function declaration

```
function nameOfFunction (p1, p1, p3...) {
    ...
}
```

function expression

→ why do we need it ??

→ how to identify if a written func^n is declaration or expression ??

→ if the first valid word is not function in the statement then it is func^ expression.

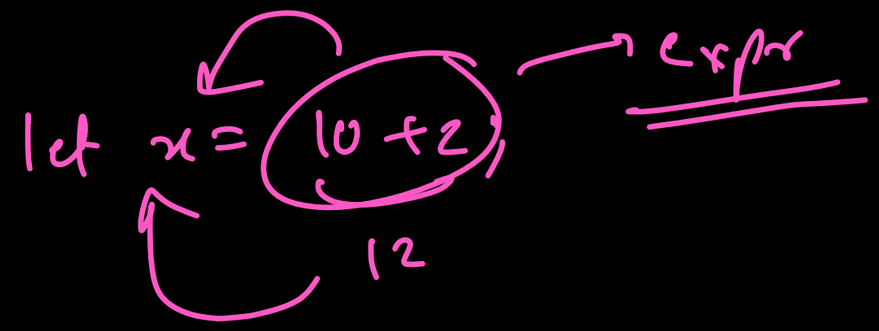
why in JS we consider func^ as first class citizen??

→ you can store a func^ ✓

→ pass func^ as an argument ✓

→ return a func^ from a func^ ✓

} we use func^ expr



Qn Is there any diff apart from Syntax btw func declaration & expression ??

↳ the scoping mechanism is diff.

Types of func^n expression

↳ named func^n expression

↳ anonymous func^n expression

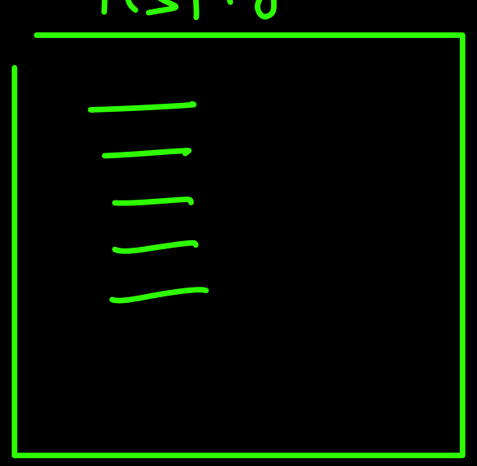
↳ IIFE (immediately invoked function expression)

why named func^n expr is imp??

1. Debugging
2. Recursion
3. Readability

Note → using console.trace we can print the order in which func^n is called.

Memory Architecture



① saw the file (saved in HDD or SSD)

② Run the file

file gets executed in the RAM.

Process

code in a execution stage is called Process

RAM

Process 1

Process 2

Process 3

Lets view a process.

uniquely identify process

actual code that you wrote

Process

→ Process ID

→ Process Name

→ Data Segment

→ Text Segment

→ Call Stack

→ Heap

⋮

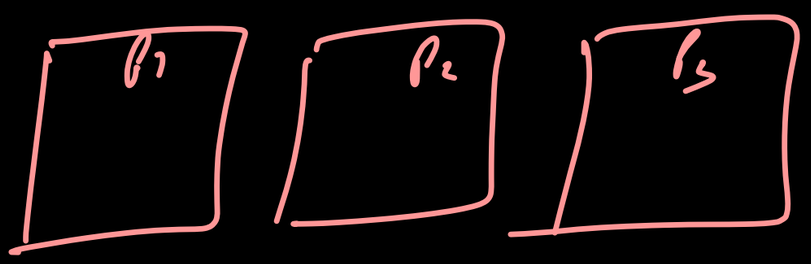
includes global & other static variables

] → VVI

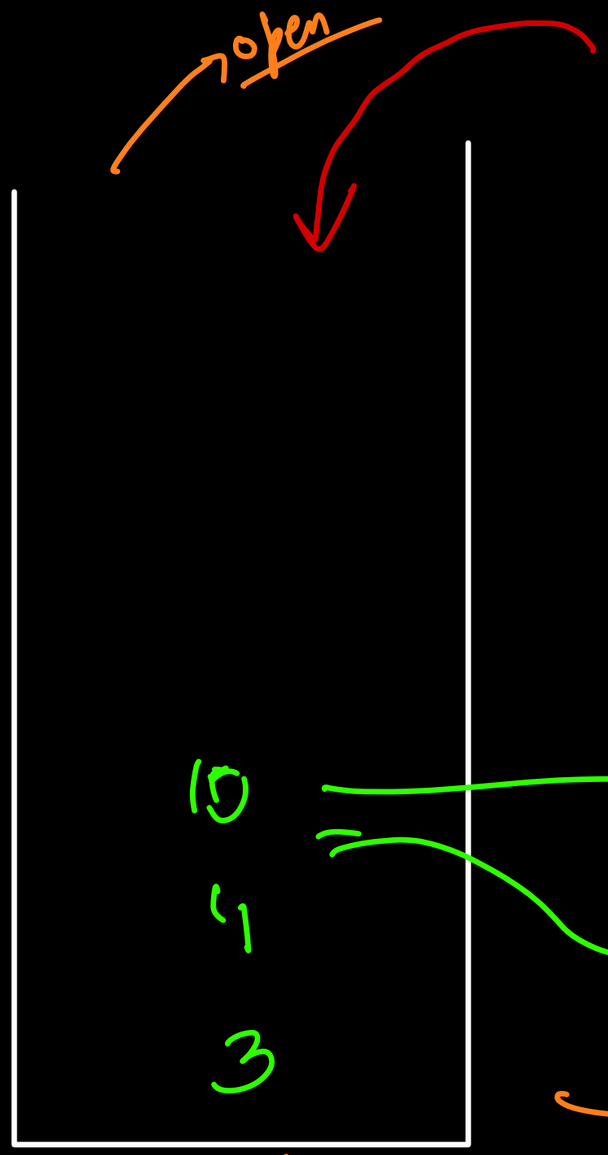
is call stack the only stack present for a process ?? → No. Kernel Stack

