

31927 32998: Application Development with .NET

Week-5 Lecture

Programming in C#

Part-4



Outline

- ArrayList
- Foreach loop
- Exception Handling
- Testing basics
- NUnit
- Visual Studio Debugger



ArrayList

- An alternative to Arrays
- Belongs to the **System.Collections** Namespaces
- *ArrayList* is a non-generic type of collection in C#
- It is similar to Arrays, but it **grows Automatically** as items are added!
- Don't need to specify the size of *ArrayList*
- It allows to add and remove at a specific position
- It can hold **heterogenous** type of objects!



ArrayList

Syntax to initialize ArrayList:

```
using System.Collections ;
```

```
ArrayList <ArrayList_Name> = new ArrayList();
```

Example:

```
ArrayList students = new ArrayList();
```



ArrayList

- Important Properties:

Property	Description
Capacity	Gets/Sets the number of elements in the ArrayList
Count	Gets the number of elements actually contain in the ArrayList
IsFixedSize	Check if the ArrayList of fixed size
IsReadOnly	Check whether the ArrayList is read-only
Item	Gets/Sets the element at a specified position/index



ArrayList

- Common Methods:

Method	Description
Add()	Add a single element to the end of an ArrayList
Insert()	Inserts a single element at a specific location/index in the ArrayList
Remove()	Removes a first occurrence of the specified element from an ArrayList
RemoveAt()	Removes an element at a specified position/index from ArrayList
Sort()	Sorts the entire ArrayList
Contains()	Check if a specified element is present in the ArrayList or not, returns true/false, bool.



ArrayList

- Common Methods:

Method	Description
Clear()	Remove all elements from an ArrayList
Clone()	Create a Shallow copy of the ArrayList
BinarySearch()	Searches the entire sorted ArrayList for an element. The ArrayList should be Sorted before using this search method
Reverse()	Reverses the order to the elements in the ArrayList
IndexOf()	Returns the index (int) of the first occurrence of a specified element.



ArrayList

- Example

Output:

```
Arraylist Contents:0 1 2 3
Number of Elements: 4
Capacity : 4
Arraylist Contents after reverse:3 2 1 0
```

```
using System;
using System.Collections;
// ArrayList demo
namespace Week5ClassProgram
{
    0 references
    class ArrayListDemo
    {
        0 references
        static void Main(string[] args)
        {
            //Create an ArrayList
            ArrayList myArrayList = new ArrayList();
            // Add data into the arraylist
            for(int loopVar =0; loopVar<4; loopVar++)...
            // Display the contents of the arraylist
            Console.Write("Arraylist Contents:");
            for (int loopVar = 0; loopVar < 4; loopVar++)...
            Console.WriteLine("");
            // Count the number of elements.
            Console.WriteLine("Number of Elements: {0}", myArrayList.Count);
            // Find the capacity.
            Console.WriteLine("Capacity : {0}", myArrayList.Capacity);
            // Reverse the order of elements
            myArrayList.Reverse();
            // Display the contents of the arraylist
            Console.Write("Arraylist Contents after reverse:");
            for (int loopVar = 0; loopVar < 4; loopVar++)
            {
                Console.Write(myArrayList[loopVar] + " ");
            }
            Console.ReadKey();
        }
    }
}
```



foreach in Loop

- *foreach* loop executes a block of statements for each element in an instance of Type [System.Collections.IEnumerable](#), [System.Collections.Generic.IEnumerable<T>](#) interface
- It can be used to iterate through the elements of an array as well.
- For 1D arrays, foreach processes each elements from index 0 to Length -1
- Can be used for multi-dimensional, but nested **for** loop are preferred as it give better control.
- Explicit use of indexes is not required.
- Continue, break can be used to control the loop.



foreach in Loop

- Example:

