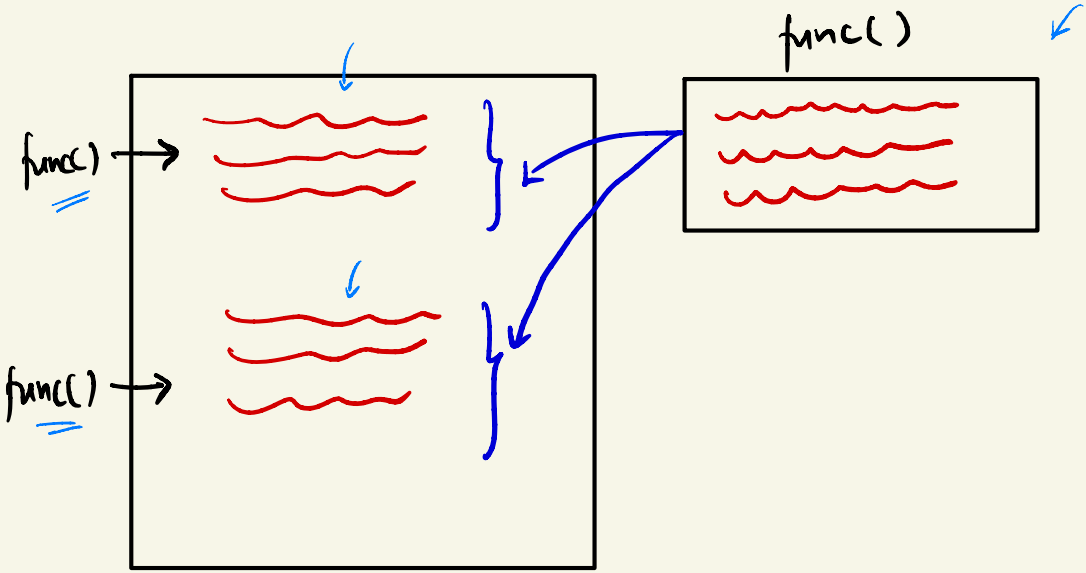
JS functions :- Reusable code that we want to use at different places but dont want to write the code again and again.



Some Terminologies :→

int x;

var x;

var x = 10;

x = 10

- → Functions & Methods
- → Declaration & Definination
- → Argument & parameters
- → Callbacks, Higher order functions
- → Self invoking functions etc

---

args => parameters OR arguments

Create a function

- function myFunc(args)

```
{
    ~~~
    ~~~
    return xyz;
}
```

- const myFunc = (args) =>

```
{
    ~~~
    ~~~
    return xyz;
}
```

- const myfunc = function(){

```
    ~~~
    ~~~
    return xyz;}
```

default parameters :-

```
function add (a, b)
{   return a+b; }

add(3)   // Error
```

→

```
function add (a, b=0)
{   return a+b; }

add(3)   // default value will be used.
```

rest parameters :-

```
function add ( a, ... b)
{   // loop over b to add all numbers in x
    return x+a;
}
```

rest of the parameters

one parameter

arrow functions :→

```
const add = (x,y) =>
            { return x+y; }
```

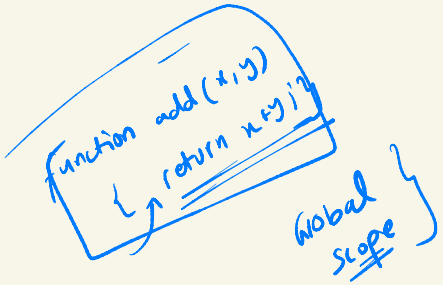
OR

```
const add = (x,y) => x+y;
```
short form

