantiating the delegate using the named methods
         nc =  new NumberChanger(AddNum);
         // calling the delegate using the named methods
         nc(5);
         // instantiating the delegate using another named methods
         nc =  new NumberChanger(MultNum);
         // calling the delegate using the named methods
         nc(2);
         Console.ReadKey();
      }
   }
}
```

Unsafe Codes

- C# allows using pointer variables in a function of code block when it is marked by the **unsafe** modifier
- The **unsafe code** or the unmanaged code is a code block that uses a **pointer** variable

Pointers

- A **pointer** is a variable whose value is the address of another variable i.e., the direct address of the memory location.
- Similar to any variable or constant, you must declare a pointer before you can use it to store any variable address.
- Syntax to declare a pointer:

```
type *var-name;
```

- Example:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

Pointers Example

```
using System;
namespace UnsafeCodeDemo
{
   class Program
   {
      static unsafe void Main(string[] args)
      {
         int var = 20;
         int* p = &var;
         Console.WriteLine("Data is: {0} ",  var);
         Console.WriteLine("Address is: {0}",  (int)p);
         Console.ReadKey();
      }
   }
}=
```

Retrieving Data Value Using a Pointer

You can retrieve the data stored at the located referenced by the pointer variable, using the ToString() method:

```
using System;
namespace UnsafeCodeDemo
{
   class Program
   {
      public static void Main()
      {
         unsafe
         {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} " , var);
            Console.WriteLine("Data is: {0} " , p->ToString());
            Console.WriteLine("Address is: {0} " , (int)p);
         }
         Console.ReadKey();
      }
   }
}
```

Passing Pointers as Parameters to Methods

You can pass a pointer variable to a method as parameter:

```
using System;
namespace UnsafeCodeApplication {
   class TestPointer {
      public unsafe void swap(int* p, int *q)
      {
         int temp = *p;
         *p = *q;
         *q = temp;
      }
      public unsafe static void Main()
      {
         TestPointer p = new TestPointer();
         int var1 = 10;
         int var2 = 20;
         int* x = &var1;
         int* y = &var2;
         Console.WriteLine("Before Swap: var1:{0}, var2: {1}", var1, var2);
         p.swap(x, y);
         Console.WriteLine("After Swap: var1:{0}, var2: {1}", var1, var2);
         Console.ReadKey();
      }
   }
}
```

Accessing Array Elements Using a Pointer

- In C#, an array name and a pointer to a data type same as the array data, are not the same variable type. For example, int *p and int[] p, are not same type.
- You can increment the pointer variable p because it is not fixed in memory but an array address is fixed in memory, and you can't increment that.
- If you need to access an array data using a pointer variable, as we traditionally do in C, or C++, you need to fix the pointer using the **fixed** keyword.

Accessing Array Elements Using a Pointer

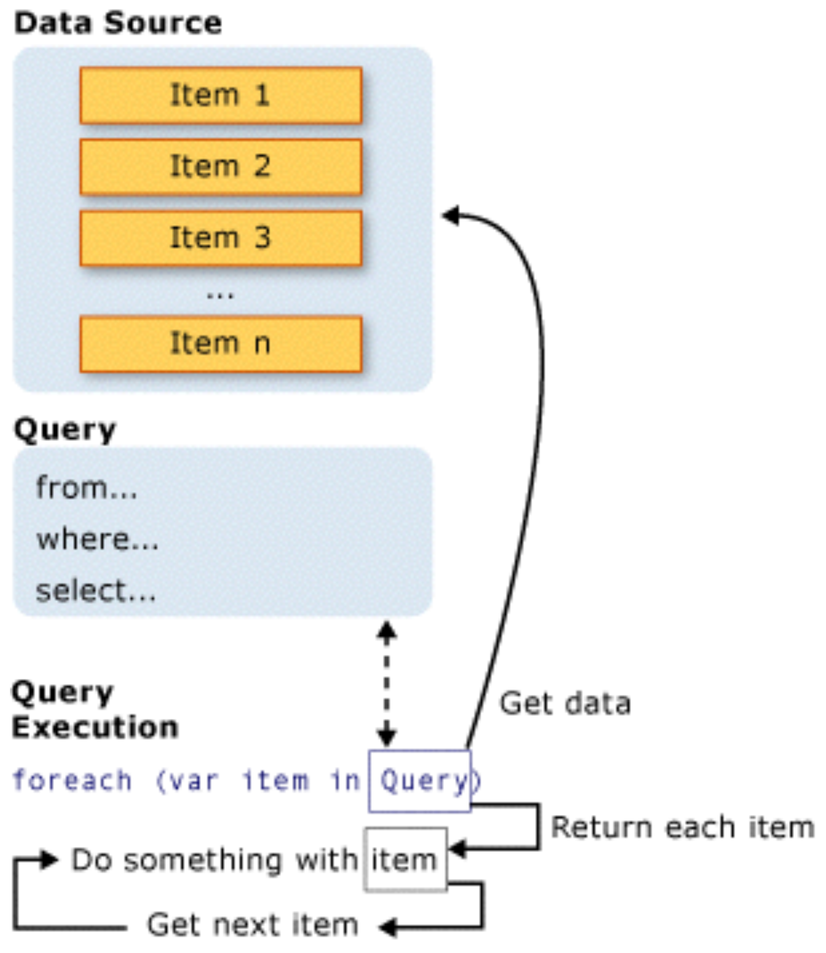
```
using System;

namespace UnsafeCodeDemo
{
   class Program
   {
      public unsafe static void Main()
      {
         int[]  list = {10, 100, 200};
         fixed(int *ptr = list)

         /* let us have array address in pointer */
         for ( int i = 0; i < 3; i++)
         {
            Console.WriteLine("Address of list[{0}]={1}",i,(int)(ptr + i));
            Console.WriteLine("Value of list[{0}]={1}", i, *(ptr + i));
         }

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

Introduction to LINQ Queries

- A query is an expression that retrieves data from a data source. Queries are usually expressed in a specialized query language.
- Different languages have been developed over time for the various types of data sources, for example SQL for relational databases and XQuery for XML. Therefore, developers have had to learn a new query language for each type of data source or data format that they must support.
- LINQ simplifies this situation by offering a consistent model for working with data across various kinds of data sources and formats

Three Parts of a Query Operation

- All LINQ query operations consist of three distinct actions:
- Obtain the data source
- Create the query
- Execute the query



LINQ Example

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        //  1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        // 3. Query execution.
        foreach (int num in numQuery)
        {
            Console.Write("{0,1} ", num);
        }
    }
}
```

Query Execution

- **Deferred Execution:** As stated previously, the query variable itself only stores the query commands. The actual execution of the query is deferred until you iterate over the query variable in a `foreach` statement

```
//  Query execution.
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

Query Execution

- **Forcing Immediate Execution:** Queries that perform aggregation functions over a range of source elements must first iterate over those elements
- The following query returns a count of the even numbers in the source array:

```
var evenNumQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

int evenNumCount = evenNumQuery.Count();
```

Query Execution

- To force immediate execution of any query and cache its results, you can call the ToList or ToArray methods:

```
List<int> numQuery2 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToList();

// or like this:
// numQuery3 is still an int[]

var numQuery3 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToArray();
```

Summary

- C# provides a special data types, the **nullable** types, to which you can assign normal range of values as well as null values
- A **regular expression** is a pattern that could be matched against an input text
- Anonymous methods are declared with the creation of the delegate instance, with a **delegate** keyword.
- LINQ simplifies this situation by offering a consistent model for working with data across various kinds of data sources and formats
- C# allows using pointer variables in a function of code block when it is marked by the **unsafe** modifier