**13-Nov-2023**

1. **What is Namespace?**

A namespace is used to organize the code and it as a collection of classes, delegates and namespaces etc.

1. **What is console?**

🡺Console provides some methods for input and output operations in console.

🡺Console.WriteLine is used to display text on the console.

🡺Console.ReadLine is used to read the text from user input data.

🡺Console is present inside the System namespace.

1. **2 Ways to write to Console…**

**🡺** Concatenation.

**🡺** Place holder Syntax.

using System;

class Program

{

static void Main(string[] args)

{

Console.WriteLine(“Please enter your name”);

string myName= Console.ReadLine();

Console.WriteLine(“Hello” + myName); -----> Concatenation

Console.WriteLine(“Hello {0}”, myName); ----> Place Holder

}

}

1. **Built-in or Primitive or Value data types in C#**

* **Boolean type (True or False):** A Boolean data type is declared with the bool keyword and can only take the values true or false.
* **Integer type (int, byte, short, long):** The int data type is the preferred data type when we create variables with a numeric value.
* **Float type (float):** You should use a floating-point type whenever you need a number with a decimal, such as 9.99 or 3.14515. The precision of float is only six or seven decimal digits.
* **Double:** Double variables have a precision of about 15 digits. Therefore it is safer to use double for most calculations.
* **Character type (Char):** The char data type is used to store a **single** character. The character must be surrounded by single quotes, like 'A' or 'c'.
* **Decimal type**

1. **Abstract or Reference types in C#**

* Interface.
* Class.
* Delegates.
* Arrays.

1. **Operators in C#**

* Arithmetic Operators. (+, -, \*, %, /)
* Assignment Operator. (=)
* Conditional Operators. (&&[All], ||[Any One])
* Comparison Operators. (<, >, <=, >=, ==, !=)
* Ternary Operators. (?:)

int Number=10;

bool Number10 = Number == 10? true : false

1. **Reference Types** can have null values whereas **Value Types** cannot have null values.

By default, Value types are non-nullable. To make them nullable use question mark beside the data type.

**int? i=null;**

1. **Type Casting:**

When the variable of one data type is changed to another data type is known as the Type Casting. (or) Type casting is when you assign a value of one data type to another type.

In C#, there are two types of casting:

* **Implicit Casting** (automatically) - converting a smaller type to a larger type size  
  char -> int -> long -> float -> double
* **Explicit Casting** (manually) - converting a larger type to a smaller size type  
  double -> float -> long -> int -> char

**Implicit Casting**

Implicit casting is done automatically when passing a smaller size type to a larger size type.

int myInt = 9;

double myDouble = myInt; // Automatic casting: int to double

Console.WriteLine(myInt); // Outputs 9

Console.WriteLine(myDouble); // Outputs 9

## Explicit Casting

Explicit casting must be done manually by placing the type in parentheses in front of the value:

double myDouble = 9.78;

int myInt = (int) myDouble; // Manual casting: double to int

Console.WriteLine(myDouble); // Outputs 9.78

Console.WriteLine(myInt); // Outputs 9

## Type Conversion Methods

It is also possible to convert data types explicitly by using built-in methods, such as Convert.ToBoolean, Convert.ToDouble, Convert.ToString, Convert.ToInt32 (int) and Convert.ToInt64 (long).

int myInt = 10;

double myDouble = 5.25;

bool myBool = true;

Console.WriteLine(Convert.ToString(myInt)); // convert int to string

Console.WriteLine(Convert.ToDouble(myInt)); // convert int to double

Console.WriteLine(Convert.ToInt32(myDouble)); // convert double to int

Console.WriteLine(Convert.ToString(myBool)); // convert bool to string

1. **Math Functions:**

Console.WriteLine(Math.Max(5, 10));

Console.WriteLine(Math.Min(12,26));

Console.WriteLine(Math.Round(12.26));

Console.WriteLine(Math.Sqrt(64));

Console.WriteLine(Math.Abs(-26));

**14-Nov-2023**

**Control Statements:**

In **C#**, a **control statement** is a programming construct that allows you to control the flow of execution in your code based on certain conditions.

There are different types of Control statements in C#. They are:

1. If-else
2. Switch statements
3. Loops
4. Break
5. Continue
6. Goto statement (Jump statement)

# **C# if-else**

In C# programming, the if statement is used to test the condition. There are various types of if statements in C#.

* if statement
* if-else statement
* nested if statement
* if-else-if ladder

**if Statement:** The C# if statement tests the condition. It is executed if the condition is true.

using System;

**public** **class** IfExample

    {

**public** **static** **void** Main(string[] args)

        {

**int** num = 10;

**if** (num % 2 == 0)

            {

                Console.WriteLine("It is even number");

            }

        }

    }

## if-else Statement

The C# if-else statement also tests the condition. It executes the if block if condition is true otherwise else block is executed.

using System;

**public** **class** IfExample

    {

**public** **static** **void** Main(string[] args)

        {

**int** num = 11;

**if** (num % 2 == 0)

            {

                Console.WriteLine("It is even number");

            }

**else**

            {

                Console.WriteLine("It is odd number");

            }

        }

    }

## if-else-if ladder Statement

The C# if-else-if ladder statement executes one condition from multiple statements.

using System;

**public** **class** IfExample

    {

**public** **static** **void** Main(string[] args)

        {

            Console.WriteLine("Enter a number to check grade:");

**int** num = Convert.ToInt32(Console.ReadLine());

**if** (num <0 || num >100)

            {

                Console.WriteLine("wrong number");

            }

**else** **if**(num >= 0 && num < 50){

                Console.WriteLine("Fail");

            }

**else** **if** (num >= 50 && num < 60)

            {

                Console.WriteLine("D Grade");

            }

**else** **if** (num >= 60 && num < 70)

            {

                Console.WriteLine("C Grade");

            }

**else** **if** (num >= 70 && num < 80)

            {

                Console.WriteLine("B Grade");

            }

**else** **if** (num >= 80 && num < 90)

            {

                Console.WriteLine("A Grade");

            }

**else** **if** (num >= 90 && num <= 100)

            {

                Console.WriteLine("A+ Grade");

            }

        }

    }

**Switch Statements:**

The C# switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement.

**Syntax:**

**switch**(expression){

**case** value1:

 //code to be executed;

**break**;

**case** value2:

 //code to be executed;

**break**;

......

**default**:

 //code to be executed if all cases are not matched;

**break**;

}

**Example:**

using System;

**public** **class** SwitchExample

    {

**public** **static** **void** Main(string[] args)

      {

          Console.WriteLine("Enter a number:");

**int** num = Convert.ToInt32(Console.ReadLine());

**switch** (num)

          {

**case** 10: Console.WriteLine("It is 10"); **break**;

**case** 20: Console.WriteLine("It is 20"); **break**;

**case** 30: Console.WriteLine("It is 30"); **break**;

**default**: Console.WriteLine("Not 10, 20 or 30"); **break**;

          } } }

**Loops:**

1. for loop
2. while loop
3. do-while loop

**for loop:**

The C# for loop is used to iterate a part of the program several times. If the number of iterations is fixed, it is recommended to use for loop than while or do-while loops. Here we can initialize variables, check condition and increment/decrement value.

**Syntax:**

**for**(initialization; condition; increment/decrement){

//code to be executed

}

**Example:**

using System;

**public** **class** ForExample

    {

**public** **static** **void** Main(string[] args)

      {

**for**(**int** i=1;i<=10;i++){

            Console.WriteLine(i);

          }

      }

    }

## Nested For Loop

In C#, we can use for loop inside another for loop, it is known as nested for loop. The inner loop is executed fully when outer loop is executed one time.

So, if the outer loop and inner loop is executed 3 times, inner loop will be executed 3 times for each outer loop i.e., total 9 times.

**Output**

using System;   1 1

**public** **class** ForExample   1 2

    {   1 3

**public** **static** **void** Main(string[] args)   2 1

      {   2 2

**for**(**int** i=1;i<=3;i++){     2 3

**for**(**int** j=1;j<=3;j++){     3 1

                    Console.WriteLine(i+" "+j);     3 2

                }     3 3

        }

      }

    }

## Infinite For Loop

If we use double semicolon in for loop, it will be executed infinite times. And we need to stop the loop using Ctrl+c

using System;

**public** **class** ForExample

    {

**public** **static** **void** Main(string[] args)

      {

**for** (; ;)

          {

                  Console.WriteLine("Infinitive For Loop");

          }

      }

 }

# **While Loop:**

In C#, while loop is used to iterate a part of the program several times. If the number of iterations is not fixed, it is recommended to use while loop than for loop.

using System;

**public** **class** WhileExample

    {

**public** **static** **void** Main(string[] args)

      {

**int** i=1;

**while**(i<=10)

          {

              Console.WriteLine(i);

              i++;

          }

     }

   }

### **Nested While Loop:**

In C#, we can use while loop inside another while loop, it is known as nested while loop. The nested while loop is executed fully when outer loop is executed once.

using System;

**public** **class** WhileExample

    {

**public** **static** **void** Main(string[] args)

      {

**int** i=1;

**while**(i<=3)

          {

**int** j = 1;

**while** (j <= 3)

              {

                  Console.WriteLine(i+" "+j);

                  j++;

              }

              i++;

          }  } }

### **Infinitive While Loop**

We can also create infinite while loop by **passing true** as the test condition.

using System;

**public** **class** WhileExample

    {

**public** **static** **void** Main(string[] args)

      {

**while**(**true**)

          {

                  Console.WriteLine("Infinitive While Loop");

          }

      }

    }

# **Do-While Loop**

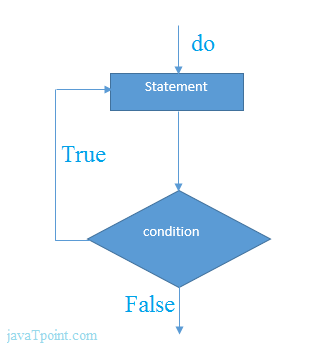
The C# do-while loop is used to iterate a part of the program several times. If the number of iterations is not fixed and you must execute the loop at least once, it is recommended to use do-while loop. The C# do-while loop is executed at least once because condition is checked after loop body.

**Syntax:**

**do**{

//code to be executed

}**while**(condition);



**using** System;

**public** **class** DoWhileExample

    {

**public** **static** **void** Main(**string**[] args)

      {

**int** i = 1;

**do**{

              Console.WriteLine(i);

              i++;

          } **while** (i <= 10) ;

     }

   }

## Nested do-while Loop

In C#, if you use a do-while loop inside another do-while loop, it is known as nested do-while loop. The nested do-while loop is executed fully for each outer do-while loop.

**using** System;

**public** **class** DoWhileExample

    {

**public** **static** **void** Main(**string**[] args)

      {

**int** i=1;

**do**{

**int** j = 1;

**do**{

                  Console.WriteLine(i+" "+j);

                  j++;

              } **while** (j <= 3) ;

              i++;

          } **while** (i <= 3) ;

     }

   }

## Infinitive do-while Loop

In C#, if you pass **true** in the do-while loop, it will be infinitive do-while loop.

**using** System;

**public** **class** WhileExample

    {

**public** **static** **void** Main(**string**[] args)

      {

**do**{

              Console.WriteLine("Infinitive do-while Loop");

          } **while**(**true**);

      }

    }

# **Break Statement**

The C# break is used to break loop or switch statement. It breaks the current flow of the program at the given condition. In the case of the inner loop, it breaks only inner loop.

**Syntax:**

jump-statement;

**break**;

**Example:**

**for** (**int** i = 1; i <= 10; i++)

          {

**if** (i == 5)

              {

**break**;

              }

              Console.WriteLine(i);

          }

### **Break Statement with Inner Loop**

The C# break statement breaks the inner loop only if you use break statement inside the inner loop.

**public** **static** **void** Main(**string**[] args)

      {

**for**(**int** i=1;i<=3;i++){

**for**(**int** j=1;j<=3;j++){

**if**(i==2&&j==2){

**break**;

                        }

                        Console.WriteLine(i+" "+j);

                    }

          }

      }

# **Continue Statement**

The C# continue statement is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In the case of the inner loop, it continues only inner loop.

**using** System;

**public** **class** ContinueExample

    {

**public** **static** **void** Main(**string**[] args)

      {

**for**(**int** i=1;i<=10;i++){

**if**(i==5){     //Skips the condition 5

**continue**;

            }

            Console.WriteLine(i);

        }

      }

   }

### **Continue Statement with Inner Loop**

C# Continue Statement continues inner loop only if you use continue statement inside the inner loop.

**using** System;

**public** **class** ContinueExample

    {

**public** **static** **void** Main(**string**[] args)

      {

**for**(**int** i=1;i<=3;i++){

**for**(**int** j=1;j<=3;j++){

**if**(i==2&&j==2){

**continue**;

                        }

                        Console.WriteLine(i+" "+j);

                    }

            }

      }

   }

**Goto (Jump) Statement**

🡺The C# goto statement is also known jump statement. It is used to transfer control to the other part of the program. It unconditionally jumps to the specified label.

🡺It can be used to transfer control from deeply nested loop or switch case label.

🡺Currently, it is avoided to use goto statement in C# because it makes the program complex.

using System;

**public** **class** GotoExample

    {

**public** **static** **void** Main(string[] args)

      {

      ineligible:

          Console.WriteLine("You are not eligible to vote!");

      Console.WriteLine("Enter your age:\n");

**int** age = Convert.ToInt32(Console.ReadLine());

**if** (age < 18){

**goto** ineligible;

      }

**else**

      {

              Console.WriteLine("You are eligible to vote!");

      }

      }

   }

**16-Nov-2023**

**C# Keywords:**

**abstract:** Used to declare an abstract class or abstract method. Abstract classes cannot be instantiated and are meant to be subclassed.

**as:** Used for type casting or to perform a safe cast. It is used to convert one type to another safely.

**base:** Refers to the base class in an inheritance hierarchy. It is used to access members of the base class from within a derived class.

**bool:** Represents a Boolean value, either true or false.

**break:** Exits from the innermost loop or switch statement.

**byte:** Represents an 8-bit unsigned integer.

**case:** Used in switch statements to define different cases.

**catch:** Catches exceptions that are thrown in a try block.

**char:** Represents a single 16-bit Unicode character.

**checked:** Enables overflow checking for integral-type arithmetic operations. It ensures that arithmetic overflow is checked at runtime.