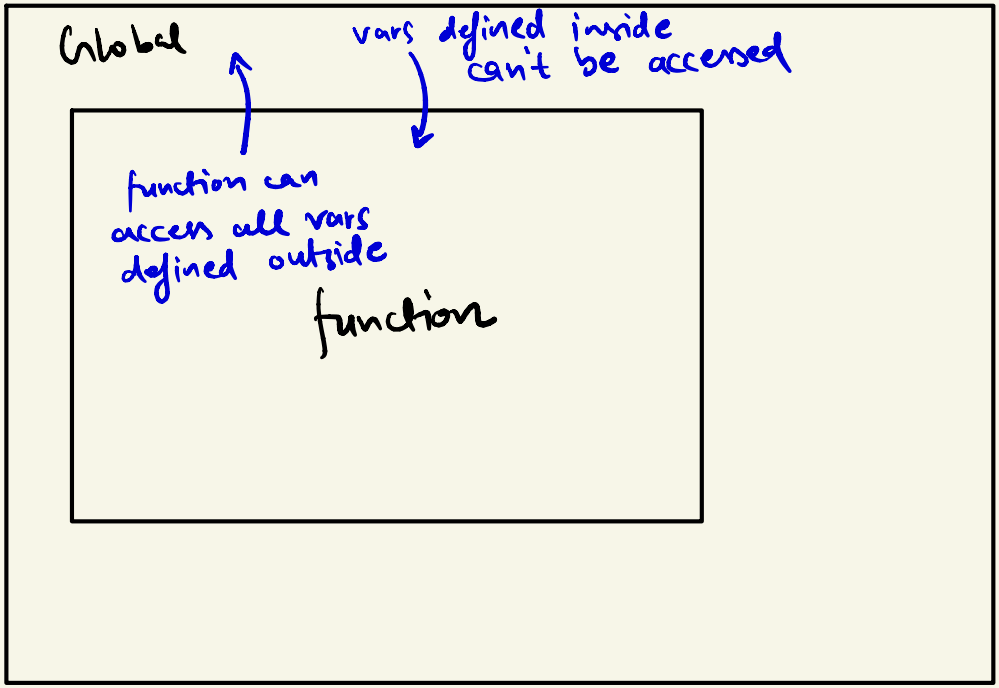
nested functions :→

```
function outer()
{
  console.log("Hello");
  function inner() {
    console.log("Inner");
  }
}
```

1-level nesting



function scope :-



Global ≡ outer function

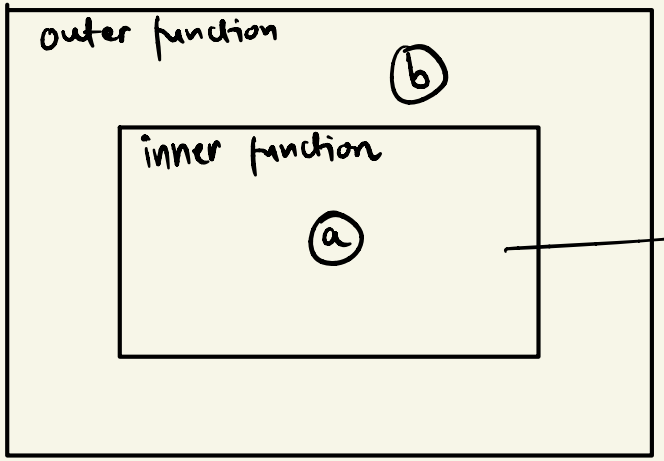
function ≡ inner function

```
function hello()
{
    let x = 10;
    var y = 20;
    const z = 30;
}

hello();
console.log(x, y, z);
// Error, can't access x, y, z.
```

```
{
    let x = 20;
    var y = 30;
    const z = 40;

    function hello()
    {
        console.log(x, y, z);
    }
}
// will run successfully
```

#Javascript closures :->

↓



closure

a nested function that has the context of outer world and independent of the execution of outer func

// inner function can use the outer function vars.

```
function outer(x)  // return inner
{
    function inner(y)
    { return x+y ; }
}
```

```
const outerFunc = outer(10);
outerFun(3);
// will print 13 as 10 will be preserved independent of the execution completion of the outer() func.
```

