Fifth Pillar of JS

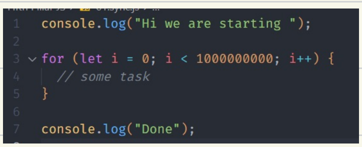
Async Programming with JS :

JS is sync in nature. (sync ⇒ line by line)
JS is single threaded.

All of this only applicable if we execute valid ECAMScript code which is given by standard.

For e.g. for loop

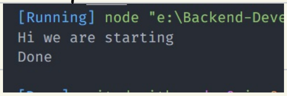
Code



```
console.log("Hi we are starting ");

for (let i = 0; i < 1000000000; i++) {
  // some task
}

console.log("Done");
```

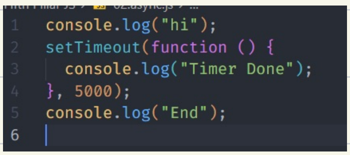
Sync Code here it's wait for for-loop.

output

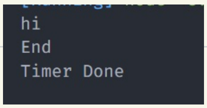


```
[Running] node "e:\Backend-Deve
Hi we are starting
Done
```

Async Code

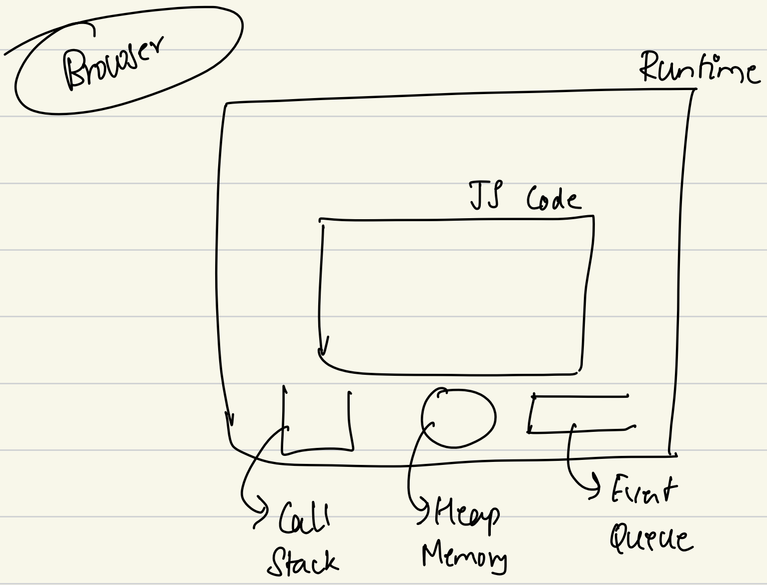


```
console.log("hi");
setTimeout(function () {
  console.log("Timer Done");
}, 5000);
console.log("End");

```



```
hi
End
Timer Done
```

here it doesn't wait for setTimeOut

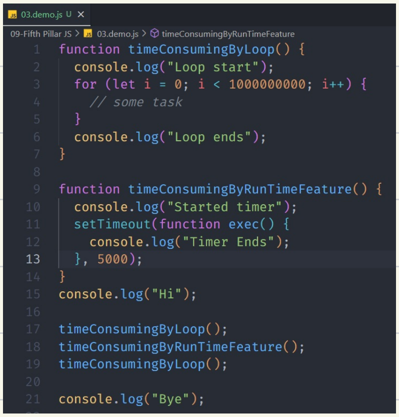


Runtime : — Inside this you can run JS code.
— JS + Runtime = Powerful

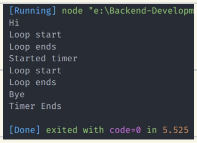
# SetTimeout given by Node not by javascript
Non-native JS ⇒ Async Nature

