Call stacks

Understanding flow

Small puzzle:

It's important to understand, what's happening when we're calling a function and what's going to be the next line executed.

```
/*01*/    void A() {
/*02*/    Console.Write("A");
/*03*/    }
/*04*/    void B() {
/*05*/    Console.Write("B");
/*06*/    }
/*07*/    void C() {
/*08*/    B();
/*09*/    Console.Write("C");
/*10*/    }
/*11*/    C();
/*12*/    A();
```

So, what's happening here, exactly?

Step by step

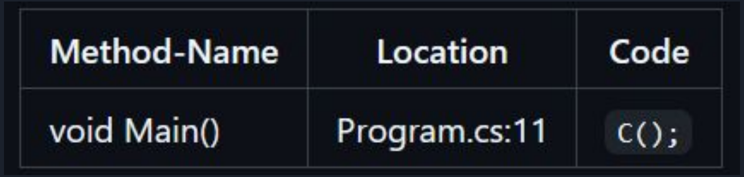
Let’s examine what goes on exactly:

Well, first of all, the bodies of any of the three functions don't get executed, unless the function is invoked.

Which means, that the program falls all the way through to line 11:

- /*11*/ C();

The CallStack will look something like this:

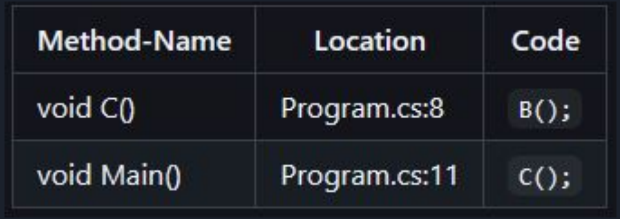


Method-Name	Location	Code
void Main()	Program.cs:11	`C();`

Where Main() is the Function-Name that's enclosing our whole C#-Script, Program.cs is the File-Name of our C#-Script and 11 is the Line Number where the execution currently stands.

In this line, function C() will be invoked, which will put the function's routine on top of the stack:

Step 2



Method-Name	Location	Code
void C()	Program.cs:8	`B();`
void Main()	Program.cs:11	`C();`

You can see, that the original Program routine, Main() is still in Line 11. It is waiting for the routine C(), on top of the stack, to finish execution.

What does Stack mean exactly? A Stack in Programming is a Data Structure, which always pushes new items on the top and only allows access to the top-most item. When you're done with the top-most item, you can pop it.

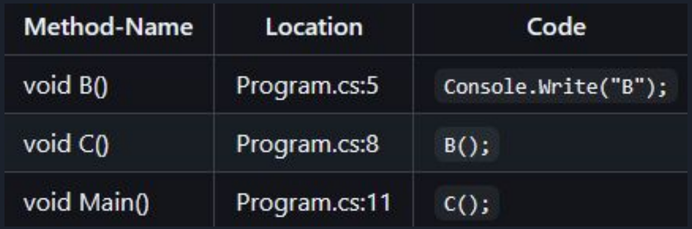
You can imagine a stack of cards or a stack of plates as an analogy. You usually put cards and plates on the top of the stack and also remove them only from the top.

Let's look at line 8 again:

- **/*08*/ B();**

Oof, another Function-Invocation. Guess what's happening next?

Step 3



Method-Name	Location	Code
void B()	Program.cs:5	`Console.Write("B");`
void C()	Program.cs:8	`B();`
void Main()	Program.cs:11	`C();`

Okay, we can see the new Method Invocation being pushed on top of the stack as well. So, both Main() and C() are paused, waiting for B(), which is on top of the stack, to finish.

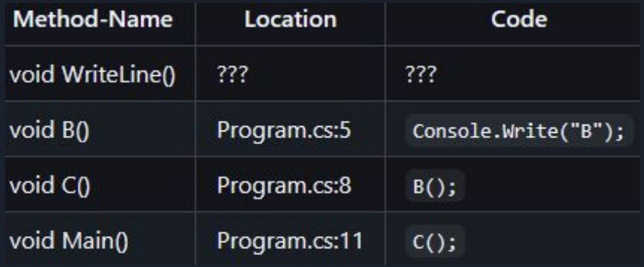
In Line 5, we find:

- /*05*/ Console.Write("B");

Oh no, actually we're invoking another function here.