Callback function :→ we can pass function as parameter to other function.

```
function foo(bar)
{
    bar(); // callback
}
```

```
foo( function(){
    console.log("Hello");
})
```

anonymous function

Higher order function :→

- It takes one or more functions as args
- It might return function as a value

return

```
function foo(bar)
{ bar(); }

foo( function(){
      console.log("bar");})
```

```
function foo()
{ return function(){
      console.log("bar");}
}
```

// [1,2,3].filter(function(){})

passing fun

HOF

similarly map, forEach, filter etc.

pure function :- same input will always give same output.

impure functions :- same input will not guarantee to produce same output

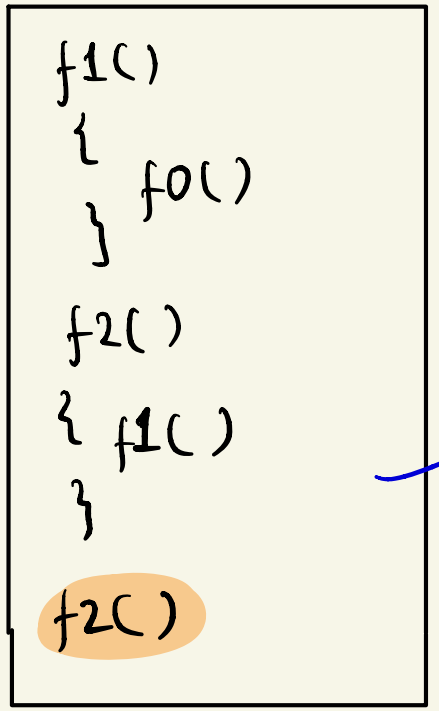
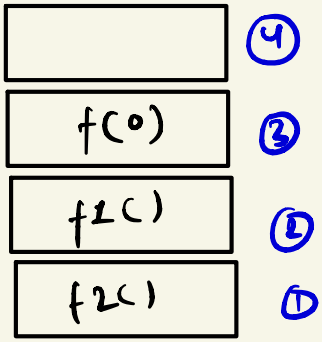
It might happen due to the sideffects.

```
func name(myName)
{ console.log( myName + className); }

className = "Btech"; // sideffect
name("Lovepreet")
```

IIFE :- Immediately invoked function expression

↓

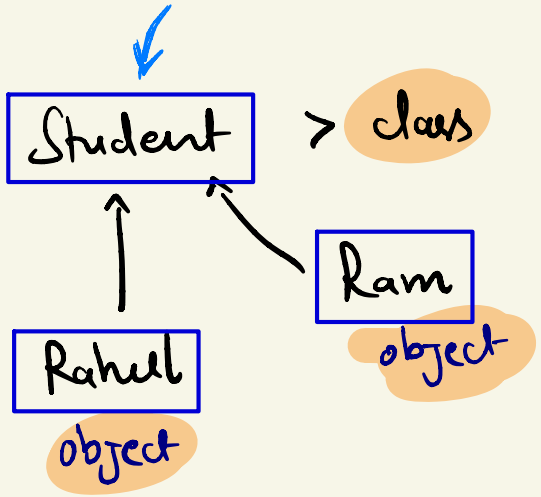
(~~~

~~~)()

⇒

```
( function(){
    console.log("Hello");
})();
```

call Stack !=



code
[Text]

push

pop



call Stack
[In memory]
~~~