**class:** Declares a class. A class is a blueprint for creating objects that encapsulate data and behavior.

**const:** Declares a constant field or a constant local. The value of a const variable cannot be changed.

**continue:** Skips the rest of the loop and continues with the next iteration.

**decimal:** Represents a 128-bit decimal value, suitable for financial calculations.

**default:** Used in switch statements to define the default case.

**delegate:** Declares a reference type that represents a method signature. Delegates are used for defining and referencing methods.

**do:** Initiates a do-while loop. It executes a block of code repeatedly as long as a specified condition is true.

**double:** Represents a double-precision 64-bit floating-point number.

**else:** Defines the block of code to be executed if the condition in an if statement is false.

**enum:** Declares an enumeration, a distinct type with named values.

**event:** Declares an event. Events enable a class or object to notify other classes or objects

when something happens.

**explicit:** Specifies that a user-defined type conversion should be used. It is used to define conversion rules between custom types.

**extern:** Indicates that a method is implemented externally. It is often used with platform invoke services.

**false:** Represents the Boolean value false.

**finally:** Defines a block of code that is executed regardless of whether an exception is thrown. Useful for cleanup operations.

**fixed:** Declares a fixed-size array. Used in contexts where memory should not be relocated.

**float:** Represents a single-precision 32-bit floating-point number.

**for:** Initiates a for loop. It executes a block of code repeatedly for a specified number of times.

**foreach:** Iterates over elements in a collection. It simplifies the iteration process for collections.

**goto:** Transfers control to a labeled statement. Its use is generally discouraged due to the potential for creating unreadable and hard-to-maintain code.

**if:** Tests a condition and executes a block of code if the condition is true.

**implicit:** Specifies that a user-defined type conversion should be used. Similar to explicit, it defines conversion rules between custom types.

**in:** Used in foreach statements to iterate over the elements of a collection.

**int:** Represents a 32-bit signed integer.

**interface:** Declares an interface. Interfaces define a contract for classes, specifying which methods and properties they must implement.

**internal:** Specifies that a class or method is accessible within the same assembly.

**is:** Used for type testing. It checks whether an object is of a specified type.

**lock:** Acquires the mutual-exclusion lock for a given object. Used for synchronization in multithreaded applications.

**long:** Represents a 64-bit signed integer.

**namespace:** Declares a namespace. Namespaces are used to organize code into logical groups.

**new:** Creates an object or initializes a new instance of a class. It is also used to hide a member of the base class.

**null:** Represents a null reference. It indicates that a variable does not refer to any object.

**object:** Represents an object type. It is the base type for all other types in C#.

**operator:** Declares an operator. It is used to define custom operators for user-defined types.

**out:** Used to pass a variable by reference as a method parameter. It indicates that the parameter is intended to be modified by the called method.

**override:** Provides a new implementation of a base class method. It is used in derived classes to override a method from the base class.

**params:** Allows a method to accept a variable number of arguments. It simplifies method calls with a variable number of parameters.

**private:** Specifies that a class member is accessible only within its own class.

**protected:** Specifies that a class member is accessible within its own class and by derived classes.

**public:** Specifies that a class member is accessible from any other class.

**readonly:** Specifies that a field can only be assigned a value at the time of declaration or in the constructor. It creates an immutable field.

**ref:** Indicates that a parameter is passed by reference. It allows a method to modify the value of the parameter directly.

**return:** Exits a method and returns a value to the caller.

**sbyte:** Represents an 8-bit signed integer.

**sealed:** Prevents a class from being inherited. A sealed class cannot be used as a base class.

**short:** Represents a 16-bit signed integer.

**sizeof:** Returns the size, in bytes, of a specified value type.

**stackalloc:** Allocates memory on the stack. It is often used for performance-critical scenarios.

**static:** Declares a static member, which belongs to the type itself rather than to instances of the type. Static members are shared among all instances of a class.

**string:** Represents a sequence of characters.

**struct:** Declares a value type. Structs are similar to classes but are typically used for lightweight objects.

**switch:** Selects one of many code blocks to be executed based on the value of an expression.

**this:** Refers to the current instance of the class. It is often used to disambiguate between instance variables and local variables.

**throw:** Throws an exception. Used to signal that an exceptional condition has occurred.

**true:** Represents the Boolean value true.

**try:** Defines a block of code that may throw an exception. It is followed by catch and optionally finally blocks.

**typeof:** Gets the System.Type of a type. It is often used for reflection.

**uint:** Represents a 32-bit unsigned integer.

**ulong:** Represents a 64-bit unsigned integer.

**unchecked:** Disables overflow checking for integral-type arithmetic operations. It allows arithmetic overflow to wrap around.

**unsafe:** Enables the use of unsafe code blocks. Unsafe code can manipulate pointers and perform other low-level operations.

**ushort:** Represents a 16-bit unsigned integer.

**using:** Declares a namespace or defines the scope of an IDisposable object. It is used for resource management, ensuring that resources are released when they are no longer needed.

**virtual:** Specifies that a method, property, or event can be overridden in a derived class. It is used in base classes to indicate that a method can be customized in derived classes.

**void:** Specifies that a method does not return a value.

**volatile:** Indicates that a field can be modified by multiple threads. It ensures that reads and writes to the field are not optimized or reordered.

**while:** Initiates a while loop. It repeatedly executes a block of code while a specified condition is true.

**where:** Specifies constraints on the generic parameters of a type or method. It is used in generic type and method definitions.

**17-Nov-2023**

**Arrays:**

Let us think that we need some integer or some string values. For that we usually create variables for each value. But that becomes complex. To avoid that complexity, we use arrays.

Arrays are used to store the elements of the same data types in a contiguous memory. Arrays are accessed by using their index numbers. But we can set the fixed set of elements in C# array.

**Basic Syntax of an Array:**

int[] arrname = new int [arrsize] ----> Declaration of an array.

int[] arrname = new int [arrsize] {values in the array according to size provided}----> Declaration and Initialization of an array.

**Other ways of creating arrays:**

string[] cars = new string[4];

// Create an array of four elements and add values right away

string[] cars = new string[4] { "Volvo", "BMW", "Ford", "Mazda" };

// Create an array of four elements without specifying the size

string[] cars = new string[] { "Volvo", "BMW", "Ford", "Mazda" };

// Create an array of four elements, omitting the new keyword, and without specifying the size

string[] cars = { "Volvo", "BMW", "Ford", "Mazda" };

**Arrays are of 3 types:**

🡺Single-Dimensional Arrays.

🡺Multi-Dimensional Arrays.

🡺Jagged Arrays.

**Example of a single dimensional array:**

**using** System;

**public** **class** ArrayExample

{

**public** **static** **void** Main(**string**[] args)

    {

**int**[] arr = **new** **int**[5];//creating array

        arr[0] = 10;//initializing array

        arr[2] = 20;

        arr[4] = 30;

        //traversing array

**for** (**int** i = 0; i < arr.Length; i++)

        {

            Console.WriteLine(arr[i]);

        }  } }

## Loop Through an Array

🡺You can loop through the array elements with the **for** loop, and use the **Length** property to specify how many times the loop should run.

int[] marks = new int[10]; //DECLARATION of an array....

marks[0] = 69;

marks[1] = 35;

marks[2] = 76;

marks[3] = 24;

marks[4] = 26;

marks[5] = 91;

marks[6] = 70;

marks[7] = 53;

marks[8] = 80;

marks[9] = 69;

for (int i=0;i<marks.Length;i++)

{

Console.WriteLine(marks[i]);

}

🡺There is also a **foreach** loop, which is used exclusively to loop through elements in an **array**:

string[] cars = new string[5] { "MG Hector", "Verna", "Baleno", "BMW", "Audi" }; //DECLARATION and INITIALIZATION at the same time....

Console.WriteLine(cars[0]);

Console.WriteLine(cars[4]);

foreach (string car in cars) //prints all the elements of an array....

{

Console.WriteLine(car);

}

Console.WriteLine("============= Single Dimensional Arrays================");

// Create an array of four elements, and add values later

string[] cars = new string[4];

// Create an array of four elements and add values right away

string[] cars = new string[4] { "Volvo", "BMW", "Ford", "Mazda" };

// Create an array of four elements without specifying the size

string[] cars = new string[] { "Volvo", "BMW", "Ford", "Mazda" };

// Create an array of four elements, omitting the new keyword, and without specifying the size

string[] cars = { "Volvo", "BMW", "Ford", "Mazda" };

int[] marks = new int[10]; //DECLARATION of an array....

marks[0] = 69;

marks[1] = 35;

marks[2] = 76;

marks[3] = 24;

marks[4] = 26;

marks[5] = 91;

marks[6] = 70;

marks[7] = 53;

marks[8] = 80;

marks[9] = 69;

Console.WriteLine(marks.Sum());

Console.WriteLine(marks.Min());

Console.WriteLine(marks.Max());

Console.WriteLine(marks.Average());

Console.WriteLine(marks[0]=100);

Console.WriteLine(marks.Sum());

Console.WriteLine(marks.Contains(101));

Console.WriteLine(marks.Contains(100));

Array.Sort(marks);

Array.Reverse(marks);

Console.WriteLine(Array.LastIndexOf(marks, 80));

Console.WriteLine(Array.LastIndexOf(marks, 69));

Console.WriteLine("========================foreach===========================");

for (int i = 0; i < marks.Length; i++)

{

Console.WriteLine(marks[i]);

}

Console.ReadKey();

string[] cars = new string[5] { "MG Hector", "Verna", "Baleno", "BMW", "Audi" }; //DECLARATION and INITIALIZATION at the same time....

Console.WriteLine(cars[0]); //Finding a particular element using index numbers....

Console.WriteLine(cars[4]);

cars[4] = "Benz"; //Assigning a new value to 5th element....

Console.WriteLine(cars.Length); //Finding the length of an array....

Array.Sort(cars); //Sorting the array elements....

Array.Reverse(cars); //Reversing the elements of an array....

Console.WriteLine(Array.IndexOf(cars, "MG Hector")); //Finding the index of particular element....

Console.WriteLine(cars.Contains("MG Hector")); //Checking whether an element is present or not. Returns bool value....

Console.WriteLine(cars.Contains("Swift")); //Returns false bcoz Swift is not present....

Console.WriteLine(Array.Find(cars, name => name.StartsWith("V"))); //Returns elements with starting letter "V"....

foreach (string car in cars) //Prints all the elements of an array....

{

Console.WriteLine(car);

}

Console.ReadKey();

**List of Array Methods:**

1. **concat()**
   * *Description:* Combines two or more arrays.
   * *Example:* **const newArray = array1.concat(array2);**
2. **filter()**
   * *Description:* Creates a new array with elements that pass a certain condition.
   * *Example:* **const filteredArray = array.filter(element => element > 5);**
3. **map()**
   * *Description:* Creates a new array by applying a function to each element in the existing array.
   * *Example:* **const newArray = array.map(element => element \* 2);**
4. **reduce()**
   * *Description:* Reduces the array to a single value by applying a function to each element and accumulating the result.
   * *Example:* **const total = array.reduce((accumulator, currentValue) => accumulator + currentValue, 0);**
5. **forEach()**
   * *Description:* Calls a function for each element in the array.
   * *Example:* **array.forEach(element => console.log(element));**
6. **indexOf()**
   * *Description:* Returns the index of the first occurrence of a specified element in the array.
   * *Example:* **const index = array.indexOf(5);**
7. **includes()**
   * *Description:* Checks if an array includes a certain element, returning a boolean.
   * *Example:* **const isIncluded = array.includes(3);**
8. **slice()**
   * *Description:* Returns a shallow copy of a portion of an array into a new array.
   * *Example:* **const newArray = array.slice(1, 4);**
9. **splice()**
   * *Description:* Changes the contents of an array by removing or replacing existing elements.
   * *Example:* **array.splice(2, 3, 'a', 'b', 'c');**
10. **pop()**
    * *Description:* Removes the last element from the array and returns that element.
    * *Example:* **const removedElement = array.pop();**
11. **push()**
    * *Description:* Adds one or more elements to the end of an array and returns the new length.
    * *Example:* **const newLength = array.push(6, 7);**
12. **shift()**
    * *Description:* Removes the first element from the array and returns that element.
    * *Example:* **const shiftedElement = array.shift();**
13. **unshift()**
    * *Description:* Adds one or more elements to the beginning of an array and returns the new length.
    * *Example:* **const newLength = array.unshift(1, 2);**
14. **join()**
    * *Description:* Joins all elements of an array into a string.
    * *Example:* **const resultString = array.join(', ');**
15. **reverse()**
    * *Description:* Reverses the elements of an array in place.
    * *Example:* **array.reverse();**
16. **every()**

* *Description:* Tests whether all elements in the array pass the provided function.
* *Example:* **const allGreaterThanZero = array.every(element => element > 0);**

1. **some()**

* *Description:* Tests whether at least one element in the array passes the provided function.
* *Example:* **const hasNegative = array.some(element => element < 0);**

1. **find()**

* *Description:* Returns the first element in the array that satisfies the provided testing function.
* *Example:* **const foundElement = array.find(element => element > 5);**

1. **findIndex()**

* *Description:* Returns the index of the first element in the array that satisfies the provided testing function.
* *Example:* **const index = array.findIndex(element => element > 5);**

1. **sort()**
   * *Description:* Sorts the elements of an array in place.
   * *Example:* **array.sort();** (Note: By default, it sorts elements as strings.)
2. **isArray()**
   * *Description:* Checks if a value is an array.
   * *Example:* **const isArray = Array.isArray(array);**
3. **fill()**
   * *Description:* Fills all the elements of an array with a static value.
   * *Example:* **array.fill(0);**
4. **flat()**
   * *Description:* Creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.
   * *Example:* **const flatArray = array.flat();**
5. **flatMap()**
   * *Description:* Maps each element using a mapping function, then flattens the result into a new array.
   * *Example:* **const newArray = array.flatMap(element => [element, element \* 2]);**
6. **toString()**
   * *Description:* Returns a string representing the array and its elements.
   * *Example:* **const arrayString = array.toString();**
7. **toLocaleString()**
   * *Description:* Returns a string representing the array and its elements using the system's locale-specific formatting.
   * *Example:* **const localeString = array.toLocaleString();**
8. **keys()**
   * *Description:* Returns a new array iterator that contains the keys for each index in the array.
   * *Example:* **const keys = array.keys();**
9. **values()**
   * *Description:* Returns a new array iterator that contains the values for each index in the array.
   * *Example:* **const values = array.values();**
10. **entries()**
    * *Description:* Returns a new array iterator that contains key/value pairs for each index in the array.
    * *Example:* **const entries = array.entries();**

**20-Nov-2023**

**Strings and String Methods:**

In C#, a string is a sequence of characters surrounded by double quotes (“ ”) represented by the **string** data type. C# provides a rich set of methods and features for working with strings. Here are some commonly used string methods in C#002E

static void Main(string[] args)

{

string msg1 = "Hi";

string msg2 = "How are you?";

Console.WriteLine(msg1);

Console.WriteLine(msg2);

Console.WriteLine(msg1.Length); //Displays the length of the string....