Which means, that the Call Stack will look something like this..

Step 3.1



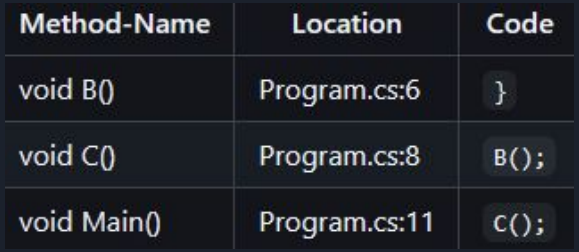
Method-Name	Location	Code
void WriteLine()	???	???
void B()	Program.cs:5	`Console.Write("B");`
void C()	Program.cs:8	`B();`
void Main()	Program.cs:11	`C();`

But since this is external code, we don't need to bother about what's exactly happening there.

We know that some code will be executed, which will print a B to the console.

And then, when that function has completed, it will be popped from the stack again, allowing our function to continue…

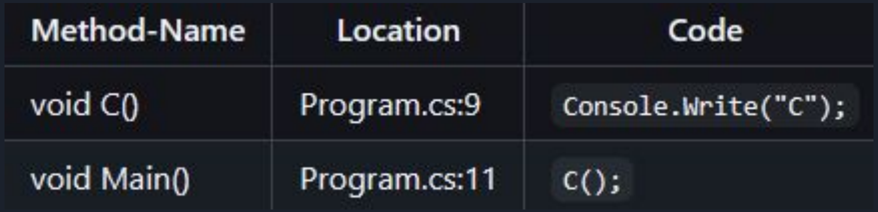
Step 3.2



Method-Name	Location	Code
void B()	Program.cs:6	`}`
void C()	Program.cs:8	`B();`
void Main()	Program.cs:11	`C();`

And here, we can see, that function B() also ends right there, which will cause it to also be popped from the stack, allowing C() to continue..

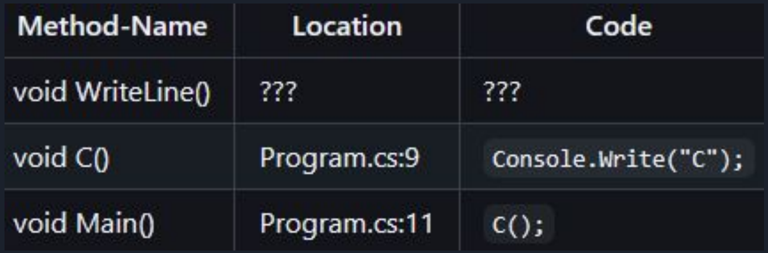
Step 4



Method-Name	Location	Code
void C()	Program.cs:9	`Console.Write("C");`
void Main()	Program.cs:11	`C();`

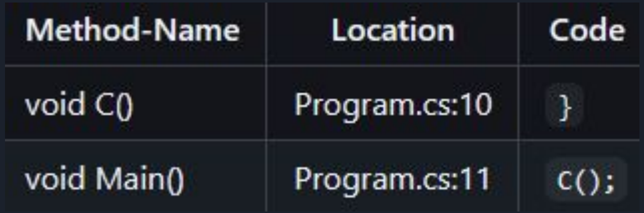
I believe that at this point, you have a good idea of how this will continue :)

4.1..



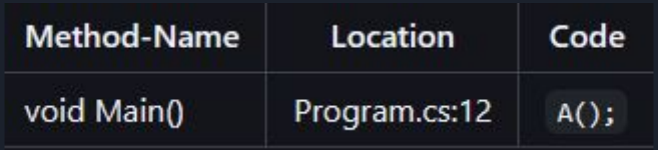
Method-Name	Location	Code
void WriteLine()	???	???
void C()	Program.cs:9	`Console.Write("C");`
void Main()	Program.cs:11	`C();`