# for loop given by javascript so it's sync nature
Native JS ⇒ Sync Nature

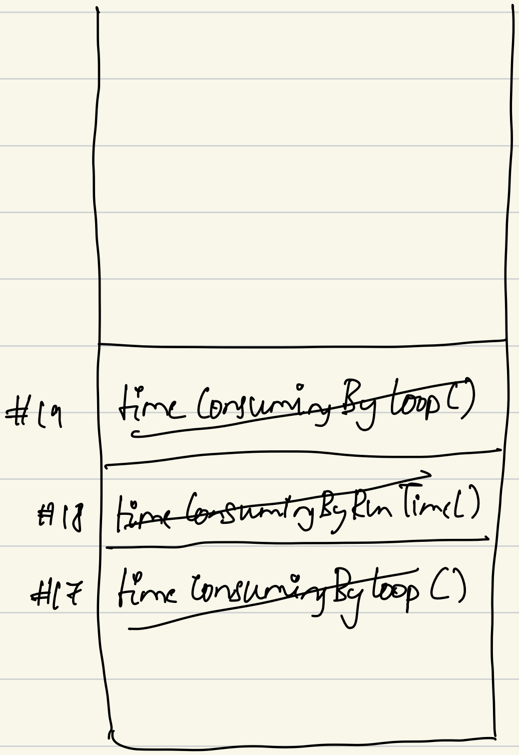
Code



```
03.demo.js U
09-Fifth Pillar JS > 03.demo.js > timeConsumingByRunTimeFeature
function timeConsumingByLoop() {
  console.log("Loop start");
  for (let i = 0; i < 1000000000; i++) {
    // some task
  }
  console.log("Loop ends");
}

function timeConsumingByRunTimeFeature() {
  console.log("Started timer");
  setTimeout(function exec() {
    console.log("Timer Ends");
  }, 5000);
}
console.log("Hi");

timeConsumingByLoop();
timeConsumingByRunTimeFeature();
timeConsumingByLoop();

console.log("Bye");
```



```
[Running] node "e:\Backend-Developm
Hi
Loop start
Loop ends
Started timer
Loop start
Loop ends
Bye
Timer Ends

[Done] exited with code=0 in 5.525
```



#19 ~~time ConsumingBy loop()~~
#18 ~~time ConsumingByRun Time()~~
#17 ~~time ConsumingBy loop ()~~

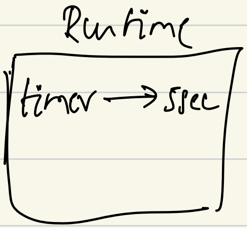
O/P

#15 Hi

#2 loop starts

#6 loop end

#10 Started timer



#2 loop start

Now there is two case

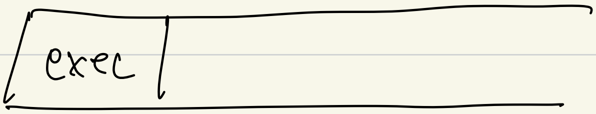
1 → for loop ends less than 5 sec

2 ⟶ for loop takes more than 5 sec

let's takes 1st case

After this timer get over

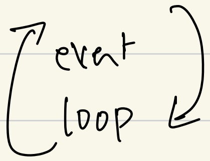
let think for loop takes 10 sec
Starting the timer and loop 1 sec passed
still the loop ends in 9 sec and the
timer ends in 5 sec
Still the timer wait for the for-loop to
end then only timer executes.
We never PAUSE the sync code of execution
Runtime ⟶ Not a native code
till then runtime goes to event queue.



event queue

#6 Loop ends
#21 Bye

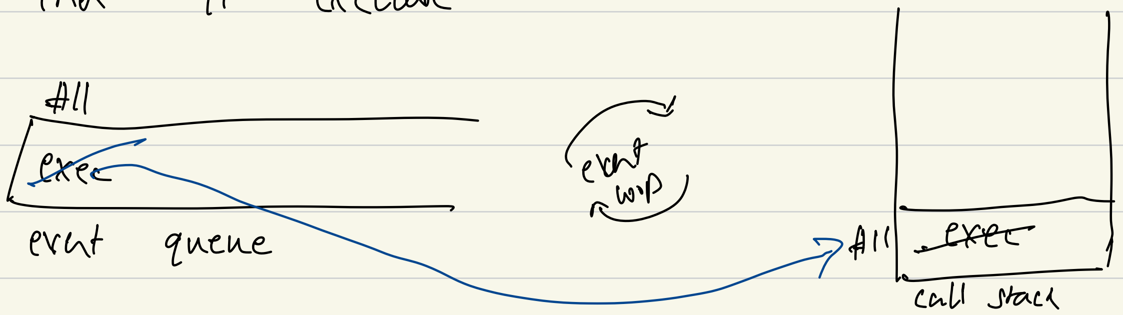
Apart from event queue there is event loop



Event loop keeps on checking whether the call stack is empty or not and no global code is left.