Console.WriteLine(msg2.Length);

Console.WriteLine(msg1.ToUpper()); //Converts all the characters into UPPERCASE....

Console.WriteLine(msg2.ToLower()); //Converts all the characters into lowercase....

Console.WriteLine(msg2.Trim()); //Removes excess white spaces....

Console.WriteLine(msg1.Equals(msg2)); //Checks whether two strings are same or not....

Console.WriteLine("==============String Clone================");

string text1 = "Hello";

string text2=(string)text1.Clone(); //Cloning string of one variable into another variable....

Console.WriteLine($"My text1 is {text1} and my text2 is {text2}");

Console.WriteLine("==================Copy=====================");

string text3 = "Byeeee";

string text4=string.Copy(text3); //Copying string of one variable into another variable....

Console.WriteLine($"My text3 is {text3} and text4 is {text4}");

Console.WriteLine("==================String Concatenation===============");

string msg3=msg1 + msg2; //Concatenation Method--1

Console.WriteLine(msg3);

string msg4 = string.Concat(msg1, msg2); //Concatenation Method--2

Console.WriteLine(msg4);

Console.WriteLine("==============String Interpolation ($)=================");

string firstName = "Durga Prasad";

string lastName = "Malthumkar";

Console.WriteLine($"My fullName is {firstName} {lastName}");

Console.WriteLine("=================CompareOrdinal()==================");

string s1 = "zebra";

string s2 = "jello"; //Compares two specified String objects by evaluating the

string s3 = "hello"; //numeric values of corresponding Char objects in each string

string s4 = "apple";

Console.WriteLine(string.CompareOrdinal(s1,s2));

Console.WriteLine(string.CompareOrdinal(s1,s3));

Console.WriteLine(string.CompareOrdinal(s1,s4));

Console.WriteLine("=====================EndsWith======================");

string n1 = "Prasad";

string n2 = "sad";

string n3 = "hii";

Console.WriteLine(n1.EndsWith(n2)); //It is used to check that the end of this string instance

Console.WriteLine(n1.EndsWith(n3)); //matches the specified string.

Console.WriteLine("==============GetHashCode=================");

string ghc = "Hello";

string ghc1 = "Who is this?";

string ghc2 = "This is Durga Prasad";

Console.WriteLine(ghc.GetHashCode()); //GetHashCode() method is used to get hash code of this string.

Console.WriteLine(ghc1.GetHashCode()); //It returns an integer value.

Console.WriteLine(ghc2.GetHashCode());

Console.WriteLine("==============IsNullOrEmpty================");

string ine = "Hi";

string ine1 = "";

string ine2 = " ";

Console.WriteLine(string.IsNullOrEmpty(ine)); //IsNullOrEmpty() checks whether the string is null or an Empty string.

Console.WriteLine(string.IsNullOrEmpty(ine1)); //It returns a Boolean value either true or false.

Console.WriteLine(string.IsNullOrWhiteSpace(ine2)); //Checks for the white spaces

Console.WriteLine("=====================Insert=======================");

string i1 = "Hello DP"; //The C# Insert() method is used to insert the specified string

string i2 = i1.Insert(5, "\_"); //at specified index number

Console.WriteLine($"Before insert {i1} after insert {i2}");

Console.WriteLine("================Accessing Strings================");

string myName = "Durga Prasad";

Console.WriteLine(myName[8]); //Access the string using index position....

Console.WriteLine(myName.IndexOf("P")); //Access the index position using character in the string....

Console.WriteLine("=====================Substring======================");

string name = "Durga Prasad";

int charpos=name.IndexOf("P");

//int charpos=name.IndexOf("r"); //Shows the index of first "r"....

//int charpos=name.LastIndexOf("r"); //Shows the index of last occurred "r"....

string Name =name.Substring(charpos);

Console.WriteLine(Name);

Console.WriteLine(name.Contains("D")); //Checks whether the given character is present or not....

Console.WriteLine(name.Contains("v"));

Console.WriteLine(name.Replace("Durga", "Malthumkar")); //Replaces the old string with the given new string....

string names = "Prasad,Vinishka,Viraaj"; //Splitting the given string based on delimiter....

string[] namesArray=names.Split(',');

foreach (string NAME in namesArray)

{

Console.WriteLine(NAME);

}

string greet = "Good Morning"; //Converting the string into Character Array....

char[] charArray=greet.ToCharArray();

foreach(char CHAR in charArray)

{

Console.WriteLine(CHAR);

}

**The below is the list of most used String methods:**

1. **Concatenation:**
   * + operator
   * String.Concat()
   * String.Join()
2. **Interpolation:**
   * String interpolation: $"Hello {name}"
3. **Length:**
   * String.Length
4. **Substring:**
   * String.Substring(startIndex, length)
   * String.Substring(startIndex)
5. **ToUpper and ToLower:**
   * String.ToUpper()
   * String.ToLower()
6. **Trim:**
   * String.Trim()
   * String.TrimStart()
   * String.TrimEnd()
7. **StartsWith and EndsWith:**
   * String.StartsWith()
   * String.EndsWith()
8. **Contains:**
   * String.Contains()
9. **IndexOf and LastIndexOf:**
   * String.IndexOf()
   * String.LastIndexOf()
10. **Replace:**
    * String.Replace()
11. **IsNullOrEmpty and IsNullOrWhiteSpace:**
    * String.IsNullOrEmpty()
    * String.IsNullOrWhiteSpace()
12. **PadLeft and PadRight:**
    * String.PadLeft()
    * String.PadRight()
13. **Remove:**
    * String.Remove()
14. **Insert:**
    * String.Insert()
15. **Compare:**
    * String.Compare()
    * String.CompareOrdinal()
16. **ToCharArray:**
    * String.ToCharArray()
17. **ToCharArray:**
    * String.ToCharArray()
18. **Join:**
    * String.Join()
19. **Format:**
    * String.Format()
20. **Copy:**
    * String.Copy()
21. **GetEnumerator:**
    * String.GetEnumerator()
22. **ToCharArray:**
    * String.ToCharArray()

**21-Nov-2023:**

***Exception Handling:***

**What is Exception?**

An exception is an abnormal (or) unexpected condition which results in abnormal termination of execution of program. All exceptions in C# are derived from **System.Exception** class. It is a runtime error which can be handled. If we don’t handle the exception, it prints the exception message and terminates the program.

**What is Exception Handling?**

Exception Handling in C# is a process to handle runtime errors. We perform Exception Handling so that normal flow of the application can be maintained even after runtime errors. The main **advantage** of Exception Handling is that it maintains the normal flow of the application. In that case, rest of the code is executed even after exception.

**🡺The below are the two main types of errors:**

* Compile Time Errors
* Run Time Errors
* Wrong implementation of logic.
* Wrong input supplied.
* Missing required resources.

**🡺The below is the list of C# common Exception classes….**

**System.DivideByZeroException** handles the error generated by dividing a number with zero.

**System.NullReferenceException** handles the error generated by referencing the null object.

**System.InvalidCastException** handles the error generated by invalid typecasting.

**System.IO.IOException** handles the Input Output errors.

**System.FieldAccessException** handles the error generated by invalid private or protected field access.

**🡺We use 4 main keywords in Exception Handling**

* try
* catch
* finally, and
* throw

**🡺How is exception raised?**

Whenever there is an error in the code, CLR creates an instance for that error and throws that instance as an exception which results in abnormal termination of rest of the program. This is the same as throwing a ball on the glass window and the glass breaks. To avoid this, we are using a mechanism called ***Exception Handling***.

**Advantages of EH: -**

* Abnormal termination stops so that the statements irrelevant to errors are executed successfully.
* We can display or deliver a user-friendly message to the end user so that we can describe the error.

**Syntax: -**

**try**

**{**

-Statements which will cause runtime errors.

-Statements which don’t require execution when a runtime error is raised.

**}**

**catch (<Exception Class Name> <variable>)**

**{**

-Statements which should execute only when there is a runtime error.

**}**

**Implementation: -**

Console.WriteLine("============Before Exception Handling============");

Console.WriteLine("Enter Numerator");

int num = int.Parse(Console.ReadLine());

Console.WriteLine("Enter Denominator");

int den = int.Parse(Console.ReadLine());

int div = num / den;

Console.WriteLine($"The result is {div}");

Console.WriteLine("End of the program");

Console.ReadKey();

**Note:** In the above case, when a DivideByZero exception rises then the program abnormally terminates at the particular statement without executing the rest of the program.

try

{

Console.WriteLine("============After Exception Handling=============");

Console.WriteLine("Enter Numerator");

int num = int.Parse(Console.ReadLine());

Console.WriteLine("Enter Denominator");

int den = int.Parse(Console.ReadLine());

int div = num / den;

Console.WriteLine($"The result is {div}");

}

catch (FormatException ex)

{

Console.WriteLine("Input must be numeric");

}

catch(DivideByZeroException ex)

{

Console.WriteLine("Division by Zero is not possible");

}

catch(Exception ex) //Additional catch block incase any unknown exception rises.

{ //Giving larger or smaller values than capacity.

Console.WriteLine(ex.Message);

}

finally

{

Console.WriteLine("End of the program");

}

Console.ReadKey();

A black screen with white text

Description automatically generated A black background with white text

Description automatically generated

**Note: -** After implementing exception handling, we will get a user-friendly error message.

* If the program successfully executes **without any exception**, then the control reaches the finally block to print “End of the program”.
* If a **DivideByZeroException occurs** during the process, the control is transferred to catch block which gives the exception message and then to finally block.
* When there is no exception raised in the try block, then no catch block will execute.

**23-Nov-2023:**

**Collections:**

In C#, a collection is a group of related objects, often referred to as elements, that are stored and manipulated as a single unit. Collections provide a way to organize and manage data efficiently. C# offers several built-in collection types in the **System.Collections** and **System.Collections.Generic** namespaces, each serving different purposes.

Once an array is declared we can never change the size of an array. Suppose we have an array of size 10 and we want to change it to size 15 all we need to do is:

* Copy the elements of previous array and create the new array of size 15 manually which is a time-taking process.
* Use the **Resize** method of Array class to increase or decrease the length of array. Resize method implicitly performs the above operation.

{

int[] arr=new int[10];

Array.**Resize**(ref arr, 15);

}

Three operations cannot be performed on Arrays:

* Increasing the size of an array.
* Adding or inserting the values in the middle of the Array.
* Deleting or removing the elements from the middle of an array.

To avoid or to overcome the above problems we go for **Collections**. Collections are the same as arrays which store many values, but collections allow us to perform the above operations.

Collections have a mechanism called **Auto-Resizing** which helps in increasing its size whenever we add new values to it.

**We have two types of collections. They are:**

* Non-Generic collections (Version 1.0)
* Generic collections (After 1.0)

**The main collections are: (Classes)**

* ArrayList
* Stack
* Queue
* Linked List
* Sorted List
* Hashtable

**Note: -** To use any of the above collections, we need to import the namespace **System.Collections** namespace into our project.

**Difference b/n Array and ArrayList:**

**Array ArrayList**

Fixed Length Variable Length

Not possible to insert items Possible to insert items in middle

Not possible to delete items Possible to dlte items from middle

**Note**: - When we create an ArrayList, it’s initial capacity is zero(0). But when we start adding something into the ArrayList, it starts increasing its size or capacity automatically starting with 4.

Whenever we fill 4 items into it, it doubles its size i.e., 8. This doubling process continues whenever we keep on adding the values into AL.

We can also fix the size of ArrayList manually during the creation. Whenever the specified size or capacity is filled, then the size is doubled.

ArrayList al = new ArrayList(); //Creating a new ArrayList

Console.WriteLine(al.Capacity);

al.Add(12);

Console.WriteLine(al.Capacity); //Finding capacity of ArrayList

al.Add(26);

al.Add(32); //Adding the values into AL

al.Add(11);

al.Add(14);

Console.WriteLine(al.Capacity);

foreach (object obj in al)

{

Console.Write(obj + " ");

Console.WriteLine();

}

al.Insert(2, 25); //Inserting based on the index value

foreach (object obj in al)

{

Console.Write(obj + " ");

Console.WriteLine();

}

al.Remove(25); //Removing the value directly

al.RemoveAt(4); //Removing using Index position

foreach (object obj in al)

Console.Write(obj + " ");

Console.WriteLine();

Console.ReadKey();

**Note**: - Array and ArrayList provides **index based accessing** which starts from 0 and ends with array length minus one and stores the data in key value combination (**Keys are the indexes here**). But unfortunately, those keys are predefined, and we cannot specify our own values to that keys. And when we want to access a random value from the Array or ArrayList, we have to remember and access it using index number which makes it complex. This drawback can be overcome by using Hashtable.

**Hashtable:**

Hashtable is also a collection which stores the data in the form of **Key:Value** pairs and can be accessed using index same as Array and ArrayList. But the only difference is Keys in Hashtable are **user-defined** i.e., we can specify our own Keys in the form of **strings, integers, double** etc. And we can easily access those Keys because we have specified them on our own.