Output:

```
Elements in the arrayList are:  
10 11 12 13 14
```

```
using System;  
using System.Collections;  
  
//Foreach demo  
  
namespace Week5ClassProgram  
{  
    0 references  
    class ForeachDemo  
    {  
        0 references  
        static void Main(string[] args)  
        {  
            // Create an ArrayList  
            ArrayList myArraylist = new ArrayList();  
  
            // Add data into the arrayList  
            for (int loopVar = 10; loopVar<15; loopVar++)  
            {  
                myArraylist.Add(loopVar);  
            }  
  
            // Display the elements in the ArrayList  
            Console.WriteLine("Elements in the arrayList are:");  
            foreach (int element in myArraylist)  
            {  
                Console.Write(element + " ");  
            }  
  
            Console.ReadKey();  
        }  
    }  
}
```



Exception Handling

- Exceptions: error occurring during the execution of a C# program, run-time errors
- Unexpected behaviour or an error conditions encountered while executing program.
- Errors can be detected and handled using the exception handling, a built in mechanism in the .Net Framework
- Exceptions are derived from `System.Exceptions`
- Exception Handling uses `try`, `catch`, and `finally` keywords



Exception Handling

- **Try** : try actions that may not succeed (e.g. based on certain user actions, or other situations)
- **catch**: handle the failure when is it reasonable to do so
- **finally** : Clean up resources (e.g. close open files/streams, closes database connections , etc.)
- Exceptions can be generated by the CLR, the .Net Framework or by any third party libraries.
- Exceptions are created by using the **throw** keyword
- There can be multiple **catch** blocks



Exception Handling

- Syntax:

```
try{  
    // Statements which can generate an exception  
}  
catch(<Exception_type> <variable_name>){  
    // Statements to handle exceptions  
}  
finally{  
    // (optional) code to clean up resources.  
}
```



Exception Handling

- Common Exception:

Exception type	Description
System.IO.Exception	Handles IO errors
System.DivideByZeroException	Handles errors generated due to dividing a dividend with zero
System. IndexOutOfRangeException	Thrown by runtime when an array is indexed improperly, such as referring to array index which is out of range
System. NullReferenceException	Thrown by the runtime when a null object is referenced.



Exception Handling

- Common Exception:

Exception type	Description
<code>System.ArgumentOutOfRangeException</code>	Thrown by methods that verify that arguments are in a given range.
<code>System.ArgumentNullException</code>	Thrown by methods that do not allow an argument to be null.
<code>FileNotFoundException</code>	Thrown when an attempt to access a file that does not exist on disk fails.



Exception Handling

- Exception examples:

```
myArray[myArray.Length+1]
```

IndexOutOfRangeException

```
String str = null;  
"Calculate".IndexOf(str);
```

ArgumentNullException

```
String str = "Hello";  
str.SubString(str.Length + 1);
```

ArgumentOutOfRangeException



Exception Handling

- Examples:

Output 1:

```
Enter the first number:1
Enter the second number:0
The Error is 'System.DivideByZeroException: Attempted to divide by zero.
   at Week5ClassProgram.ExceptionHandlingDemo.Main(String[] args) in H:\UTS\D Drive\Teaching\
Spring 2018\31927 Net programming\Lectures-pptx\Week-5\Week5-ExamplePrograms-in-Lecture\Excep
tionHandlingDemo\ExceptionHandlingDemo\Program.cs:line 25'
Bye bye... I cleaned the memmory!
```

Output 2:

```
Enter the first number:5
Enter the second number:10
Bye bye... I cleaned the memmory!
```

```
using System;
// Exception Handling demo
// Create a list of students and write them into
// an exixsting file
namespace Week5ClassProgram
{
    0 references
    class ExceptionHandlingDemo
    {
        0 references
        static void Main(string[] args)
        {
            int number1, number2;
            double result;
            // Accept two number from user
            Console.WriteLine("Enter the first number:");
            string userInput = Console.ReadLine();
            number1 = Convert.ToInt32(userInput);

            Console.WriteLine("Enter the second number:");
            userInput = Console.ReadLine();
            number2 = Convert.ToInt32(userInput);
            // dividing two numbers might have a division by zero error
            // hence run it inside a try block
            try
            {
                result = number1 / number2;
            }
            // If number2 is zero as exception will be created
            catch(DivideByZeroException e)
            {
                Console.WriteLine("The Error is '{0}'", e);
            }
            // Always executed
            finally
            {
                Console.WriteLine("Bye bye... I cleaned the memmory!");
            }
            Console.ReadKey();
        }
    }
}
```