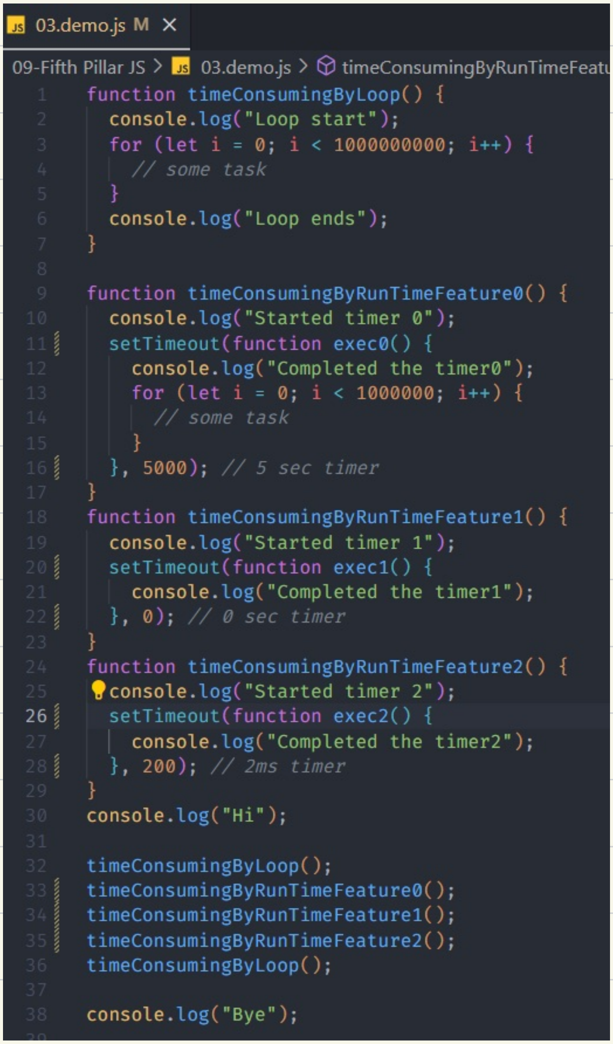
Event queue code doesn't execute immediately, it only execute if and only if there is nothing in call stack and no global code is left. (f(), print)

Now the call stack & global code is empty so the event loop will take one call back from event queue and move to the call stack then it execute.



#12 Timer Ends

Example 2 :