Hahtable **Add** method is also same as Array Add method but the only difference is Add method in Array takes only one parameter i.e., **object value** whereas Add method in HashList takes two inputs i.e., **object key and object value.**

**Note: -** It is important to remember that, whenever we give randon keys(strings or integers) and we fetch values from Hashtable they will come in the same order as we gave but not in the sequential order. This is because when we specify keys and values, a **HashCode** is generated for every key and when we fetch all the values from Hashtable they are displayed in the same way.

static void Main(string[] args)

{

Hashtable ht=new Hashtable();

ht.Add("EmpId", 1267);

ht.Add("Ename", "Durga Prasad");

ht.Add("Dept", "MCB");

ht.Add("Role", "Dotnet Developer");

ht.Add("Email", "dp@gmail.com");

ht.Add("Phone", 6304195695);

ht.Add("Company", "Carbynetech");

ht.Add("Location", "Hi-Tech City");

ht.Add("Salary", 18000);

Console.WriteLine(ht.Count);

Console.WriteLine(ht.ContainsValue("Durga Prasad"));

Console.WriteLine(ht.ContainsValue("Hello")); //Checks whether given value is present or not

Console.WriteLine(ht.ContainsKey("EmpId"));

Console.WriteLine(ht.ContainsKey("StdId")); //Checks whether given key is present or not

ht.Remove("Dept"); //Removes the value with given key

Console.WriteLine("DP".GetHashCode());

Console.WriteLine("=========Only Keys=========");

foreach (object key in ht.Keys)

{

Console.WriteLine(key);

}

Console.WriteLine("=========Only Values=========");

foreach (object value in ht.Values)

{

Console.WriteLine(value);

//Console.WriteLine(ht[key]); value or ht[key] are similar

}

Console.WriteLine("=========Key : Value pair=========");

foreach (object key in ht.Keys)

{

Console.WriteLine(key + " : " + ht[key]);

}

Console.ReadKey();

**Note:-**

**Arrays: Types safe but fixed length.**

**Collections: Auto-Resizing but not type safe.**

**But we need the combination of Auto-Resizing and Type safe.**

**That is possible with a concept called Generics which exhibits the Auto-Resizing and Type safe properties.**

**Generics**

🡺The ArrayList what we have called in collections is now called **List<T>.** The ‘T’ represents the specified type of data that we want to store in the list.

**List<int> li=new List<int>().**

If we try to store data other than integers in the above list, we will get an error because the above list is specified only for integers.

**Few Examples:**

🡺List<string> ls=new List<string>();

🡺public class **Customer**

{

public int CustId{get; set;}

public string Name{get; set;}

}

List<**Customer**> lc=new List<**Customer**>();

In Generics, we use a namespace called **System.Collections.Generics**.

**List: -**

List<int> li = new List<int>();

li.Add(1);

li.Add(2);

li.Add(3);

li.Add(4);

li.Add(5);

li.Add(6); //Adds a value sequentially

li.Add(7);

li.Add(8);

li.Add(9);

li.Remove(9); //Removes a specified value

li.RemoveAt(7); //Removes a value from specified index

li.Insert(0, 0); //Adds a value at specified index

Console.WriteLine(li.Capacity); //Shows the capacity of List

Console.WriteLine("==========Using for loop==========");

for (int i=0;i<li.Count;i++)

{

Console.WriteLine(li[i]);

}

Console.WriteLine("==========Using foreach loop==========");

foreach (int intlist in li)

{

Console.WriteLine(intlist);

}

Console.ReadKey();

**How a method is made as Generic(accepting a specified value):-**

public bool Compare(int a, int b)

{

if (a == b)

return true;

return false;

}

static void Main(string[] args)

{

Generics1 obj1 = new Generics1();

bool result=obj1.Compare(1, 1);

Console.WriteLine(result);

Console.ReadKey();

}

**The above method is used only for comparing the integers**. If we want to compare **other data** types, then we need to write **another method** which is a complex process.

**Note:-** To reduce this complexity, we pass **object as a parameter in Compare method** instead of particular data type. But while passing objects as input parameters we use [**a.Equals(b)**] method instead of equal(**==**) operator.

namespace Generics1

{

class Generics

{

public bool Compare(object a, object b)

{

if (a.Equals(b))

return true;

return false;

}

}

internal class Program

{

static void Main(string[] args)

{

Generics obj = new Generics();

bool result=obj.Compare(true, 5);

Console.WriteLine(result);

Console.ReadKey();

}

}

}

But the objects take any type of data (not type safe). For example, if you pass integer for one parameter and different data type for other parameter the compare method doesn’t show any error. In addition, these objects requires boxing and unboxing. This is the drawback here.

**To overcome above drawbacks, we use below method.**

class Generics

{

public bool Compare<T>(T a, T b)

{

if (a.Equals(b))

return true;

return false;

}

}

internal class Program

{

static void Main(string[] args)

{

Generics obj = new Generics();

bool result=obj.Compare<**float**>(5.66f, 5.66f);

Console.WriteLine(result);

bool result1 = obj.Compare<**bool**>(false, true);

Console.WriteLine(result1);

Console.ReadKey();

}

}

While creating a Compare method we are specifying a **<T>** beside the method name and while passing the parameters which represents the type. That can be seen from above code. In this method when you specify a type, arguments should be passed with that data type only. Or else we’ll get an error. This is the main advantage. While the other advantage is it doesn’t require boxing and unboxing.

**Note:-** Here, we can call the call the Compare method any number of times for any data type.

**Different methods of Generics:**

* **Passing <T> beside a method name.**

namespace Generics2

{

class Generics2

{

public void Add<T>(T a, T b)

{

dynamic d1 = a;

dynamic d2 = b;

Console.WriteLine(d1+d2);

}

public void Sub<T>(T a, T b)

{

dynamic d1 = a;

dynamic d2 = b;

Console.WriteLine(d1 - d2);

}

public void Mul<T>(T a, T b)

{

dynamic d1 = a;

dynamic d2 = b;

Console.WriteLine(d1 \* d2);

}

public void Div<T>(T a, T b)

{

dynamic d1 = a;

dynamic d2 = b;

Console.WriteLine(d1 / d2);

}

}

internal class Program

{

static void Main(string[] args)

{

Generics2 obj = new Generics2();

obj.Add<int>(12, 26);

obj.Sub<int>(12, 26);

obj.Mul<int>(12, 26);

obj.Div<int>(12, 26);

Console.ReadKey();

}

}

}

From the above method while calling the methods, we are specifying the data type at every method name which makes it complex. Instead of that we can follow the below method. We can pass the **<T>** directly to the **class name.**

* **Passing <T> beside a class name.**

class Generics2**<T>**

{

public void Add(T a, T b)

{

dynamic d1 = a;

dynamic d2 = b;

Console.WriteLine(d1 + d2);

}

public void Sub(T a, T b)

{

dynamic d1 = a;

dynamic d2 = b;

Console.WriteLine(d1 - d2);

}

public void Mul(T a, T b)

{

dynamic d1 = a;

dynamic d2 = b;

Console.WriteLine(d1 \* d2);

}

public void Div(T a, T b)

{

dynamic d1 = a;

dynamic d2 = b;

Console.WriteLine(d1 / d2);

}

}

internal class Program

{

static void Main(string[] args)

{

Generics2**<int>** obj = new Generics2**<int>**();

obj.Add(10, 20);

obj.Sub(10, 20);

obj.Mul(10, 20);

obj.Div(10, 20);

Console.ReadKey();

}

}

**24-Nov-2023:**

**Delegates: -**

**Note: -**

* When a method is **static**, then we **need not to create an instance** for the class in which the method exists. We can call it using class name.
* When a method is **non-static**, then we **need to create an instance** for the class and then we need to call the method.
* Delegate is a type-safe function pointer.
* A delegate holds the reference of the method and then calls it for the execution.

To call a method using Delegate, we have 3 main steps:

1. **Define a delegate:**

[<modifiers>] delegate void|type <Name>(<parameter list>);

public delegate **void** AddDelegate(**int a, int b**); -----> **Delegate**

public **void** AddNums(**int a, int b**); -----> **Method**

public delegate **string** SayDelegate(string str)

public static **string** SayHello(string str)

1. **Instantiating the delegate:**

Instantiating is the process of creating an instance to the delegate and in this process, we need to pass the method name as a parameter to the delegate constructor.

AddDelegate ad=new AddDelegate(p.AddNums);

SayDelegate sd=new SayDelegate(SayHello or Program.SayHello);

1. **Calling the delegate:**

Now call the delegate by passing the required parameter values so that the method which is bound internally with the delegate will execute.

**Note: -**

1. The return type of the delegate should be same as the return type of the method.
2. The type of parameters to the delegate should be the same as type of parameters to the method.
3. That’s why we say delegate as a type-safe pointer.

namespace Delegates2

{

public delegate void AddDelegate(int a, int b); //Same return type

public delegate string SayDelegate(string str); //Same parameter type

internal class Program

{

public void AddNums(int a, int b) //Same return type

{

Console.WriteLine(a+b);

}

public static string SayName(string name) //Same parameter type

{

return "Hello " + name;

}

static void Main(string[] args)

{

Program p=new Program(); //Non-static method

AddDelegate ad = new AddDelegate(p.AddNums);

//ad(12, 26);

ad.Invoke(12, 26);

SayDelegate sd = new SayDelegate(SayName); //Static method

//string str = sd("Durga Prasad");

string str = sd.Invoke("Durga Prasad");

Console.WriteLine(str);

Console.ReadKey();

}

}

}

**Multicast Delegates: -**

A delegate which holds the reference of more than one method and can be called with the help of a delegate is called as **Multicast Delegate.**

* The type and number of parameters in the method should be same as the type and number of parameters in the delegate.
* Write the methods as non-value returning without any output parameters.

namespace Delegates2

{

public delegate **void** RectDelegate(**double Width, double Height**);

internal class Rectangle

{

public **void** GetArea(**double Width, double Height**)

{

Console.WriteLine("Area of Rectangle : "+Width \* Height);

}

public **void** GetPerimeter(**double Width, double Height**)

{

Console.WriteLine("Perimeter of Rectangle : "+2\*(Width+Height));

}

public **void** GetSum(**double Width, double Height**)

{

Console.WriteLine("The sum is : "+ (Width+Height));

}

static void Main(string[] args)

{

Rectangle rect = new Rectangle();

//Non-static method, hence use class object

RectDelegate rd = new RectDelegate(rect.GetArea);

//RectDelegate rd = rect.GetArea; Another way to create reference

rd += rect.GetPerimeter;

rd+= rect.GetSum;

rd.Invoke(12.26, 26.32);

//rd.Invoke(26.12, 32.26);

Console.ReadKey();

}}}

**25-Nov-2023:**

**Anonymous Methods: -**

Initially **create** a delegate, **instantiate** it and **invoke** it.

An Anonymous method refers to a method or function that is defined without name. These methods are mostly used for short and simple operations (less code). Anonymous methods are used in languages that have support for Delegates, Lambda expressions, or function pointers.

**Before Anonymous:**

namespace AnonymousMethod

{

//Create a delegate

//Input parameter type should be same as method

public delegate string GreetDelegate(string str);

internal class Program

{

//Input parameter type should be same as delegate

public static string Greeting(string name)

{

return "Hello " + name + " Good Morning";

}

static void Main(string[] args)

{

//Instantiate the delegate

//Greeting is a static method so call it directly

GreetDelegate gd = new GreetDelegate(Greeting);

string greet=gd.Invoke("Durga Prasad");

Console.WriteLine(greet);

Console.ReadKey();

}

}

}

In the above code, we are creating a method as a separate block and invoking it using a delegate. This looks a little complex. We can simplify it using an anonymous method as shown below. Here anonymous method is nothing but, we are not creating a method separately instead we are assigning a keyword called delegate during the delegate instantiation and assigning it with the parameter. Also write the printing statement for that.

**After Anonymous:**

namespace AnonymousMethod

{

public delegate string GreetDelegate2(string str);

internal class Anonymous

{

static void Main(string[] args)

{

GreetDelegate2 gd = delegate (string name)

{

return "Hello " + name + " Good Morning";

};

string str=gd.Invoke("Viraaj");

Console.WriteLine(str);

Console.ReadKey();

}

}

}

Using anonymous methods will make our code simpler, but it is only suggested in the cases where the code volumes are less. When we have larger volume of code, anonymous method is not recommended.

**🡺Create a Windows Form App**

Add a button from Toolbox tab. Double click the button, a code space is generated and we can specify our requirements there.

**Before Anonymous:**

namespace BtnDelegate

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

Button b=new Button();

b.Text = "Click Me";

b.Size = new Size(100, 20);

b.Location = new Point(250,250);

this.Controls.Add(b);

b.Click += new EventHandler(b\_Click);

}

private void b\_Click(object sender, EventArgs e)

{

MessageBox.Show("Hellooo!");

}}}

**We can rewrite the above code as follows to make it Anonymous:**

Instead of writing a separate method we are using an anonymous method in the following code and assigning it with the printing statement.

**After Anonymous:**

private void button1\_Click(object sender, EventArgs e)

{

Button b=new Button();

b.Text = "Click Me";

b.Size = new Size(100, 20);

b.Location = new Point(250,250);

this.Controls.Add(b);

//b.Click += new EventHandler(b\_Click);

**b.Click += delegate (object sender, EventArgs e)**

{

**MessageBox.Show("Hellooo!");**

};

}

**27-Nov-2023:**

**Lambda Functions: -**

In simple words, a Lambda Expression is shorthand for writing the Anonymous functions. It provides a more compact syntax to create the delegates. Lambda expressions are used in LINQ queries, event handling and other cases where short and inline function is needed.

namespace Lambda

{

public delegate string MyDelegate(string name);

internal class Program

{

/\*public static string MyMethod(string name)

{

return "Hi " + name + " This is my method";

}\*/

static void Main(string[] args)

{

MyDelegate md = (name)=>

{

return "Hi " + name + " This is my method";

};

string msg=md.Invoke("Vinishka");

Console.WriteLine(msg);

Console.ReadLine();

}

}

}

**Generic Delegates: -**

* Generic Delegates are the type of delegates that can work with any data types.
* In C#, we have 3 main types of Generic Delegates. They are:

1. Func delegate
2. Action delegate
3. Predicate delegate

**Basic Delegate: -**

namespace GenericDelegates

{

public delegate double Delegate1(int x, float y, double z);

public delegate void Delegate2(int x, float y, double z);

public delegate bool Delegate3(string str);

internal class Program

{

public static double AddNumbers(int x, float y, double z)

{

return (x + y + z);

}

public static void AddNumbers2(int x, float y, double z)

{

Console.WriteLine(x+y+z);

}

public static bool CheckLength(string str)

{

if(str.Length>5)

return true;

return false;

}

static void Main(string[] args)

{

Delegate1 d1 = new Delegate1(AddNumbers);

double result=d1.Invoke(12, 12.26f, 32.26);

Console.WriteLine(result);

Delegate2 d2 = new Delegate2(AddNumbers2);

d2.Invoke(12, 12.26f, 32.26);

Delegate3 d3 = new Delegate3(CheckLength);

bool res2 = d3.Invoke("Vinishka");

Console.WriteLine(res2);

Console.ReadKey();

}

}

}

**🡺Func:** The **Func** delegate represents a function that takes input parameters and returns a value. It can have up to 16 input parameters and one output parameter. The last type parameter is the return type.