Throwing exceptions


- **throw** keyword is used to throw an exception

Example 1

```
try
{
    if (number2 == 0)
    {
        throw new DivideByZeroException();
    }
    else
    {
        result = number1 / number2;
        Console.WriteLine("Result is: {0}", result);
    }
}
```

Example 2

```
try
{
    if (number2 == 0)
    {
        throw new Exception("My Division by Zero exception");
    }
    else
    {
        result = number1 / number2;
        Console.WriteLine("Result is: {0}", result);
    }
}
// If number2 is zero as exception will be created
catch(DivideByZeroException e)
{
    Console.WriteLine("The Error is '{0}'", e);
}
catch(Exception e)
{
    Console.WriteLine("The Error is '{0}'", e);
}
// Always executed
finally
{
    Console.WriteLine("Bye bye... I cleaned the memory");
}
```



Testing Basics

Testing: Problem 1

- *Testing cannot be exhaustive*
- By “exhaustive”, we mean it is impossible to test a program for every possible combination of inputs

Testing: Problem 2

- *Testing cannot prove the **absence** of faults*
- Since we cannot test all cases, we may miss the cases that demonstrate particular faults.
- So, we cannot promise to continue testing until all faults are found and eliminated.
- Instead, people usually test until cost per discovered fault becomes too high (or time runs out).

Conclusion

- All software has bugs. Better software has less bugs.



Testing Basics

Testing Terminology:

- ***White box testing***: tests based on the logic of the program
- ***Black box testing***: uses “random” test cases
 - Genuinely random, OR
 - Weighted to match expected usage patterns (better)
- ***Regression testing***: running all the old test cases after a program is changed (to check for introduced bugs)
- ***Path coverage***: measuring the fraction of statements in the program tested to date.
- ***Weighted test set***: tests chosen according to expected frequency of use.



Test Driven Development

- First articulated by Kent Beck – “Test-Driven Development”
- Used in Agile programming, especially Extreme Programming.
- Based upon the following paradigm
 1. Write a test
 2. Write code
 3. Make it run
 4. Refactor to remove duplication
- The concept is applied incrementally to very small pieces of code.
- Every time we find a new bug we create unit tests for it as part of removing the bug.
- All tests are then run again (regression testing) to ensure the bug fix has not added a new bug somewhere else



Refactoring

- Making many incremental changes to code to improve it's overall design.
- Martin Fowler – “Refactoring : Improving the design of Existing Code”.
- Each change is small to avoid breaking the program but the cumulative effect can end up making significant improvements to the entire program



Unit Testing

- **Objective:** Isolate a small unit of code and validate its correctness and reliability
- Testing individual small parts called Unit.
- Test cases are written in form of functions
- Functions evaluate whether a returned value is equal the expected value.



MSTest and NUnit

- <http://www.nunit.org>
- Do complete install to get all NUnit tests
- Reference on NUnit testing:
- <https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-nunit>
- Reference for MSTest with a complete example:
<https://www.c-sharpcorner.com/article/a-basic-introduction-of-unit-test-for-beginners/>



Debugger - Breakpoints and Continue

Setting Breakpoints

- Set on code, not white space or comments
- Pauses running program at that point in code
- Left click column on left of code

Run To Cursor

- Place cursor on line of code
- Right click and select “run to cursor”

DataTips : While program paused, place mouse over data in code to see contents

Continue : Select Debug Continue or F5

Removing Breakpoint: Right click breakpoint and select delete option or left click breakpoint