Context Switching

Single core processor



what the hell is a stack ??



↓

data structures → way to organise and store data.

FILO → LIFO

↓

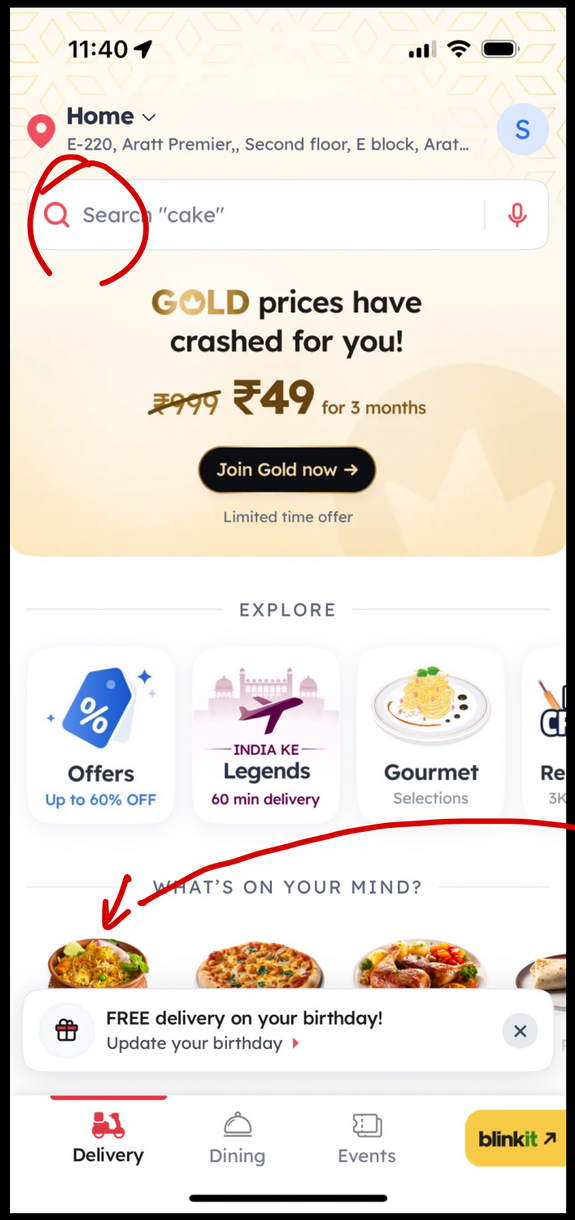
First In Last Out

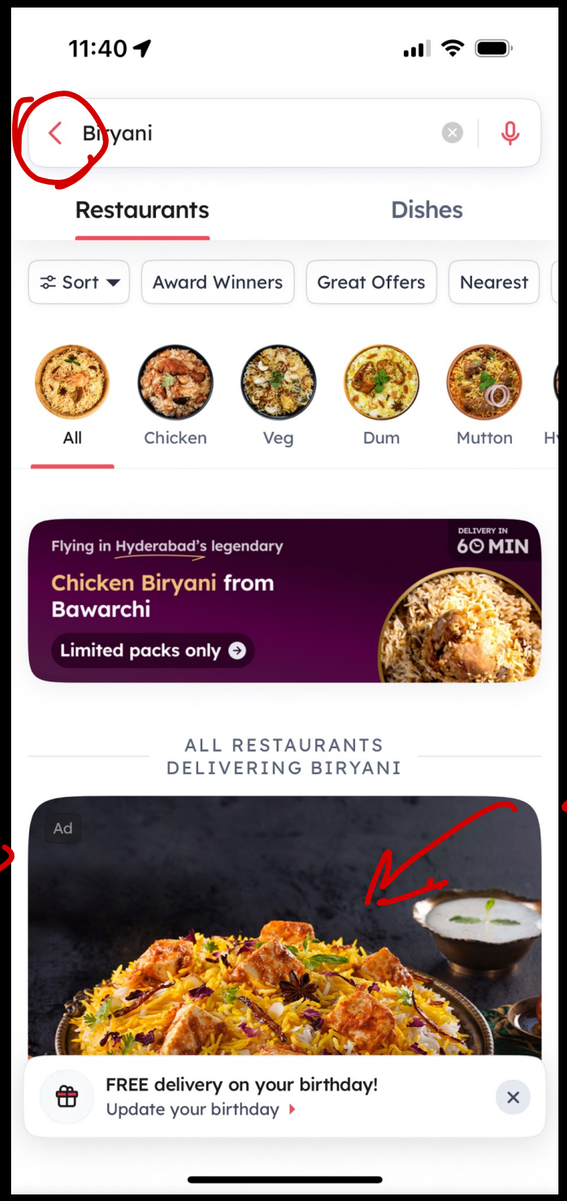
insert → push
remove → pop

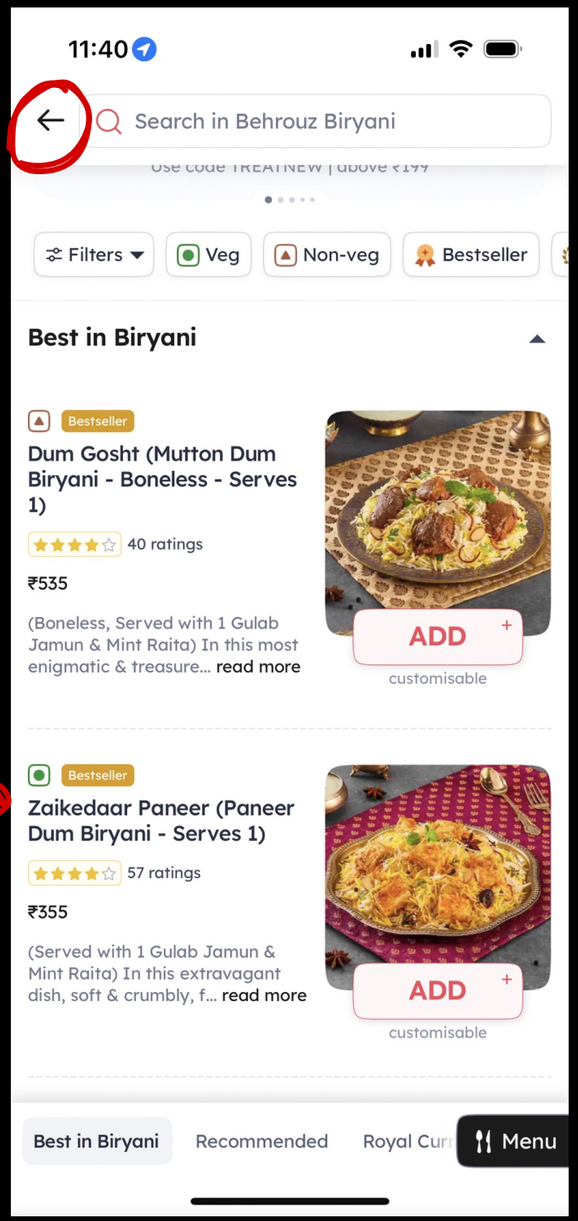
→ insertion is done from top from the open end only the topmost elmt is accessible

→ In one removal, only the topmost is removed.

→ closed.







Call Stack

→ In memory of a process, we have a data structure Call Stack, whenever & wherever we call a function, that function is pushed into the call stack

when we return from a func^ or if func^ ends itself then we pop the func^ from call stack.

the func^ at the top of call stack is the current executing one.

Discrete Maths

Recursion

it is not just a C.S. concept
it is a math concept.

A child couldn't sleep, so her mother told
a story about a little frog,
who couldn't sleep, so the frog's mother told
a story about a little bear,
who couldn't sleep, so bear's mother told
a story about a little weasel
...who fell asleep.
...and the little bear fell asleep;
...and the little frog fell asleep;
...and the child fell asleep. (See: story.cpp)

```
function fA() {
    fB()
}
```

→ fun^n calls some other fun^n

→ In Recursion a function calls itself.

function f (...) {

Recursive func^n

f (...)

}

Factorial → $n! = n \times (n-1) \times (n-2) \times (n-3) \cdots \times 1$

↳ $4! = 4 \times 3 \times 2 \times 1 \rightarrow 24$

$3! = 3 \times 2 \times 1 \rightarrow 6$

$5! = 5 \times \underbrace{4 \times 3 \times 2 \times 1}_{4!}$

$5! = 5 \times 4!$ → 5×24 → 120

Say we have a function 'F' that takes a parameter 'n' and returns $n!$

$F(n) = n \times F(n-1)$

return $n!$

"func" calling itself

Base Case

↓

It is the smallest sub problem for which we already know the ans.

$0! = 1$

$F(5) = 5 \times F(4)$

$5!$

$4!$

```
1) function f(n) {
2)     if(n==0) return 1;
3)     let result = n x f(n-1);
4)     return result;
5) }
```

Base case

6) `console.log(f(3))`

6

first call f(3) & point the value