**🡺Action:** The **Action** delegate represents a method that takes input parameters but does not return a value (void). Like Func, it can have up to 16 input parameters.

**🡺Predicate:** The **Predicate** is a specific type of generic delegate that represents a method that takes a single parameter of type **T** and returns a Boolean value. It's commonly used for defining conditions or criteria when filtering elements in a collection.

**Using Generic Delegates: -**

namespace GenericDelegates

{

/\*public delegate double Delegate1(int x, float y, double z);

public delegate void Delegate2(int x, float y, double z);

public delegate bool Delegate3(string str);\*/

internal class Program

{

public static double AddNumbers(int x, float y, double z)

{

return (x + y + z);

}

public static void AddNumbers2(int x, float y, double z)

{

Console.WriteLine(x+y+z);

}

public static bool CheckLength(string str)

{

if(str.Length>5)

return true;

return false;

}

static void Main(string[] args)

{

//Delegate1 d1 = new Delegate1(AddNumbers);

Func<int, float, double, double> d1 = AddNumbers; //returns a double value

double result=d1.Invoke(12, 12.26f, 32.26);

Console.WriteLine(result);

//Delegate2 d2 = new Delegate2(AddNumbers2);

Action<int, float, double> d2 = AddNumbers2; //returns nothing

d2.Invoke(12, 12.26f, 32.26);

//Delegate3 d3 = new Delegate3(CheckLength);

Predicate<string> d3 = CheckLength; //Func can also be used

bool res2 = d3.Invoke("Vinishka"); //returns a boolean value

Console.WriteLine(res2);

Console.ReadKey();

}

}

}

**Using Lambda functions in Generic Delegates: -**

namespace GenericDelegates

{

/\*public delegate double Delegate1(int x, float y, double z);

public delegate void Delegate2(int x, float y, double z);

public delegate bool Delegate3(string str);\*/

internal class LambdaGD

{

/\*public static double AddNumbers(int x, float y, double z)

{

return (x + y + z);

}

public static void AddNumbers2(int x, float y, double z)

{

Console.WriteLine(x + y + z);

}

public static bool CheckLength(string str)

{

if (str.Length > 5)

return true;

return false;

}\*/

static void Main(string[] args)

{

Func<int, float, double, double> func = (x, y, z) =>

{

return (x + y + z);

};

double res1 = func.Invoke(12, 12.26f, 32.26);

Console.WriteLine(res1);

Action<int, float, double> action = (x, y, z) =>

{

Console.WriteLine(x + y + z);

};

action.Invoke(12, 12.26f, 32.26);

Predicate<string> predicate = (str) =>

{

if (str.Length > 5)

return true;

return false;

};

bool res2 = predicate.Invoke("Viraaj");

Console.WriteLine(res2);

Console.ReadKey();

}

}

}

**Using Anonymous functions in Generic Delegates: -**

namespace GenericDelegates

{

internal class AnonymousGD

{

static void Main(string[] args)

{

Func<int, float, double, double> d1 = delegate (int x, float y, double z)

{

return (x + y + z);

};

double result = d1.Invoke(12, 12.26f, 32.26);

Console.WriteLine(result);

Action<int, float, double> d2 = delegate (int x, float y, double z)

{

Console.WriteLine(x + y + z);

};

d2.Invoke(12, 12.26f, 32.26);

Predicate<string> d3 = delegate (string str)

{

if (str.Length > 5)

return true;

return false;

};

bool res2 = d3.Invoke("Vinishka");

Console.WriteLine(res2);

Console.ReadKey();

}

}

}

**LINQ (Language Integrated Query)**

* Language Integrated Query (LINQ) is a new query language that has been designed by **Microsoft** in its **.Net 3.5.**
* This language syntax is like SQL query language.
* Using LINQ we can write queries on a wide variety of Data Sources like **Arrays, Collections, Data Tables, Data Sets, XML Data**.
* Before LINQ it was difficult to work with the arrays. Let us look at an example.
* Let us consider an array of marks and we need a new array of marks which consists of marks greater than 40.

namespace LINQ

{

internal class Program

{

static void Main(string[] args)

{

int[] arr = {56, 48, 12, 90, 43, 55, 12, 26, 46, 87, 14, 24, 60, 12, 23, 45, 67, 26};

int count = 0;

for(int i=0;i<arr.Length; i++)

{

if (arr[i] > 40)

count += 1;

}

int[] brr=new int[count];

int index = 0;

for(int i=0;i<arr.Length;i++)

{

if (arr[i]>40)

{

brr[index] = arr[i];

index += 1;

}

}

Array.Sort(brr);

Array.Reverse(brr);

foreach(int i in brr)

{

Console.Write(i+" ");

}

Console.ReadKey();

}

}

}

The above code snippet explains the filtering of marks (>40) from an array **arr** and storing those marks into a new array **brr** which is a complex process. But LINQ made it very simple just by using some clauses.

Suppose if we have the above values in the form of a table, we can access it by using SQL commands.

**select Col from Table1 where Col1>40 orderby Col1 desc 🡺** Table.

* If we can write a query on a table, why can’t we write a query for an array instead of writing the complex code. Hence LINQ has been introduced.

**from <alias> in <col|arr> [<clauses>]**

**select [<alias>] 🡺** Array or Table**.**

Using clauses is not mandatory but while using the clauses, they should be in **sequence** as mentioned below.

1. where
2. groupby
3. having
4. orderby

🡺To work with LINQ, **using System.Linq;** is required.

namespace LINQ

{

internal class LINQSyntax

{

static void Main(string[] args)

{

int[] arr = { 56, 48, 12, 90, 43, 55, 12, 26, 46, 87, 14, 24, 60, 12, 23, 45, 67, 26 };

var brr = from i in arr

where i>40

orderby i descending

select i;

foreach ( var i in brr )

{

Console.Write(i+" ");

}

Console.ReadKey();

}

}

}

As we have already discussed that LINQ is used to interact with different data sources, they are:

1. LINQ to Objects: Arrays, Collections etc.
2. LINQ to Databases: Data Tables, Relational Database Tables.

* Linq to ADO.NET
* Linq to SQL
* Linq to Entities

1. LINQ to XML.

🡺LINQ to SQL is not only about querying the data but also used to perform the CRUD operations.

🡺We can also call stored procedures by using LINQ to SQL.

**Comparison:**

**SQL to SQL Server: - LINQ to SQL Server: -**

* Run-time syntax checking. Compile-time syntax checking.
* Not Type Safe. Type-Safe.
* No Intellisense support. Intellisense support is available.
* Debugging of sql stmts is impossible. Debugging of Linq stmts is possible.
* Code is object-oriented and relational. Purely object-oriented code.

**In LINQ to SQL,**

Table---------------------------------------Class

Columns-----------------------------------Properties

Rows or Records-------------------------Instance/Object

Stored Procedures------------------------Methods

**èWhat is LINQ to SQL actually?**

Ø LINQ to SQL is an ORM (Object-Relational-Mapping) framework that can automatically create strongly typed .net classes based on the database tables.

Ø We can then write LINQ to SQL queries (Create, Read, Update, Delete) in any .Net supported Languages (C#, F#, VB.Net).

Ø The LINQ to SQL queries will then convert the LINQ queries to Transact SQL queries which can be understandable by SQL server database.

**Note: -** LINQ to SQL supports only the **SQL Server database**.

Since LINQ to SQL models a relational database using strongly typed .net classes, we have the following advantages:

1)     Intellisense Support.

2)     Compile time error checking.

3)     Debugging support.

**In SQL:**

* create database LinqtoSQL
* use LinqtoSQL
* Create two tables named Department and Employees.
* Create a relation between those two tables using DeptId.

**In Visual Studio:**

* Create an ASP.NET Web Application.
* Add LINQ to SQL class template.
* Create a connection to SQL Server.
* Add the tables into the white space.
* Now add a Web Form.
* Drag a GridView into the Form.

<div>

<asp:GridView ID="GridView1" runat="server" ></asp:GridView>

</div>

* Right click and open the code here.

namespace LINQtoSQL

{

public partial class WebForm1 : System.Web.UI.Page

{

protected void Page\_Load(object sender, EventArgs e)

{

SampleDataContext sdc=new SampleDataContext();

GridView1.DataSource = sdc.Employees;

GridView1.DataBind();

}

}

}

* If SampleDataContext() shows error, add the below constructor in Sample.designer.cs.

public SampleDataContext():

base(global::System.Configuration.ConfigurationManager.ConnectionStrings["LinqtoSQLConnectionString"].ConnectionString)

{

OnCreated();

}

**Note: -** After running the above code, we will get the list of all employees in a grid view. To get our desired list of employees we apply LINQ Queries as below.

protected void Page\_Load(object sender, EventArgs e)

{

SampleDataContext sdc=new SampleDataContext();

GridView1.DataSource = from employees in sdc.Employees

where employees.Gender=="Male"

orderby employees.Salary descending

select employees;

GridView1.DataBind();

}

This code sets the data source for the GridView1 control. The data source is defined using a LINQ query against a collection of Employees in the sdc object. And binds the data from the data source (set in the previous line) to the GridView1 control. In other words, it populates the GridView with the data retrieved from the data source.

🡺What is **Data Binding**?

* Data binding is the process of connecting data from a source to a user interface control (GridView). It establishes a link between the data and the presentation of that data in the user interface.

As we know that we are writing the LINQ queries in the code, how is the SQL Server able to understand those queries and send the data for us?

A diagram of a server

Description automatically generated

* Now rewrite the code such that it returns all the employees. Refract the code and name it as GetData().
* Call that GetData() method in Page\_Load method.
* Now create required buttons under the GridView and change their names, ID’s. Raise the events for them and write the below code.

**CRUD operations using LINQ to SQL**

namespace LINQtoSQL

{

public partial class WebForm1 : System.Web.UI.Page

{

protected void Page\_Load(object sender, EventArgs e)

{

}

private void GetData()

{

SampleDataContext sdc = new SampleDataContext();

GridView1.DataSource = sdc.Employees;

GridView1.DataBind();

}

protected void btnGetData\_Click(object sender, EventArgs e)

{

GetData();

}

protected void btnInsert\_Click(object sender, EventArgs e)

{

using(SampleDataContext sdc= new SampleDataContext())

{

Employee emp = new Employee

{

EmpName = "Mahesh",

Gender="Male",

Salary=3000,

DeptId=2

};

sdc.Employees.InsertOnSubmit(emp);

sdc.SubmitChanges();

}

GetData();

}

protected void btnUpdate\_Click(object sender, EventArgs e)

{

using(var sdc=new SampleDataContext())

{

Employee emp=sdc.Employees.FirstOrDefault(x=>x.EmpId==5);

emp.Salary = 18000;

sdc.SubmitChanges();

}

GetData();

}

protected void btnDelete\_Click(object sender, EventArgs e)

{

using(var sdc=new SampleDataContext())

{

Employee emp=sdc.Employees.FirstOrDefault(x=>x.EmpId == 8);

sdc.Employees.DeleteOnSubmit(emp);

sdc.SubmitChanges();

}

GetData();

}

}

}

**How to view the SQL statements generated by LINQ to SQL?**

For debugging , it is important to view the generated SQL statements. There are several ways to do that. The easiest way is to write the generated SQL query to ASP.NET page or a console window.

**For ASP.NET:-** SampleDataContext sdc = new SampleDataContext();

sdc.Log = Response.Output;

**For Console:-** sdc.Log = Console.Out;

**🡺How to view the exact SQL statements?**

private void GetData()

{

SampleDataContext sdc = new SampleDataContext();

//sdc.Log = Response.Output;

//sdc.Log = Console.Out;

var linqQuery = from employee in sdc.Employees

select employee;

string sqlQuery= linqQuery.ToString();

Response.Write(sqlQuery);

GridView1.DataSource = linqQuery;

GridView1.DataBind();

}

**🡺Another way to view the exact SQL statements?**

private void GetData()

{

SampleDataContext sdc = new SampleDataContext();

var linqQuery = from employee in sdc.Employees

select employee;

Response.Write(sdc.GetCommand(linqQuery).CommandText);

GridView1.DataSource = linqQuery;

GridView1.DataBind();

}

**Note: -** We can also use SQL Server Profiler to view the generated SQL commands (SQL Server---> Tools---> SQL Server Profiler).

**Stored Procedures with LINQ to SQL:**

**Retrieving data using Stored Procedures (Select query):**

A stored procedure is a SQL code which can be saved, so that it can be reused over and over again. So, when we have a SQL query which needed to be written and executed again and again, save it as a Stored Procedure and we can run it whenever needed.

1. Create a stored procedure with the name **GetEmployees** in SQL database for LinqtoSQL database.

create procedure GetEmployees

as

begin

select \* from Employee

end;

1. Now open the Visual Studio. Drag the stored procedure from Server Explorer to Sample.dbml whitespace.
2. After this, GetEmployees() will be displayed as method on top right. Right click on it, go to properties and change the **return type to Employee**.
3. Then write the below method. (Comment the previous method to avoid confusion).

private void GetData()

{

SampleDataContext sdc=new SampleDataContext();

GridView1.DataSource=sdc.GetEmployees();

GridView1.DataBind();

}

**Insert, Update, Delete using Stored Procedures:**

**Insert:**

create procedure InsertEmployees

@EmpName nvarchar(50),

@Gender nvarchar(50),

@Salary int,

@DeptId int

as

begin

insert into Employee(EmpName, Gender, Salary, DeptId)

values(@EmpName, @Gender, @Salary, @DeptId)

end

go

**Update:**

create procedure UpdateEmployee

@EmpId int,

@EmpName nvarchar(50),

@Gender nvarchar(50),

@Salary int,

@DeptId int

as

begin

update Employee

set EmpName = @EmpName, Gender=@Gender, Salary=@Salary, DeptId=@DeptId

where EmpId=@EmpId

end

go

**Delete:**

create procedure DeleteEmployee

@EmpId int

as

begin

delete from Employee where EmpId=@EmpId

end

go

Create the stored procedures for insert, update, delete as shown above. After doing this, drag these stored procedures into the white space in Visual Studio. Then right click on Employee table and click Configure Behavior. Select the behavior, customize it and check whether everything is okay or not. Do not change anything in the backend code. Run the project and implement the changes.

