FLOW OF FUNCTION EXCEUTION:

NOTE:
The basic difference between Methods and Functions is that the **method comes with a void** as return type whereas **function has a return type**.

The steps of how the function executes in the backend of the machine are:

1. **Compilation**: You write C# source code, and the C# compiler translates it into Common Intermediate Language (CIL or IL). This IL code is stored in an assembly, which can be a DLL or an EXE file.

2. **Loading the Assembly**: When you run a .NET program, the .NET runtime loads the assembly containing your code. The runtime performs various tasks like security checks, verifying the assembly's digital signature (if any), and ensuring that the code meets safety and permission requirements.

3. **Execution Starts in Main**: In a console application or an entry point for a desktop application, execution begins in the Main method. In the case of class libraries, execution starts from where the library is invoked in the application.

4. **Method Invocation**: When the Main method (or any other entry point) calls the *AddNumbers* method, it creates a stack frame for the method call. The stack frame includes space for local variables, method parameters, and the return address. This space is essentially a memory area allocated for the method's execution.

5. **Parameter Passing**: The values 5 and 3 are passed as arguments to the *AddNumbers* method. These values are typically stored on the method's stack frame as local variables, and they are accessible to the method during execution.

6. **Method Execution**: The CIL code generated by the compiler is executed by the Common Language Runtime (CLR). The '*AddNumbers'* method is executed, performing the addition operation, and the result is computed.

7. **Return**: When the '*AddNumbers*' method completes execution, it returns the result, which is an int. This return value is typically placed in a designated location (e.g., a CPU register or a memory location) for the caller to access.

8. **Stack Cleanup**: The stack frame for the '*AddNumbers*' method is removed, and the program continues execution with the result from the method call.

9. **Display**: The result is displayed using 'Console.WriteLine' or another method, as appropriate for the application.

CONSOLEAPP.PROGRAM

```
using System;

namespace ConsoleApp

public class Program

{

Public static void Main()

    {

        Console.WriteLine("Here We are adding two numbers.");

        int result = AddNumbers(5, 3);

        Console.WriteLine("The result of adding 5 and 3 is: " + result);

    }

    static int AddNumbers(int a, int b)

    {

        return a + b;

    }

}
```

EXAMPLE 2:

name1.n1

```
using System;
namespace name1
{
    internal class n1
    {
        public static int add(int a, int b)
        {
            int result = a + b;
            return result;
        }

        public static void main(String[] args)
        {
            int x = 5;
            int y = 7;
            int sum = add(x, y);
            Console.Writeline("this is the sum ", sum);
        }
    }
}
```

As C# supports procedural or imperative programming paradigm Main function will be considered as the starting/entry point of the program.

Here as the program encounters a function call Main(entry point), it needs to record the current state of execution, including the memory location of the next instruction to be executed.

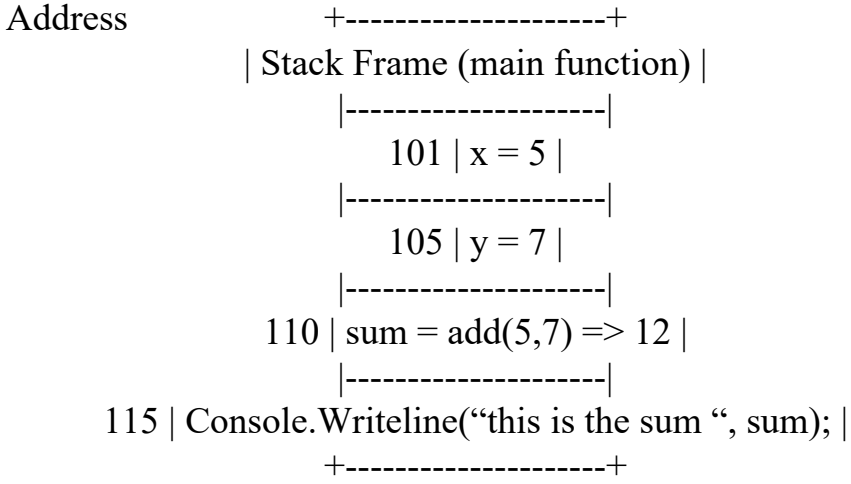
Program record: **101 | 105**

Once the program encounters a function it allocates a stack memory to it and then jumps to the memory location of the function and starts executing the first statement inside the function.

The function might allocate memory on the stack or in registers for local variables. This memory is used to store data that is specific to the function.

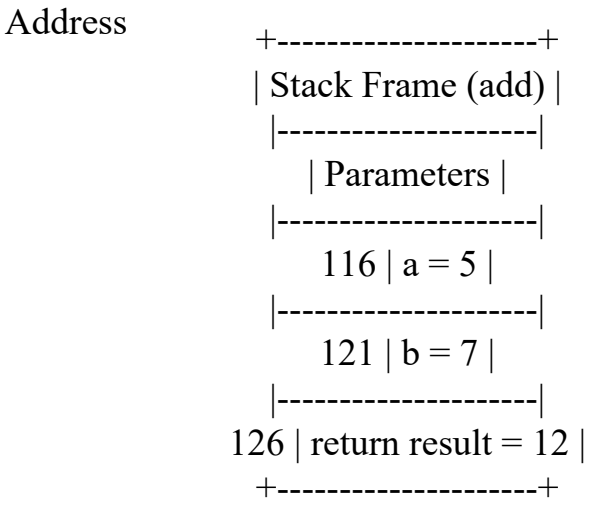
The statements inside the function are executed in order. Control flows through conditional statements (if-else), loops, and other constructs as specified in the function's code. Local variables are read from and written to during this process.

Program record: **105 | 110**



If the function makes calls to other functions, the process of recording the state, entering the new function, and executing it stepwise is repeated. This can lead to a call stack of functions being executed, with the innermost function executing first and the outer functions waiting for the inner ones to complete.

Program record: **110 | 116**



When the function reaches a return statement, it typically returns a value to the caller. The return value is often placed in a designated register or memory location. The function's local variables may also be deallocated at this point.

The program returns to the calling function, typically by popping the state information from the call stack and restoring the previous instruction pointer and data. The return value from the function call is used as needed in the calling function.

The program continues executing the instructions following the original function call, using the returned value or continuing with other operations.

Process record: **110 | 115**

//**Output :** ***this is the sum 12.***