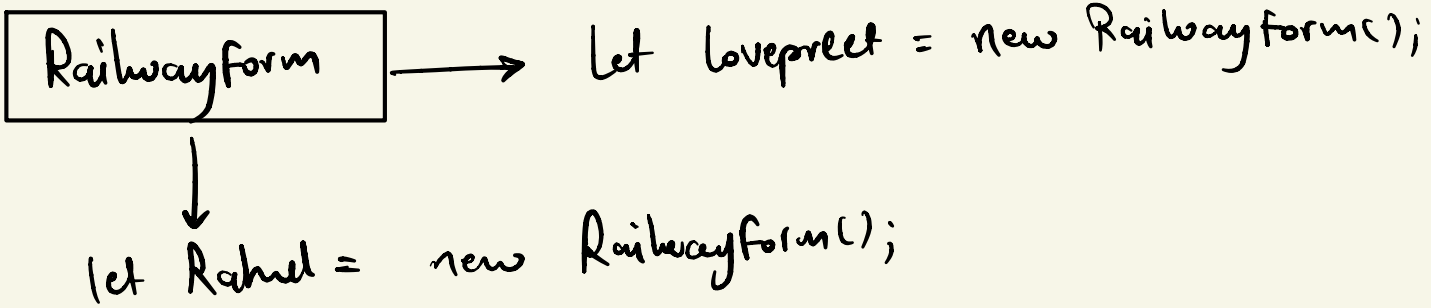
classes and objects :-

classes → Template to create something

object → instance / print made using the template given.




```
class RailwayForm {
    submit()
    { alert("Form submitted"); }

    cancel()
    { alert("Cancelled form"); }

}
```



Lovepreet.submit();
Rahul.submit();

using OOPS :-

↳ makes code more readable

↳ our own datatypes can be made

↳ Real life things makes more sense while using OOPS.

```
class RailwayForm {
    ~~~~
    ~~~~

    fillForm ( givenName )
    {
        this.name = givenName ;
        console.log ("form by " + this.name);
    }
}
```

this.name → variable associated with class

↓

belongs to the current instance of the class

use Constructor ↦

```
constructor ( --- )
{ ~~~ }
```

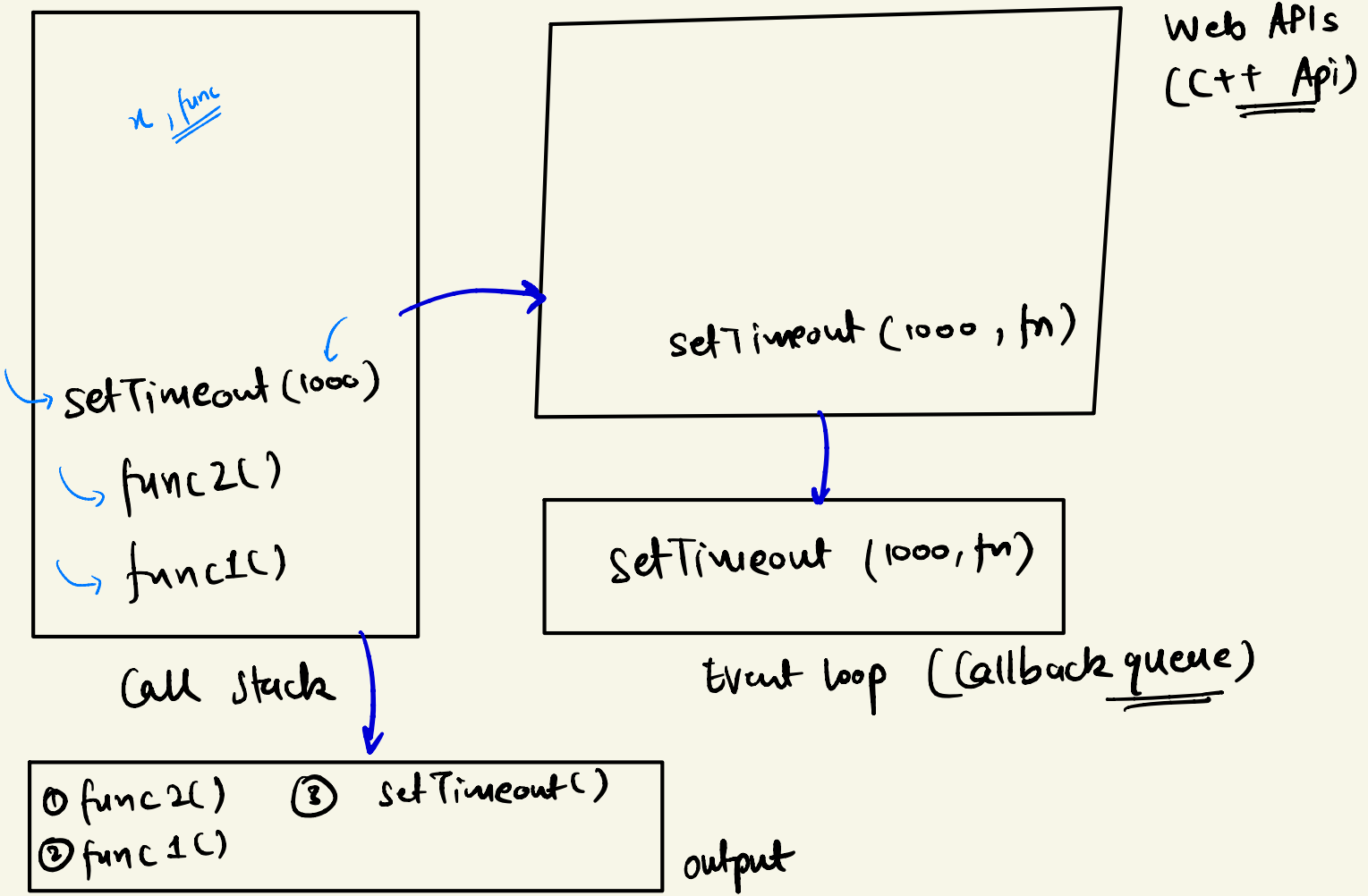
* Constructor run at the start of object creation using the "new" keyword.