**Calling a Stored Procedures with output parameters:**

create procedure GetEmployeesByDept

@DeptId int,

@Name nvarchar(50) out

as

begin

select @Name=Name

from Department where DeptId=@DeptId

select \* from Employee

where DeptId=@DeptId

end

declare @DepartmentName nvarchar(50)

Execute GetEmployeesByDept 1, @DepartmentName out

select @DepartmentName

* After writing the above stored procedure, drag it into whitespace from the Server Explorer.
* Add a button in the webform and name it as GetEmployeeByDept.
* Add a label under the GridView.

A screenshot of a computer error

Description automatically generated

protected void btnGetEmployeeByDept\_Click(object sender, EventArgs e)

{

using(SampleDataContext sdc=new SampleDataContext())

{

string deptName=string.Empty;

GridView1.DataSource = sdc.GetEmployeesByDept(3, ref deptName);

GridView1.DataBind();

lblDept.Text="Department Name = " + deptName;

}

}

**04-Dec-2023:**

**Constructors: -**

* A constructor is a special method present under the class which is responsible for initializing the variables of that class.
* The name of the constructor should have the same name as that of the class under which it is present.
* It is a non-value returning method.
* Every class requires the constructor if we want to create the instance for that class.

**Note: -** It is the responsibility of a programmer to define a constructor under their class. If they fail to create the constructor for the class, then compiler will define an implicit constructor by default in that class. That can be seen in the case below.

* class Test()

{

int i;

}

Test obj = new Test();

**Note: -** Implicitly defined constructors are also known as Default constructors.

* class Test()

{

int i; string s; bool b;

public Test() //Explicit Parameter less constructor

{

i=0; //Initializing the variables

s=null;

b=false;

}

}

Test obj = new Test();

**Note: -**

* Implicitly defined constructors are parameter less and these constructors are also known as default constructors.
* Implicitly defined constructors are public.
* If we define a constructor by ourselves under a class, then it is known as Explicit Constructor.
* Explicit Constructor may be parameterized or parameter less.

**Implicit Constructor: -**

namespace Constructors

{

internal class Program

{

int i; bool b; //Declaring the variables

static void Main(string[] args)

{

//Implicit Constructor

Program prg=new Program();

Console.WriteLine("Value of i is: " + prg.i);

Console.WriteLine("Value of b is: " + prg.b);

Console.ReadKey();

}

}}

In the above program we did not create any explicit constructor, instead it was created implicitly by the CLR. So, when we create an instance of the class, a call to the implicit constructor is going to happen and it is going to initialize the variables i and b with their default values 0 and false.

**Syntax of a constructor: -**

[<modifier>] <name> (<parameter list>)

{

-Statements

}

**Explicit Constructor: -**

namespace Constructors

{

internal class ExplicitCons

{

public ExplicitCons() //Parameter less Constructor

{

Console.WriteLine("Explicit Constructor is called same as no. of instances created");

}

static void Main(string[] args)

{

ExplicitCons ec1 = new ExplicitCons();

ExplicitCons ec2 = new ExplicitCons();

ExplicitCons ec3 = new ExplicitCons();

Console.ReadKey();

}

}

}

In the above program, the number of times the constructor called(invoking) is equal to the no. of instances created to the class.

**Defining**: Implicit (or) Explicit.

**Calling**: Explicit (should be done explicitly by us).

**Types of Constructors: -**

1. Default or Parameter less Constructors.
2. Parameterized Constructors.
3. Copy Constructors.
4. Static Constructors.

* If a constructor method doesn’t take any parameters, then it is known as **Default** or **Parameter less** Constructor. These constructors can be defined explicitly by a programmer or can be defined implicitly by the CLR when there is no explicit constructor defined.  
  **E.g.**: public ExplicitCons(**no parameters**)
* If a constructor method is defined with any parameters, then it is known as **Parameterized Constructor**. These constructors can be defined by programmers (explicitly) only but not by the CLR.

**E.g.**: public ExplicitCons(**string s; int i;**)

namespace Constructors

{

internal class ParameterizedCons

{

int x;

public ParameterizedCons(int i) //Passing the parameter

{

x = i;

Console.WriteLine("Parameterized Constructor is called : "+i);

}

public void Display()

{

Console.WriteLine("The value of x : "+x);

}

static void Main(string[] args)

{

ParameterizedCons pc1 = new ParameterizedCons(12);

ParameterizedCons pc2 = new ParameterizedCons(26);

ParameterizedCons pc3 = new ParameterizedCons(32);

pc1.Display();

pc2.Display();

pc3.Display();

Console.ReadLine();

}

}

}

**Copy Constructor: -**

* When we want to create multiple instances with the same values, then we use this **Copy Constructor.**
* In Copy Constructor the constructor takes the same class name as input parameter to it.

namespace Constructors

{

internal class CopyCons

{

int x;

public CopyCons(int i) //Parameterized Constructor

{

x = i;

}

public CopyCons(CopyCons obj) //Copy Constructor

{

x=obj.x;

}

public void Display()

{

Console.WriteLine("The value of x : "+x);

}

static void Main(string[] args)

{

CopyCons cc1=new CopyCons(26);

CopyCons cc2=new CopyCons(cc1);

CopyCons cc3 = new CopyCons(cc1);

cc1.Display();

cc2.Display();

cc3.Display();

Console.ReadKey();

}

}

}

**Static Constructor: -**

If a constructor is declared explicitly using the static modifier, then it is known as **Static Constructor.** All the above constructors which are defined previously were **Non-Static Constructors** or **Instance constructors**.

Class Test

{

static Test() //Static constructor defined explicitly

{}

public Test() //Implicit default constructor

{}

}

1. If a class contains static variables, then only implicit default static constructors will be present or else we need to define them explicitly. Whereas non-static constructors will be implicitly defined in every class (except static class) provided, we do not define them explicitly.
2. Static Constructors are responsible in initializing static variables and these constructors are never called explicitly. They are called implicitly and moreover these constructors are first to execute under any class. (static cons executes first).
3. Overloading static constructors is not possible i.e., passing the parameters to static constructors is not possible.

namespace Constructors

{

internal class StaticCons

{

static StaticCons()

{

Console.WriteLine("Static Constructor is executed");

}

static void Main(string[] args)

{

Console.WriteLine("Main method is executed");

Console.ReadKey();

}

}

}

**Destructors: -**

A destructor works opposite of a constructor. It destroys the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

**using** System;

**public** **class** Employee

    {

**public** Employee()   //Constructor

        {

            Console.WriteLine("Constructor Invoked");

        }

        ~Employee()   //Destructor

        {

            Console.WriteLine("Destructor Invoked");

        }

    }

**class** TestEmployee{

**public** **static** **void** Main(**string**[] args)

        {

            Employee e1 = **new** Employee();

            Employee e2 = **new** Employee();

        }

    }

**05-Dec-2023:**

**Indexers: -**

* In C#, an indexer is a special type of property that allows us to access the elements of a class or a struct like an array.

* It enables instances of a class or struct to be indexed in a manner similar to arrays.
* Indexers are defined using **‘this’** keyword followed by square brackets **‘[ ]’** and can take one or more parameters to specify the indices.

**Syntax of Indexers: -**

[<modifier>] <type> this [int index]

{

[get {<statements>}] ====> get accessor

[set {<statements>}] ====> set accessor

}

**Employee.cs (contains get and set cases)**

namespace Indexers

{

internal class Employee

{

int Eno; //class members are private by default

string Ename;//we cannot access them in other class

double Salary; //If we make them public to access in other class everyone will access which is drawback

string Job; //properties-get, set access

string Dname; //Indexers- provides an access to access values using their index position

string Location;

public Employee(int Eno, string Ename, double Salary, string Job, string Dname, string Location)

{

this.Eno = Eno;

this.Ename = Ename;

this.Salary = Salary;

this.Job = Job;

this.Dname = Dname;

this.Location = Location;

}

public object this[string name]

{

get

{

if (index == 0)

return Eno;

else if (index == 1)

return Ename;

else if (index == 2)

return Salary;

else if (index == 3) //For indexes

return Job;

else if (index == 4)

return Dname;

else if (index == 5)

return Location;

return null;

if (name.ToUpper() == "ENO")

return Eno;

else if (name.ToUpper() == "ENAME")

return Ename;

else if (name.ToUpper() == "SALARY") //For strings

return Salary;

else if (name.ToUpper() == "JOB")

return Job;

else if (name.ToUpper() == "DNAME")

return Dname;

else if (name.ToUpper() == "LOCATION")

return Location;

return null;

}

set

{

if (index == 0)

Eno = (int)value;

else if (index == 1)

Ename = (string)value;

else if (index == 2) //For indexes

Salary = (double)value;

else if (index == 3)

Job = (string)value;

else if (index == 4)

Dname = (string)value;

else if (index == 5)

Location = (string)value;

if (name.ToUpper() == "ENO")

Eno = (int)value;

else if (name.ToUpper() == "ENAME")

Ename = (string)value;

else if (name.ToUpper() == "SALARY") //For strings

Salary = (double)value;

else if (name.ToUpper() == "JOB")

Job = (string)value;

else if (name.ToUpper() == "DNAME")

Dname = (string)value;

else if (name.ToUpper() == "LOCATION")

Location = (string)value;

}

}

}

}

**Indexers.cs (contains Main method)**

namespace Indexers

{

internal class Indexer

{

static void Main(string[] args)

{

Employee emp=new Employee(1226,"DP", 18000.00, "Developer", "MCB", "Hyderabad");

Console.WriteLine("============Using string name====================");

Console.WriteLine("================get================");

Console.WriteLine("Employee Id : " + emp["Eno"]);

Console.WriteLine("Employee Name : " + emp["Ename"]);

Console.WriteLine("Employee Salary : " + emp["Salary"]);

Console.WriteLine("Employee Job : " + emp["Job"]);

Console.WriteLine("Employee Department : " + emp["Dname"]);

Console.WriteLine("Location : " + emp["Location"]);

Console.WriteLine();

emp["Eno"] = 2612;

emp["Location"] = "Banglore";

Console.WriteLine("================set===================");

Console.WriteLine("Employee Id : " + emp["Eno"]);

Console.WriteLine("Employee Name : " + emp["Ename"]);

Console.WriteLine("Employee Salary : " + emp["Salary"]);

Console.WriteLine("Employee Job : " + emp["Job"]);

Console.WriteLine("Employee Department : " + emp["Dname"]);

Console.WriteLine("Location : " + emp["Location"]);

Console.WriteLine("===========Using Index=================");

Console.WriteLine("================get===================");

Console.WriteLine("Employee Id : " + emp[0]);

Console.WriteLine("Employee Name : " + emp[1]);

Console.WriteLine("Employee Salary : " + emp[2]);

Console.WriteLine("Employee Job : " + emp[3]);

Console.WriteLine("Employee Department : " + emp[4]);

Console.WriteLine("Location : " + emp[5]);

Console.WriteLine();

emp[1] = "Durga Prasad";

emp[2] = 20000.00;

Console.WriteLine("================set===================");

Console.WriteLine("Employee Id : " + emp[0]);

Console.WriteLine("Employee Name : " + emp[1]);

Console.WriteLine("Employee Salary : " + emp[2]);

Console.WriteLine("Employee Job : " + emp[3]);

Console.WriteLine("Employee Department : " + emp[4]);

Console.WriteLine("Location : " + emp[5]);

Console.ReadLine();

}

}

}

**06-Dec-2023:**

**Types of Variables: -**

There are four main types of C# variables namely:

1. Non-Static variables
2. Static variables
3. Constant variables
4. Readonly variables

**🡺 Non-Static variables:**

* In C#, non-static variables are often referred to as Instance Variables.
* These variables are associated with an instance of its existing class.
* Instance variables are declared without the static keyword.
* Initialization of instance variables can be done by directly assigning the value to the variable or can be done by Constructor Initialization.
* Instance class variables are initialized only when the instance of the class is created as well as each and every time the instance of class is created.

**Directly assigning the value:**

internal class Program

{

int d = 1226;

static void Main(string[] args)

{

Program p=new Program();

Console.WriteLine(p.d);

Console.ReadLine();

}

}

**Constructor Initialization:**

internal class InstanceVariable

{

int d;

public InstanceVariable(int i) //Constructor

{

d = i;

}

static void Main(string[] args)

{

InstanceVariable p=new InstanceVariable(26);

Console.WriteLine(p.d);

Console.ReadLine();

}

}

**🡺 Static variables:**

* If a variable is declared explicitly by using the static modifier (static int y) or else if a variable is declared under a static block, then those variables are said to be Static Variables.
* These variables do not require the instance of its exiting class.
* Static variables of a class are initialized immediately once the execution of class starts.
* We cannot initialize the static variables using Constructor.

internal class StaticVariables

{

static int y = 90;

static void Main(string[] args)

{

Console.WriteLine(y);

Console.ReadKey();

}

}

**🡺 Constant variables:**

* If a variable is declared using the “const” keyword, then those variables are known as Constant variables.
* Once a value is assigned (declared) to Constant variables it cannot be modified. Hence it is must to initialize the constant variable at the time of declaration.

**07-Dec-2023:**

**Types of Variables: -**

**C# Method Overloading:**

* C# Method Overloading can be defined as creating two or more methods with the same name but with different parameters (number or type of parameters).

(or)

* C# Method Overloading can be defined as an approach of defining method with multiple behaviors where behaviors of the method will be changing based on the parameters of that method.
* If input (number, type and order) changes, the output also changes.
* The main advantage of method overloading is, it increases the readability of the program because we need not create different names for the same action.

**Method Overloading in C# can be performed in two ways:**

1. By changing the number of arguments.
2. By changing the data type of arguments.

**Note : -** Though we have two same data type arguments, we can differ those constructors by interchanging the data types.