4.2



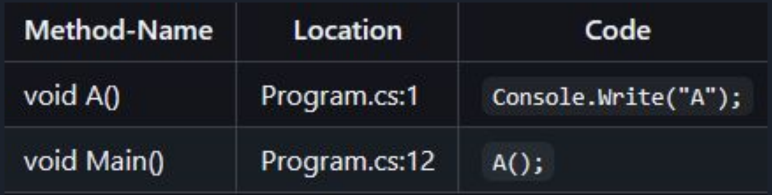
Method-Name	Location	Code
void C()	Program.cs:10	`}`
void Main()	Program.cs:11	`C();`

5



Method-Name	Location	Code
void Main()	Program.cs:12	`A();`

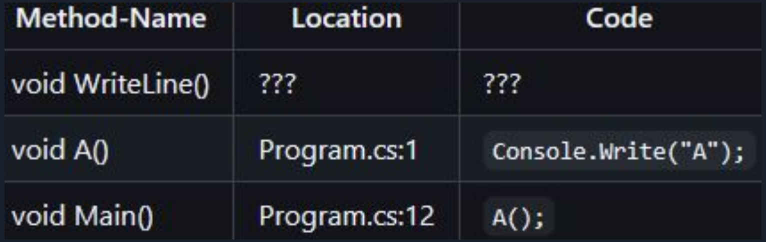
6



Method-Name	Location	Code
void A()	Program.cs:1	`Console.Write("A");`
void Main()	Program.cs:12	`A();`

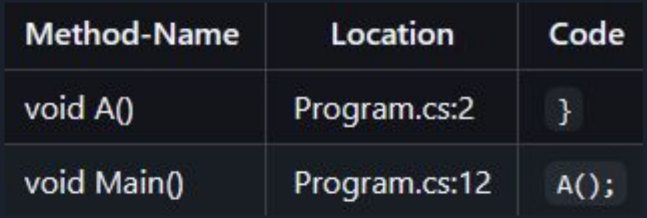
And the last steps..

6.1



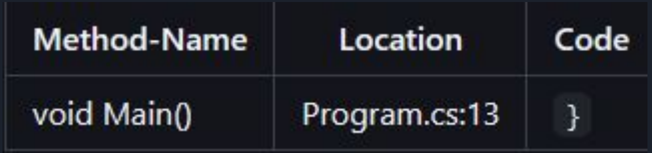
Method-Name	Location	Code
void WriteLine()	???	???
void A()	Program.cs:1	`Console.Write("A");`
void Main()	Program.cs:12	`A();`

6.2



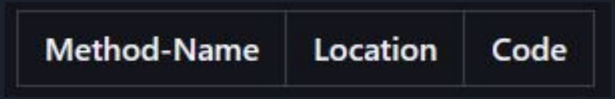
Method-Name	Location	Code
void A()	Program.cs:2	`}`
void Main()	Program.cs:12	`A();`

6.3



Method-Name	Location	Code
void Main()	Program.cs:13	`}`

6.4



Method-Name	Location	Code

Now, the call stack is empty and the application will quit!

Goal 1 - Function Puzzle

- Use the code sample to the right.
- Use the functions of F1, F2, F3 and F4 in the correct order to generate the output: **BAA-BABBA-AAA-AA-BAB**

```
void F1(){
   Console.Write("A"); // A
}

void F2(){
   Console.Write("B"); // BA
   F1();
}

void F3(){
   F1();
   Console.Write("-"); // A-BA
   F2();
}

void F4(){
   Console.Write("-"); // -A
   F1();
}

// Use F1, F2, F3 and F4 in the correct order here.
```

Goal 2 - Triangle... again :D

- Draw the triangle again, but this time using a recursive function instead of **goto**
- Side note: recursive = repetitive / self invoking / calling itself
- Think about the fibonacci sequence: **F(n) = F(n − 1) + F(n − 2)**, where F is the function and n is the number of times it repeats itself.

Goal 3 - Stack a call

- Create a local variable named **timer** of type int and assign 5 to it
- Create a function called **CountDown()**
    - return type **void**
    - without parameters
    - when called:
        - it prints the value of the variable **timer** to the console
        - it reduces the value of the variable **timer** by 1
        - if the times is not negative, the function calls itself again
- Invoke the function **CountDown()**
- How large will the call stack be at its highest?
- What happens after that?
- Run the code, line by line, using the debugger
- Observe the call stack, can you confirm your guesses?