Functions in C#

In C#, functions (also known as methods) are blocks of code that perform a specific task. They are a fundamental concept in programming, allowing you to encapsulate code into reusable units. Here's a detailed explanation of functions in C#, along with examples.

There are two types of Functions:

1- Built-in Functions
2- User defined Functions

➔ Built-in Functions

These functions are available by .Net Framework to perform common tasks, ranging from mathematical operations to string manipulation, array handling, and more.

1. Math Functions

The Math class provides various mathematical functions.

- **Math.Abs**: Returns the absolute value of a number.

```
int value = -10;

int absoluteValue = Math.Abs(value); // absoluteValue = 10
```

- **Math.Pow**: Raises a number to a specified power.

```
double result = Math.Pow(2, 3); // result = 8
```

- **Math.Sqrt**: Returns the square root of a number.

```
double squareRoot = Math.Sqrt(16); // squareRoot = 4
```

- **Math.Max and Math.Min**: Return the larger or smaller of two numbers.

```
int max = Math.Max(5, 10); // max = 10

int min = Math.Min(5, 10); // min = 5
```

- **Math.Round**: Rounds a floating-point number to the nearest integer or specified number of decimal places.

```
double rounded = Math.Round(4.56789, 2); // rounded = 4.57
```

----

2. String Functions

The String class provides methods to manipulate strings.

- **string.Length**: Gets the length of a string.

```
string message = "Hello, world!";
int length = message.Length; // length = 13
```

- **string.Substring**: Extracts a substring from a string.

```
string sub = message.Substring(7, 5); // sub = "world"
```

- **string.ToUpper and string.ToLower**: Convert a string to uppercase or lowercase.

```
string upper = message.ToUpper(); // upper = "HELLO, WORLD!"
string lower = message.ToLower(); // lower = "hello, world!"
```

- **string.Trim**: Removes leading and trailing white-space characters from a string.

```
string trimmed = "  Hello  ".Trim(); // trimmed = "Hello"
```

- **string.Replace**: Replaces all occurrences of a specified string with another string.

```
string replaced = message.Replace("world", "C#"); // replaced = "Hello, C#!"
```

- **string.Contains**: Determines whether a string contains a specified substring.

```
bool contains = message.Contains("world"); // contains = true
```

- **string.Split**: Splits a string into an array of substrings based on a delimiter.

```
string[] words = message.Split(' '); // words = ["Hello,", "world!"]
```

---

3. Date and Time Functions

The DateTime class provides functions to work with dates and times.

- **DateTime.Now**: Gets the current date and time.

```
DateTime now = DateTime.Now;
```

- **DateTime.Today**: Gets the current date with the time component set to 00:00:00.

```
DateTime today = DateTime.Today;
```

- **DateTime.AddDays**: Adds a specified number of days to a DateTime object.

```
DateTime tomorrow = today.AddDays(1);
```

- **DateTime.ToString**: Converts the date and time to a string in a specified format.

```
string formattedDate = now.ToString("yyyy-MM-dd HH:mm:ss"); // formattedDate = "2024-08-18 15:30:00"
```

- **DateTime.Parse and DateTime.TryParse**: Parse a string representation of a date and time into a DateTime object.

```
DateTime parsedDate = DateTime.Parse("2024-08-18");

bool success = DateTime.TryParse("2024-08-18", out DateTime result);
```

---

4. Array Functions

C# arrays have several built-in methods.

- **Array.Sort**: Sorts the elements of an array.

```
int[] numbers = { 3, 1, 4, 1, 5 };

Array.Sort(numbers); // numbers = { 1, 1, 3, 4, 5 }
```

- **Array.Reverse**: Reverses the sequence of the elements in an array.

```
Array.Reverse(numbers); // numbers = { 5, 4, 3, 1, 1 }
```

- **Array.IndexOf**: Searches for the specified object and returns the index of its first occurrence in an array.

```
int index = Array.IndexOf(numbers, 4); // index = 1
```

- **Array.Resize**: Changes the size of a one-dimensional array.

```
Array.Resize(ref numbers, 10); // numbers is now of length 10
```

---

5. Console Functions

The Console class is used for basic input/output operations.

- **Console.WriteLine**: Writes the specified data, followed by the current line terminator, to the console.

```
Console.WriteLine("Hello, world!");
```

- **Console.Write**: Writes the specified data to the console without appending a new line.

```
Console.Write("Enter your name: ");
```

- **Console.ReadLine**: Reads the next line of characters from the standard input stream.

```
string name = Console.ReadLine();
```

---

6. Type Conversion Functions

C# provides several built-in methods for converting between types.

- **Convert.ToInt32**: Converts a specified value to a 32-bit signed integer.

```
string numberString = "123";

int number = Convert.ToInt32(numberString); // number = 123
```

- **Convert.ToDouble**: Converts a specified value to a double-precision floating-point number.

```
string doubleString = "123.45";

double doubleNumber = Convert.ToDouble(doubleString); // doubleNumber = 123.45
```

- **Convert.ToString**: Converts a specified value to a string.

```
int num = 123;

string numString = Convert.ToString(num); // numString = "123"
```

- **int.Parse and int.TryParse**: Convert a string representation of a number to its integer equivalent.

```
int parsedNumber = int.Parse("456"); // parsedNumber = 456

bool success = int.TryParse("456"); // success = true
```

---

8. Random Number Functions

The Random class is used to generate random numbers.