**Why do we need Method Overloading: -**

The concept of Method Overloading falls under Polymorphism. Polymorphism can be defined as changing the output bases on the input. Let us look at an example.

string s= “Hello World”;

s.IndexOf(‘o’); 4 //Returns first occurrence

s.IndexOf(‘o’, 5); 7 //Returns next occurrence

s.IndexOf(“l”); 2

The above program shows that the method named **IndexOf** produces different outputs when given different inputs. This is the best example of Overloading.

namespace Polymorphism

{

internal class Program

{

public void overload()

{

Console.WriteLine("Method with no parameter");

}

public void overload(int i)

{

Console.WriteLine("Method with integer parameter");

}

public void overload(string s)

{

Console.WriteLine("Method with string parameter");

}

public void overload(int i,string s)

{

Console.WriteLine("Method with integer and string parameter");

}

public void overload(string s,int i)

{

Console.WriteLine("Method with string and integer parameters");

}

static void Main(string[] args)

{

Program p=new Program();

p.overload();

p.overload(1226);

p.overload("Hello");

p.overload(1226, "Hello");

p.overload("Hello", 1226);

Console.ReadKey();

}

}

}

**C# Method Overriding:**

C# Method Overriding can be defined as an approach of reimplementing a parent class method under the child class with the same signature.

Class1----------Test( )

Class2 : Class1----------Test( )

**The main differences between Method Overloading and Overriding: -**

**Method Overloading: -**

1. In this case we define multiple methods with the same name but with different parameters.
2. This can be performed in the same class or between a parent class and child classes also.
3. While overloading a method of parent class under child class, child class need not take any permission from the parent class.
4. Overloading is all about defining multiple behaviors to a method.

**Method Overriding: -**

1. In the method overriding, we define multiple methods with same name and same parameters.
2. This can be performed only in parent class and child classes. It can never be performed in the same class.
3. While overriding a parent class method under the child class, child class need to take the permission (inherit) from parent class.
4. Overriding is all about changing the behavior of parent class method under the child class.

Note: - If we want to override the parent class method in its child class, then that method should be declared by using “**virtual**” keyword in the parent class. And “**override**” keyword should be used while declaring that method in the child class.

namespace Overriding

{

internal class Program

{

public virtual void method1()

{

Console.WriteLine("This is method 1 from Parent");

}

public virtual void method2()

{

Console.WriteLine("This is method 2 from Parent");

}

}

class Child: Program

{

public override void method1()

{

Console.WriteLine("This is method1 from child class using int parameter");

}

public override void method2() //overridden--signature must be same in parent and child class

{

Console.WriteLine("This is method 2 from Child"); //implementation can change

}

static void Main(string[] args)

{

Program p= new Program();

p.method1();

p.method2();

Child c= new Child();

c.method1();

c.method2();

Console.ReadLine();

}

}

}

**Output**

**A black background with white text

Description automatically generated**

**C# Method Hiding/Shadowing:**

**Case 1:-** As we already know that the method overriding is the mechanism/approach of re-implementing the methods of parent class under the child class with the same name and with the same signature.

**Case 2:-** Method Hiding is also an approach of re-implementing the methods of parent class under the child class exactly with the same name and same signature.

Note:- In the Case-1 child class re-implements the parent class methods only when the parent method is declared with the keyword “**virtual**”, whereas in case-2 child class implements parent methods even though parent method is not declared using “**virtual**” keyword.

**08-Dec-2023:**

**Abstract Classes: -**

A method without any method body (or) functionality is known as **Abstract method**. What the method contains is only the declaration of the method. If we use a keyword called “abstract” in a method without any functionality, then the method becomes Abstract method.

**Eg: public abstract void Method1();**

A class under which we define the abstract methods is known as **Abstract class**. Same as abstract method, we must use “abstract” keyword to declare abstract class too.

**Eg: abstract class Program**

**{**

**public abstract void Method1();**

**}**

If a method is declared as abstract under any class, then the child class of that abstract class is responsible for the implementation of abstract method without fail.

The concept of **abstract methods** will be **similar** to **Method Overriding**. The only main difference among these two concepts is:

* In Method Overriding we provide a re-implementation for an already existing implementation. Simply it can be called as Overriding the method. This is **optional**.
* In abstract class, we provide the pure implementation for an abstract method which has no initial implementation. This is **mandatory**.

**Method Overriding: Abstract Method:**

class MyClass1 abstract class MyClass1

{ {

public virtual void Show() public abstract void Show();

{ }

//Initial Implementation class MyClass2:MyClass1

} {

} public override void Show()

class MyClass2:MyClass1 {

{ //Implementation

public override void Show() }

{ }

//Re-implementation

}

}

**Note: -**

* Abstract class doesn’t mean that it only contains abstract methods. Along with abstract methods it also contains non-abstract methods.
* Whereas a non-abstract class contains only non-abstract methods but not abstract methods.
* Let us consider that a parent abstract class has two abstract methods (liabilities) and two non-abstract (assets) methods. If we want to use those non-abstract methods, initially we must and should create the implementation for abstract methods in its child class. After this we can consume non-abstract methods also.
* We cannot create an instance for abstract class.

namespace Abstract

{

abstract class Parent

{

public void Add(int x, int y)

{

Console.WriteLine(x+y);

}

public void Sub(int x, int y)

{

Console.WriteLine(x-y);

}

public abstract void Mul(int x, int y);

public abstract void Div(int x, int y);

}

}

namespace Abstract

{

internal class Child: Parent

{

public override void Mul(int x, int y)

{

Console.WriteLine(x\*y);

}

public override void Div(int x, int y)

{

Console.WriteLine(x/y);

}

static void Main(string[] args)

{

Child c=new Child();

c.Add(26,12);

c.Sub(26,12);

c.Mul(26,12);

c.Div(26,12);

Console.ReadKey();

}

}

}

**Interface: -**

* An interface is a user-defined data type which contains only abstract methods.

**Class: -** Non-abstract methods (Methods with functionality/method body).

**Abstract Class: -** Abstract methods as well as non-abstract methods.

**Interface: -** Only abstract methods (Methods without method body).

* Every abstract method of an interface should be implemented by the child class of the interface without fail.

A diagram of a group of members

Description automatically generated

* If a class is inherited from another class, it means that the child class is consuming the members of parent class. Whereas if a class is inheriting from an interface, it means that child class is implementing the members of the parent class (interface).

**Note: -** A class can be inherited from a class and also from an interface at a time.

**Class: Interface:**

[<modifiers>] class [<name>] [<modifiers>] interface [<name>]

{ {

-Define class members here -Define abstract methods here

} }

**Note: -**

* The default scope of **interface** members is **public** whereas the default scope of **class** members is **private.**
* By default, the methods in the interface are abstract. Hence, we need not use an abstract keyword just like we did in the abstract class .
* It is important to remember that **interfaces don’t contain any fields** (int x).

namespace Interfaces

{

internal interface MyInterface

{

void Add(int a, int b);

}

interface MyInterface2: MyInterface

{

void Sub(int a, int b);

}

class ImplementationClass: MyInterface

{

//void MyInterface.Add(int a, int b)

public void Add(int a, int b) //public modifier should be used

{ //while implementing the methods

Console.WriteLine(a+b);

}

}

class ImplementationClass2: MyInterface2

{

//void MyInterface.Add(int a, int b)

public void Add(int a, int b)

{

Console.WriteLine(a+b);

}

public void Sub(int a, int b)

{

Console.WriteLine(a-b);

}

static void Main(string[] args)

{

ImplementationClass imp1 = new ImplementationClass();

imp1.Add(12, 26);

ImplementationClass2 imp2 = new ImplementationClass2();

//imp2.Sub(12, 26); //Direct using of the object

//imp2.Add(12, 26);

MyInterface2 mi2 = imp2; //Giving implementation class object as a reference

mi2.Add(12, 26);

mi2.Sub(12, 26);

Console.ReadKey();

}

}

}

**Multiple Inheritance with Interface:**

A diagram of a class

Description automatically generated

* Single, Multi-level and Hierarchical inheritances are generally supported by classes. Bue Multiple inheritance is not supported by classes hence we go for interfaces to implement the Multiple inheritance.

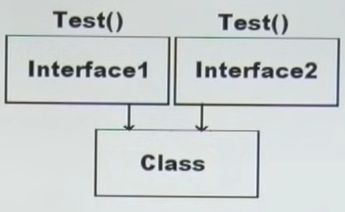
A class can have one and only one immediate parent class, whereas the same class can have any number of interfaces as it is a parent i.e., multiple inheritance in C# is supported through interfaces.

Why multiple inheritance is not supported by classes and how is it supported through interfaces?

A diagram of a class

Description automatically generated

Let us consider that we have two classes with names Class1 and Class2. Both classes contain two methods with **the same name and signatures (Test())**. Another class named Class3 inherits from these both classes. Because of this Class3 contains two methods with the same name and same signatures which raises the ambiguity error. Here Class1 and Class2 are asking Class3 to consume their similar methods which is not desirable.



To avoid this, we are using Multiple Inheritance with interfaces. In the above figure instead of two parent classes we have two interfaces. As we know that interface contains abstract methods, and its child class implements those methods. Here also the same happens. The methods of two interfaces are implemented by Child class instead of consuming it.

namespace Interfaces

{

interface Interface1

{

void Test();

void Show();

}

interface Interface2

{

void Test();

void Show();

}

internal class MultipleInheritance:Interface1, Interface2

{

public void Test()

{

Console.WriteLine("I am Test method from interfaces 1 and 2");

}

void Interface1.Show() //Give object reference when implementing a method like this

{

Console.WriteLine("I am Show method from Interface 1");

}

void Interface2.Show()

{

Console.WriteLine("I am Show method from Interface 2");

}

static void Main(string[] args)

{

MultipleInheritance mi=new MultipleInheritance();

mi.Test();

Interface1 i1 = mi;

i1.Show();

Interface2 i2 = mi;

i2.Show();

Console.ReadKey();

}

}

}

From the above code, we can observe that we have two Show() methods in two interfaces. But there is only one implementation of the Show method. Though there is a single implementation for two methods we didn’t get Ambiguity error. This is because those two interfaces don’t know about each other’s methods present in them. And here the advantage is that they have methods with the same name and same parameters. When we implement the method only once, those abstract methods think that the implementation is for them, and they get executed. Hence, we don’t get any ambiguity error.

**Access Specifiers: -**

In C#, access specifiers are defined as the keywords used to define the accessibility or visibility of members (fields, methods, properties, etc.) within a class. The most commonly used access specifiers are:

1. Public
2. Private
3. Protected
4. Internal
5. Protected Internal

* **Public: -** Members declared as public can be accessed from any other code in the same assembly or project.
* **Private: -** Members declared as private are accessible only within the same class but not in other classes.
* **Protected: -** Members declared as protected can be accessible only within the same class and its derived class (child class).
* **Internal: -** Internal members are accessed within the same assembly but not outside the assembly.
* **Protected Internal: -** Members declared as Protected Internal are accessible within the same assembly or in its derived class, whether they are in same assembly or in different one.

namespace AccessSpecifiers

{

internal class Program //Use only internal and public for class

{

private void Test1()

{

Console.WriteLine("I'm Private");

}

internal void Test2()

{

Console.WriteLine("I'm Internal");

}

protected void Test3()

{

Console.WriteLine("I'm Protected");

}

protected internal void Test4()

{

Console.WriteLine("I'm Protected Internal");

}

public void Test5()

{

Console.WriteLine("I'm Public");

}

static void Main(string[] args)

{

Program p=new Program();

p.Test1();

p.Test2();

p.Test3();

p.Test4();

p.Test5();

Console.WriteLine();

Console.ReadKey();

}

}

}

**File Handling: -**

File handling in C# refers to the ability to read from and write to files using the features provided by the C# programming language. This includes some tasks such as creating, opening, reading, writing and closing files.

namespace FileHandling

{

internal class Program

{

static void Main(string[] args)

{

FileStream fs1 = new FileStream("C:\\FileHandling\\FH.txt", FileMode.Create);

StreamWriter sw=new StreamWriter(fs1);

sw.WriteLine("Hi");

sw.WriteLine("Hello");

sw.WriteLine("Good Morning");

sw.Close();

Console.WriteLine("Successfully Added");

fs1.Close();

FileStream fs2 = new FileStream("C:\\FileHandling\\FH.txt", FileMode.Open);

StreamReader sr=new StreamReader(fs2);

string str;

while((str=sr.ReadLine())!=null)

{

Console.WriteLine(str);

}

sr.Close();

fs2.Close();

Console.ReadKey();

}

}

}

**ASP.NET**

* ASP in ASP.NET stands for Active Server Pages.
* ASP.NET is a web development framework developed by Microsoft for building Modern and Dynamic Web applications.
* It provides us a programming model, set of tools and libraries for building web applications and services.

**Before that we need to know what a Web Application is actually?**

* A web application is an application which can be accessed using a web browser by its users.

1. Microsoft Internet Explorer.
2. Google Chrome.
3. Mozilla Firefox.
4. Microsoft Edge.

**Other technologies that can be used to build web applications:**

1. Php
2. Ruby on Rails
3. Java
4. Perl

**What are the advantages of Web Applications?**

* Web Applications need to be installed only on the web server whereas the Desktop Applications need to be installed on every computer where we want to access them.
* Maintenance, Support and Patches are easier to provide.
* Only a browser (which can understand HTML) is required on the client’s machine to access the web application.
* Accessible from anywhere, only the requirement is internet.
* Cross-platform.
* On the server side, the web application runs under Microsoft Internet Information Services (IIS).

A diagram of a network

Description automatically generated

**What are WebForms?**

ASP.NET Web Forms is a web application framework within the ASP.NET framework, developed by Microsoft. It's designed to simplify the development of dynamic web pages by providing a higher-level abstraction over the stateless nature of the HTTP protocol. ASP.NET Web Forms uses a model that is similar to traditional Windows Forms programming, allowing developers to build web applications with a component-based architecture.

* WebForms has an extension of .aspx.
* A WebForm also has code behind files and designer files.
* Code behind files has an extension of **.aspx.cs** or **.aspx.vb**. And it contains the code that the user writes.
* Designer files have an extension of **.aspx.designer.cs** or **.aspx.designer.vb.** And these files contain the auto-generated code. We should not change the code in the designer file because later the code might be changed by the Visual Studio and our changes might be overwritten.
* A webform’s HTML can be modified either in Design or Source mode. We can also choose the Split mode which shows the two modes side by side.

