|  |
| --- |
| Day-20 Morning Assignment  By  U.Joshna  [18-2-2022] |

|  |
| --- |
| 1. Research and understand scope of variables in C# |
| * The part of the program where a particular variable is accessible is termed as the Scope of that variable. A variable can be defined in a class, method, loop etc. In C/C++, all identifiers are lexically (or statically) scoped, i.e., scope of a variable can be determined at compile time and independent of the function call stack. But the C# programs are organized in the form of classes.   Mainly there are three types of variables: |
| * Class Level Scope * Method Level Scope * Block Level Scope |
| Class Level Variable Scope: |
| In C#, when we declare a variable inside a class, the variable can be accessed within the class. This is known as **class level variable scope**.  Class level variables are known as fields and they are declared outside of methods, constructors, and blocks of the class. For example, |
| Code: |
| using System;  namespace VariableScope1  {  class Program  {  // class level variable  string str = "Class Level";    public void display()  {  Console.WriteLine(str);  }    static void Main(string[] args)  {  Program ps = new Program();  ps.display();    Console.ReadLine();    }  }  } |
| Method Level Variable Scope:  |  | | --- | | Code: | | using System;  namespace VariableScope1  {  class Program  {    public void display()  {  string str = "inside method";    // accessing method level variable  Console.WriteLine(str);  }    static void Main(string[] args)  {  Program ps = new Program();  ps.display();    Console.ReadLine();  }  }  } | | Block Level Variable Scope: | | Code: | | using System;  namespace VariableScope1  {  class Program  {  public void display()  {  int j;  for ( j = 1; j <= 3; j++)  {    }  Console.WriteLine(j);  }    static void Main(string[] args)  {  Program ps = new Program();  ps.display();    Console.ReadLine();  }  }  } | |

|  |
| --- |
| 2.What are Delegates in C#? Write the points discussed in the class & illustrate with a C# Code Example. |
| Delegates in C#:  • A delegate is a type that represents references to methods with a particular parameter list and return type.  • Delegates are used to pass methods as arguments to other methods.  Declaration of Delegates:  Delegate type can be declared using the delegate keyword. Once a delegate is declared, delegate instance will refer and call those methods whose return type and parameter-list matches with the delegate declaration.  Syntax:  [modifier] delegate [return\_type] [delegate\_name([parameter\_list]);  • A Delegate is like a function pointer.  • Using delegates, we can call (or) point to one or more methods.  • When declaring a delegate’s return type & parameters must match with the methods you want to point, using delegates.  • Benefit of delegate is that, using single call from delegate all your methods pointing to delegate will be called.  • There are Two types of Delegates in C#, they are : Single cast delegate Multi cast delegate |
| Code: |
| using System;  namespace Day20Project2  {  public delegate void MyCaller(int a, int b);  internal class Program  {  public static void Add(int a, int b)  {  Console.WriteLine(a + b);  }  public static void Mul(int a, int b)  {  Console.WriteLine(a \* b);  }  public static void Div(int a, int b)  {  Console.WriteLine(a / b);  }    static void Main(string[] args)  {    MyCaller mc = new MyCaller(Add);  mc += Mul;  mc += Div;    mc(7, 8);    mc(14, 15);    mc(23, 24);    Console.ReadLine();  }    }  } |
| Output: |
|  |
|  |
|  |

|  |
| --- |
| 3. What are nullable types in C# WACP to illustrate nullable types Write some properties of nullable types (like HasValue) |
| * The Nullable type allows you to assign a null value to a variable. * Only for Reference types, we can use Nullable type. We can’t use nullable for Value Types. In order to declare a variable as a Nullable type, we place ”?” symbol, adjacent to its data type.   Points to Remember : -   * Nullable type allows assignment of null to value types. * ? operator is a shorthand syntax for Nullable types. * Use value property to get the value of nullable type. - * Use HasValue property to check whether value is assigned to nullable type or not.   Static Nullable class is a helper class to compare nullable types. |
| Code: |
| using System;  namespace Day20Project3  {  internal class Program  {  static void Main(string[] args)  {  int? firstValue = 40;  int? secondValue = null;  int? result;  result = (firstValue.HasValue) ? firstValue : null;  Console.WriteLine("The value inside First Value Nullable type is : {0}",  result);  result = (secondValue.HasValue) ? secondValue : null;  Console.WriteLine("The value inside Second Value Nullable type is : {0}",  result);  Console.ReadKey();    }  }  } |
| Output: |
|  |
|  |
|  |

|  |
| --- |
| 4. out, ref – parameters  please research on these two types of parameters  write a C# program to illustrate the same. |
| Code: |
| C# Ref & Out Keywords:  Ref and out keywords in C# are used to pass arguments within a method or function. Both indicate that an argument/parameter is passed by reference. By default parameters are passed to a method by value. By using these keywords (ref and out) we can pass a parameter by reference  Ref Keyword:  The ref keyword passes arguments by reference. It means any changes made to this argument in the method will be reflected in that variable when control returns to the calling method.  Out Keyword:  The out keyword passes arguments by reference. This is very similar to the ref keyword |
| Ref Vs Out: |
| |  |  | | --- | --- | | Ref | Out | | The parameter or argument must be initialized first before it is passed to ref. | It is not compulsory to initialize a parameter or argument before it is passed to an out. | | It is not required to assign or initialize the value of a parameter (which is passed by ref) before returning to the calling method. | A called method is required to assign or initialize a value of a parameter (which is passed to an out) before returning to the calling method | | Passing a parameter value by Ref is useful when the called method is also needed to modify the pass parameter. | Declaring a parameter to an out method is useful when multiple values need to be returned from a function or method. | | It is not compulsory to initialize a parameter value before using it in a calling method | A parameter value must be initialized within the calling method before its use. | | When we use REF, data can be passed bidirectionally. | When we use OUT data is passed only in a unidirectional way (from the called method to the caller method). |  |  | | --- | | Both ref and out are treated differently at run time and they are treated the same at compile time. |  |  | | --- | | Properties are not variables; therefore, it cannot be passed as an out or ref parameter. |   Code: |
| using System;  namespace Day20Project4  {  internal class Program  {  public static string NextNameByRef(ref int id)  {  string returnText = "Next-" + id.ToString(); id += 1;  return returnText;  }    public static string NextNameByOut(out int id)  {  id = 1;  string returnText = "Next-" + id.ToString(); return returnText;  }      static void Main(string[] args)  {  Console.WriteLine("\n -----------\*\*\*\* Ref Keyword Output \*\*\*\* ");    int i = 1;  Console.WriteLine("Previous value of integer i : " +  i.ToString()); string testRef = NextNameByRef(ref i);  Console.WriteLine("Current value of integer i : " + i.ToString()); Console.WriteLine("\n -----------\*\*\*\* Out Keyword Output \*\*\*\* ");  int j;  string testOut = NextNameByOut(out j); Console.WriteLine("Current value of integer j:" + j.ToString());    Console.ReadKey();    }  }  } |
| Output: |
|  |