private and public :- use # infront of the name for private and fields are public by default.

JS

Callbacks, Promises, Async await & Event loops :-

⊙ some WebApis → setTimeout, setInterval, click, scroll, click events etc.


Web APIs
(C++ Api)
setTimeout (1000, fn)
setTimeout (1000, fn)
event loop (Callback queue)
setTimeout (1000)
func2()
func1()
Call stack
① func2()
② func1()
③ setTimeout()
output

Promises :→

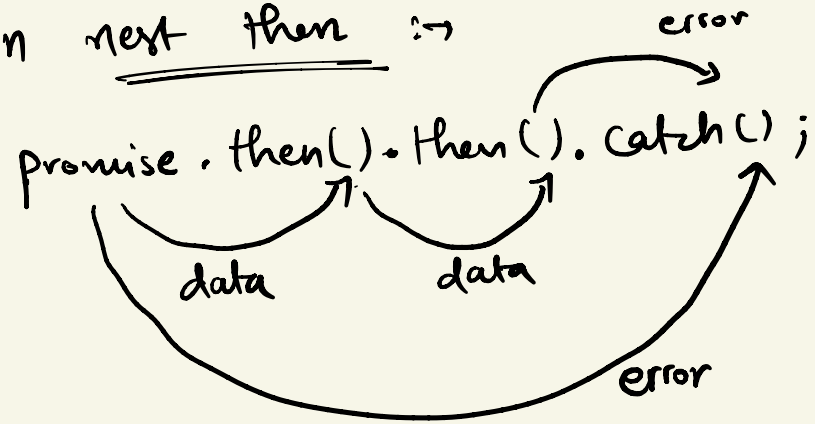
```
function callbackHell()
{
    setTimeout( () => {
        const data = { user : "Love" };
        console.log(data);
        setTimeout( () => {
            console.log("Data updated") }, 1000)
        }, 500)
    }
callbackHell();
```

- Instead of using Timeouts where functions running might take sometime & will return the data in future we can use Promises

```
const promise = new Promise ( (resolve , reject) =>
{     resolve (" Hey There you no" ); } )
// reject (" Server down")

promise . then ( (data) => { } ). catch ( (err) => { } )
```

Note :-

Can next then :-



```
const promise = new Promise ( (resolve, reject) =>
{   setTimeout ( () => resolve ("data from server"),
                                    2000) } )

promise.then ( (res) => {
        console.log (res) } )
```

fetch api :- used to get, post, put, delete data or to make requests to the server and returns promises.

Note # Try placeholder fake api for demo.

async-await :- enhanced promises syntax

```
async function getData()
  { const response = await fetch("  ");
    console.log(response);
  }
getData();
```

Homework :- See the meaning of Promises.all(---).then()