**Postback:**

A **postback** in the context of web development, and particularly in the ASP.NET framework, is like sending a filled-out form back to the server. Let's break it down into simpler terms:

**Filling out a form (Making a request):** Imagine you're filling out a form on a website, entering your name, email, and some information. When you click a button on that form, you're essentially telling the website, "Hey, I'm done! Take a look at what I wrote."

**Sending the form to server (Postback):** Clicking that button is like sending the entire form (all the information you entered) back to the website's computer (the server). This process is called a postback.

**Server Processing (Handling the form):** The server receives your filled-out form and goes through its contents. It reads the information you provided and might do something with it, like saving it to a database or performing some calculations.

**Getting a response (Server responds):** After processing your form, the server sends back a new version of the webpage. It could be an updated form with new information or a completely different page. This is what you see on your screen after clicking the button.

In summary, a postback is like submitting a form on a website, and it involves sending your information to the server, which then processes it and sends back a response. In web development, this is a fundamental concept for creating interactive and dynamic web pages.

**ViewState in ASP.NET:**

* ViewState in ASP.NET is a mechanism that allows developers to persist (preserve, continue) the state of server controls across postbacks.
* Web Applications works on HTTP protocol. HTTP protocol is stateless protocol that means it does not retain state between user requests.
* Webforms live for barely a moment. Their lifecycle is:

1.An instance of requested webform is created.

2.Events processed.

3.Generates the HTML and posted to the client.

4.The webform is immediately destroyed.

**18-Dec-2023**

**Events in the life cycle of a Web Application:**

**What is an Event?**

In the context of Software Development and Web Applications, an event is a specific occurrence that is detected or triggered by the system or user.

Events play a crucial role in developing the responsive and dynamic web applications.

**In a web application, events can occur in 3 ways:**

1. At the application level. (Application Start)
2. At the page level. (Page Load)
3. At the control level. (Button Click)

* ViewState variables are used to preserve data across the page postback. However, by default ViewState of one webform is **not available** in another webform.

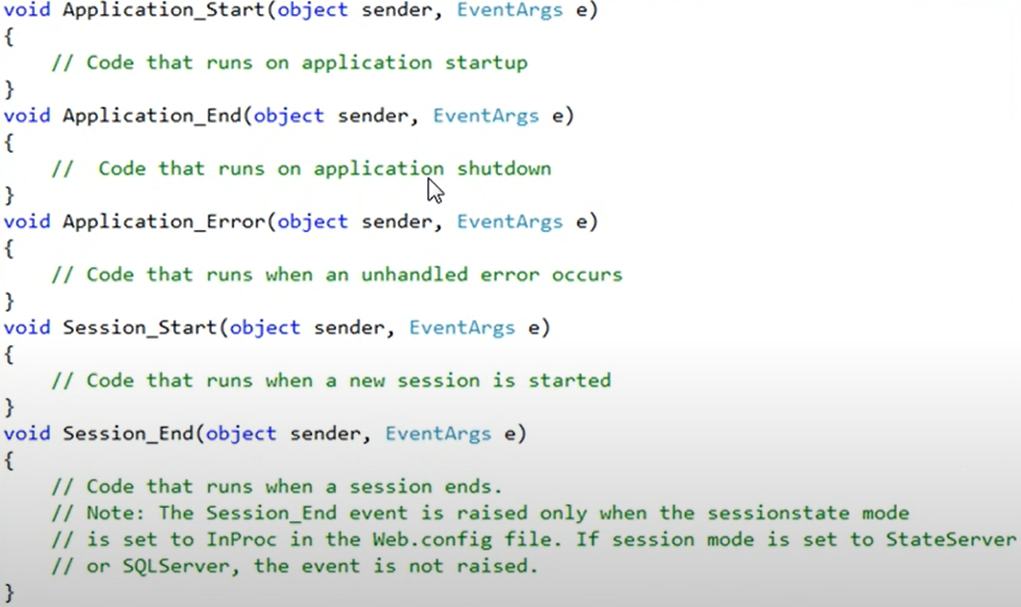
**Techniques to send data from one webform to another:**

1. Query Strings
2. Cookies
3. Session State
4. Application State

* Session State variables are available across all pages, but only for a given single session.
* Session variables are like single-user global data. Only the current session has access to its Session state.
* Application State variables are available across all pages and across all sessions too.
* Application State variables are like multi-user global data. All sessions can read and write Application State variables.
* Examples of Application events are Application\_Start, Application\_End etc.
* Examples of Session events are Session\_Start, Session\_End etc.

**Application level events:**

* In ASP.NET web application, **Global.asax** contains application level events.
* In general, Application events are used to initialize data that needs to be available to all the current sessions of application.
* Whereas session events are used to initialize data that need to be available only for a given individual session, but not between multiple sessions.



* Open visual studio. Create an ASP.NET Web Application.
* Create a web form.
* Create a Global.asax file too.
* In **Global.asax** file, write the below code.

public class Global : System.Web.HttpApplication

{

protected void Application\_Start(object sender, EventArgs e)

{

//Create application state variables

Application["TotalApplications"] = 0;

Application["TotalUserSessions"] = 0;

//Increment total applications by 1;

Application["TotalApplications"] = (int)Application["TotalApplications"] + 1;

}

protected void Session\_Start(object sender, EventArgs e)

{

//Increment TotalUserSessions by 1

Application["TotalUserSessions"] = (int)Application["TotalUserSessions"] + 1;

}

protected void Session\_End(object sender, EventArgs e)

{

//Decrement TotalUserSessions by 1

Application["TotalUserSessions"] = (int)Application["TotalUserSessions"] - 1;

}

* In **WebForm1.aspx.cs** file, write the below code.

public partial class WebForm1 : System.Web.UI.Page

{

protected void Page\_Load(object sender, EventArgs e)

{

Response.Write("Number of Applications : " + Application["TotalApplications"]);

Response.Write("</br>");

Response.Write("Number of Sessions : " + Application["TotalUserSessions"]);

}

}

**Note: -** Session hold cookies. The below explained SessionId is nothing, but the cookie generated by the web server.

A screenshot of a chat

Description automatically generated

Run the above program. We will get the above output. Try to open another tab and test the URL. We will get the same output i.e., no.of sessions is ‘1’ . Test the same URL by opening another instance of Google Chrome. Here also we’ll get the same output. This is because a **SessionId** is generated for the browser. When I open the second instance and use the same URL, the web server receives the same SessionId and thinks that the request is coming from the same user. Hence it sends the same output.

Now close the two instances of browser. Open it newly and raise the request again. Now a new SessionId is generated, and the web server thinks that this request has come from a new user.

**Open an instance of Google Chrome and test the project. Now open the instance of another browser like Microsoft Edge and test it. We’ll get different no.of sessions in two browsers. This is because the server receives requests from two different browsers and thinks that users are different. So SessionId’s (cookies) generated for them are also different.**

A screenshot of a computer

Description automatically generated

**19-Dec-2023**

**Diff b/w View State, Session State and Application State Variables:**

|  |  |  |
| --- | --- | --- |
| View State | Session State | Application State |
| ViewState of a webform is available only within that form | Session State variables are available across all pages, but only for a given single session. Session variables are like **single-user** global data. | Application State variables are available across all pages, and across all sessions. Application variables are like **multi-user** global data. |
| ViewState is stored on the page under a hidden file called \_ViewState. Because of this, the ViewState will be lost if we navigate to other page or if we close the browser. | Session State variables are stored on the web server. | Application State variables are stored on web server. |
| ViewState is used by all controls in the ASP.NET to retain their state across postback. | Session State variables are cleared if the user session times out. The default is 20 minutes. This can be configured in web.config. | Application State variables are cleared when the process hosting the application is restarted. |

**Note: -**

Let us prove the below cases:

* Session State data is actually shared across all webforms.
* ViewState data is stored/available only in that webform.

**In Visual Studio:**

1. Open the Visual Studio.
2. Create an ASP.NET Web application.
3. After this, create 6 webforms.
4. Create a textbox and button in all webforms.
5. Write the below code for all webforms.
6. In the place of ViewState, we need to use **Session** and **Application** keywords.

namespace Events

{

public partial class ViewState1 : System.Web.UI.Page

{

protected void Page\_Load(object sender, EventArgs e)

{

if(!IsPostBack)

{

if (**ViewState**["Clicks"]==null)

{

**ViewState**["Clicks"] = 0;

}

TextBox1.Text = **ViewState**["Clicks"].ToString();

}

}

protected void Button1\_Click(object sender, EventArgs e)

{

int ClicksCount = (int)**ViewState**["Clicks"] + 1;

TextBox1.Text = ClicksCount.ToString();

**ViewState**["Clicks"]=ClicksCount;

}

}

}

**For ViewState: -**

* Run the above program.
* If we click the submit button, the count increases.
* Now try to navigate to ViewState2 webform just by changing the name in URL.
* The count becomes zero for that webform. This is because each webform’s ViewState is stored in their respective hidden fields **\_ViewState.** And when we navigate to the other webform, the hidden field of previous webform is inactive and loses its value.

**A screenshot of a computer

Description automatically generated** **A screenshot of a computer

Description automatically generated**

**Web.config: -**

**<system.web>**

**<sessionState mode="InProc" cookieless="true"></sessionState>**

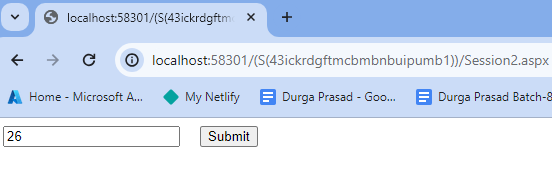
**</system.web>**

* **cookieless = “true” means we are telling the server not to use the cookies anymore. We are telling to use SessionId as a part of URL instead.**

**For Session: -**

* Change the keyword “ViewState” into “**Session**” in the code. And also add the above code into Web.config.
* Run the project.
* Click the Submit button.
* We can see that count increases. Now try to navigate to the Session2 webform.
* Though we get navigated to other webform the value in the textbox remains its state and when we click the Submit button the count begins to increase from that position. This is because the Session data available across all pages.

A screenshot of a computer

Description automatically generated 

* **In the middle of this process, once remove the SessionId from the URL and run it. A new SessionId is automatically generated in the URL and when it hits the server it identifies that this is not the SessionId which was given before. This makes the server thinks that request has came from the new user and the value in the textbox gets reset.**

1. Run the project initially.
2. Increase the value for some extent and copy paste the URL somewhere for further use.
3. Now run it using another SessionId.
4. Finally once paste the previous URL and check.
5. We’ll get the same value as before i.e., the **Session is capable of remembering the previous values**.

**How long the web server will retain/remember the data?**

If the server maintains the data indefinitely, it will run out of memory which is a drawback. Hence, we can configure the time period (minutes) for retaining the data by server.

<sessionState mode="InProc" cookieless="true" **timeout= “1**”></sessionState>

**For Application: -**

Application State variables are available across all pages, and across all sessions. Application variables are like **multi-user (**different browsers**)** global data.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

**ASP.NET page life cycle events:**

|  |  |
| --- | --- |
| **Event** **Name** | **Description** |
| **PreInit** | As the name suggests, this event happens just before the page initialization event starts. IsPostBack, IsCallBack and IsCrossPagePostBack are set at this stage. This event allows us to set the master page and theme of web application dynamically. PreInit is extensively used when working with dynamic controls. |
| **Init** | Page Init event occurs after the occurance of Init event of all the individual controls on the webform. Use this event to read or initialize the control properties. The server controls are loaded and initialized from the webform’s view state. |
| **InitComplete** | As the name suggests, this event is raised immediately after the page initialization. |
| **PreLoad** | Happens just before the page load event. |
| **Load** | Page Load event occurs before the occurance of load event of all the individual controls on the webform. |
| **Control Events** | After the page load event, the control events like Button Click, dropdownlist’s seleted index changed events are raised. |
| **Load Complete** | This event is raised after he control events are raised. |
| **PreRender** | This event is raised just before the renering stage of page. |
| **PreRenderComplete** | Raised immediately after the PreRender event. |
| **Unload** | Raised for each control and then for the page. Ath this stage the page is unloaded form the memory. |
| **Error** | This event occurs only if there is an unhandled exception. |

**The below shown code is the implementation of page life cycle events:**

namespace PageLifeCycleEvents

{

public partial class PLCEvents : System.Web.UI.Page

{

protected void Page\_PreInit(object sender, EventArgs e)

{

Response.Write("Page\_PreInit" + "<br/>");

}

protected void Page\_Init(object sender, EventArgs e)

{

Response.Write("Page\_Init" + "<br/>");

}

protected void Page\_InitComplete(object sender, EventArgs e)

{

Response.Write("Page\_InitComplete" + "<br/>");

}

protected void Page\_PreLoad(object sender, EventArgs e)

{

Response.Write("Page\_PreLoad" + "<br/>");

}

protected void Page\_LoadComplete(object sender, EventArgs e)

{

Response.Write("Page\_LoadComplete" + "<br/>");

}

protected void Page\_PreRender(object sender, EventArgs e)

{

Response.Write("Page\_PreRender" + "<br/>");

}

protected void Page\_PreRenderComplete(object sender, EventArgs e)

{

Response.Write("Page\_PreRenderComplete" + "<br/>");

}

protected void Page\_Unload(object sender, EventArgs e)

{

//Response.Write("Page\_Unload" + "<br/>");

}

}

}

**20-Dec-2023**

**ASP.NET Server Controls:**

ASP.NET Server controls, such as TextBox, Button, and DropDownList has their own events. We have a set of asp.net validation controls, that has validation events. The events that these controls expose, can be divided intlo 3 types:

**Postback Events: -** These events submit the WebPage, immediately to the server for processing. Click event of a button control is an example of PostBack event.

**Cached Events: -** These events are saved in the page’s viewstate to be processed when a postback event occurs. TextChanged event of TextBox control and SelectedIndexChanged event of a DropDownList control are examples of cached events. Cached events can be converted into postback events, by setting the AutoPostBack property of the control to true.

**Validation Events: -** These events occur on the client, before the page is posted back to the server. All the validation control use these types of events.