- **Random.Next**: Returns a random integer.

```
Random random = new Random();

int randomNumber = random.Next(); // randomNumber = any integer

int randomInRange = random.Next(1, 10); // randomInRange = integer between 1 and 9
```

- **Random.NextDouble**: Returns a random floating-point number between 0.0 and 1.0

```
double randomDouble = random.NextDouble(); // randomDouble = value between 0.0 and 1.0
```

➔ User Defined Functions

Are function defined by user to accomplish specific tasks that are not done by system and tune them to specific needs

1. Defining a Function

In C#, a function is defined within a class or struct. The general syntax for defining a function is:

```
[access_modifier] [return_type] [function_name]([parameters])
{
   // Function body
}
```

- **access_modifier:** Specifies the accessibility of the function (e.g., public, private, protected, internal). This determines which other parts of the code can call the function.
- **return_type:** Specifies the type of value the function returns (e.g., int, string, void). If the function does not return a value, the return type is void.
- **function_name:** The name of the function. It should be descriptive and follow C# naming conventions (usually PascalCase).
- **parameters:** A list of input parameters, if any, that the function requires. Each parameter has a type and a name.

2. Function Example: No Parameters, No Return Value

A simple function that does not take any parameters and does not return a value:

```
public void PrintHello()
{    Console.WriteLine("Hello, world!"); }
```

- **Explanation:** This function, PrintHello, has the public access modifier, no parameters, and a return type of void. It prints "Hello, world!" to the console when called.

3. Function Example: Parameters, No Return Value

A function that takes parameters but does not return a value:

```
public void PrintGreeting(string name)
{ Console.WriteLine($"Hello, {name}!"); }
```

- **Explanation:** This function, PrintGreeting, takes one parameter of type string named name. It prints a greeting message to the console that includes the provided name.

4. Function Example: Parameters with Return Value

A function that takes parameters and returns a value:

```
public int Add(int a, int b)
{
   return a + b;
}
```

- **Explanation:** The Add function takes two int parameters (a and b) and returns their sum. The return type of the function is int.

5. Function Example: No Parameters, Return Value

A function that does not take parameters but returns a value:

```
public DateTime GetCurrentTime()
{
   return DateTime.Now;
}
```

- **Explanation:** This function, GetCurrentTime, returns the current date and time. It does not take any parameters, and its return type is DateTime.

6. Overloading Functions

C# allows function overloading, which means you can define multiple functions with the same name but different parameter lists. The compiler differentiates them based on the number and types of parameters.

```
public int Multiply(int a, int b)
{   return a * b; }
public double Multiply(double a, double b)
{   return a * b; }
public int Multiply(int a, int b, int c)
{   return a * b * c; }
```

- **Explanation:** The Multiply function is overloaded with three different parameter lists:
    - The first version multiplies two integers.
    - The second version multiplies two doubles.
    - The third version multiplies three integers.

7. Optional Parameters

You can specify default values for parameters, allowing them to be optional when calling the function.

```
public void PrintMessage(string message = "Default Message")

{

   Console.WriteLine(message);

}
```

- **Explanation:** The PrintMessage function has one parameter, message, which has a default value of "Default Message". If the caller does not provide a value, the default value is used.

8. Named Parameters

When calling a function, you can specify the name of the parameters explicitly, which allows you to pass arguments in a different order or only pass specific optional parameters.

```
public void PrintDetails(string name, int age, string city)

{

   Console.WriteLine($"Name: {name}, Age: {age}, City: {city}");

}


// Calling with named parameters

PrintDetails(city: "New York", name: "Alice", age: 25);
```

- **Explanation:** The PrintDetails function is called using named parameters, allowing the arguments to be passed in any order.

9. Passing Parameters by Value vs. by Reference

By default, C# passes parameters by value, meaning a copy of the argument is passed to the function. However, you can pass parameters by reference using the ref or out keywords.

- **By Value (default):**

```
public void IncrementValue(int x)

{   x++; }

int number = 5;

IncrementValue(number);

Console.WriteLine(number); // Output: 5
```

  - **Explanation:** IncrementValue increments the value of x, but since x is passed by value, the original variable number remains unchanged.

- **By Reference (using ref):**

```
public void IncrementValue(ref int x)
{  x++; }
int number = 5;
IncrementValue(ref number);
Console.WriteLine(number); // Output: 6
```

  - **Explanation:** IncrementValue now uses the ref keyword, so the original variable number is incremented.

- **By Reference (using out):**

```
public void Initialize(out int x)
{
   x = 10; // Must assign a value to x
}
int number;
Initialize(out number);
Console.WriteLine(number); // Output: 10
```

  - **Explanation:** The out keyword is used to indicate that the function will assign a value to the parameter, which must be done before the function ends.

10. Returning Multiple Values (Using Tuples)

C# allows you to return multiple values from a function using tuples.

```
public (int sum, int product) Calculate(int a, int b)
{
   int sum = a + b;
   int product = a * b;
   return (sum, product);
}
// Calling the function
var result = Calculate(3, 4);
Console.WriteLine($"Sum: {result.sum}, Product: {result.product}");
```

- **Explanation:** The Calculate function returns a tuple containing both the sum and the product of two numbers. The tuple is then deconstructed to access the individual values.

11. Local Functions

In C#, you can define functions inside other functions, known as local functions. These are useful when the function is only needed within the scope of another function.

```
public int CalculateSum(int[] numbers)
{
    int Sum(int[] nums)
    {
        int total = 0;
        foreach (var num in nums)
        {
            total += num;
        }
        return total;
    }

    return Sum(numbers);
}
```

- **Explanation:** The Sum function is defined inside CalculateSum and is only accessible within that method.

12. Recursive Functions

A recursive function is one that calls itself to solve a problem. It usually has a base case to stop the recursion.

```
public int Factorial(int n)
{
    if (n == 1)
        return 1;
    else
        return n * Factorial(n - 1);
}
```

- **Explanation:** The Factorial function calculates the factorial of a number recursively. The base case is when n is 1.