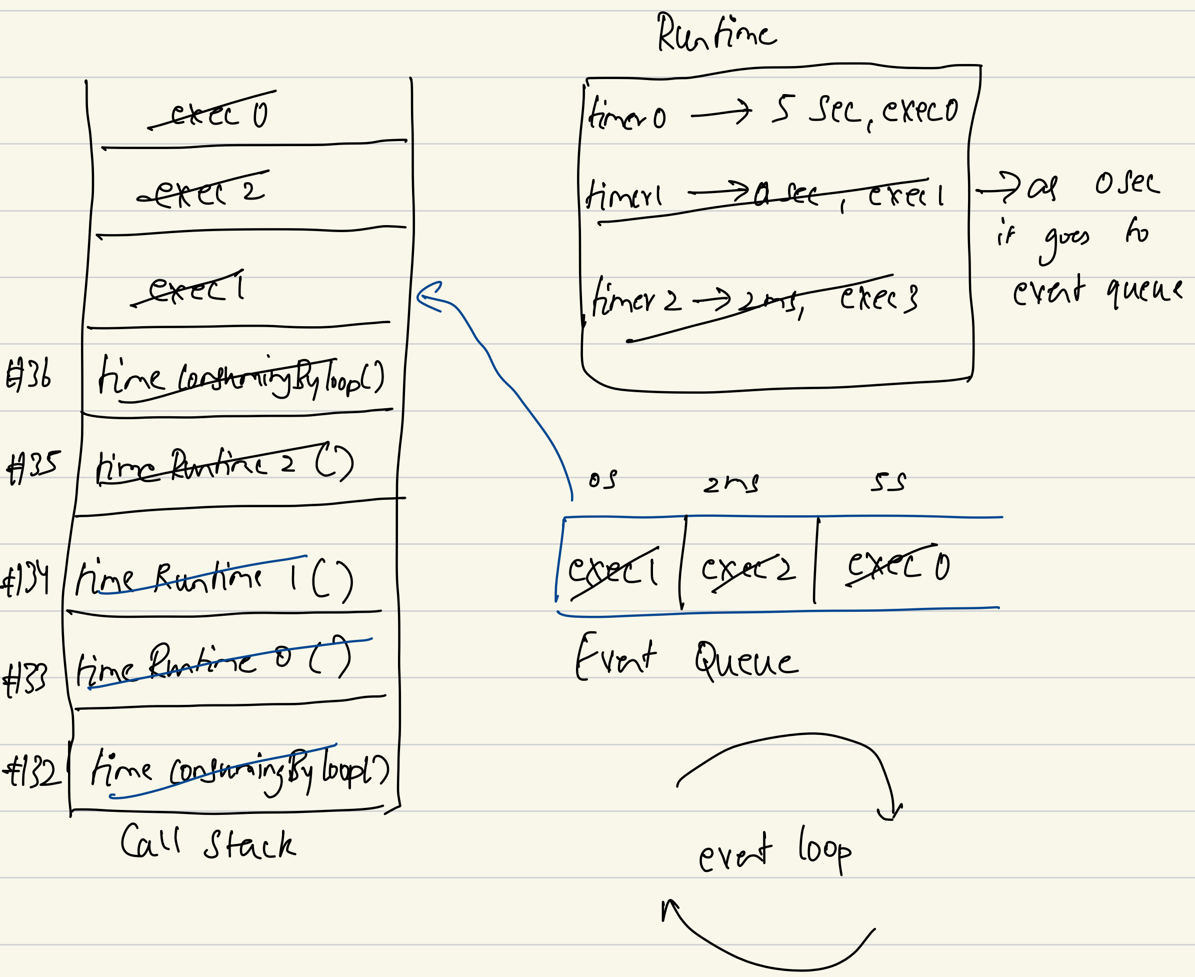


03.demo.js M ×

09-Fifth Pillar JS > 03.demo.js > timeConsumingByRunTimeFeatu

```javascript
function timeConsumingByLoop() {
  console.log("Loop start");
  for (let i = 0; i < 1000000000; i++) {
    // some task
  }
  console.log("Loop ends");
}

function timeConsumingByRunTimeFeature0() {
  console.log("Started timer 0");
  setTimeout(function exec0() {
    console.log("Completed the timer0");
    for (let i = 0; i < 1000000; i++) {
      // some task
    }
  }, 5000); // 5 sec timer
}
function timeConsumingByRunTimeFeature1() {
  console.log("Started timer 1");
  setTimeout(function exec1() {
    console.log("Completed the timer1");
  }, 0); // 0 sec timer
}
function timeConsumingByRunTimeFeature2() {
  console.log("Started timer 2");
  setTimeout(function exec2() {
    console.log("Completed the timer2");
  }, 200); // 2ms timer
}
console.log("Hi");

timeConsumingByLoop();
timeConsumingByRunTimeFeature0();
timeConsumingByRunTimeFeature1();
timeConsumingByRunTimeFeature2();
timeConsumingByLoop();

console.log("Bye");
```

Dry Run


Runtime
timer 0 → 5 sec, exec 0
~~timer1 → 0 sec, exec 1~~
→ as 0 sec it goes to event queue
~~timer 2 → 2ms, exec 3~~
~~exec 0~~
~~exec 2~~
~~exec 1~~
#36 ~~time ConsumingByLoop()~~
#35 ~~time Runtime 2 ()~~
#34 ~~time Runtime 1 ()~~
#33 ~~time Runtime 0 ()~~
#32 ~~time ConsumingByLoop()~~
Call Stack
0s
2ms
5s
~~exec 1~~
~~exec 2~~
~~exec 0~~
Event Queue
event loop

Output

#30 Hi

#2 loop start

#6 loop end

#10 Started timer0

#19 Started timer 1

#25 Started timer 2

#2 loop start

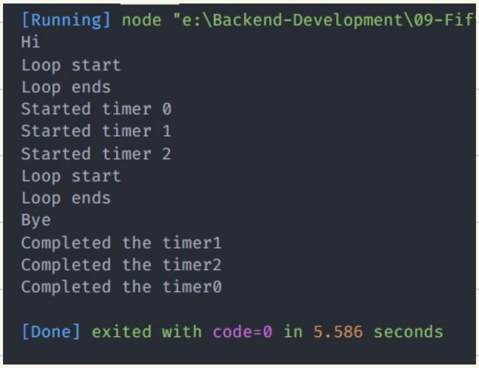
#6 loop end

#38 Bye

#21 completed timer 1

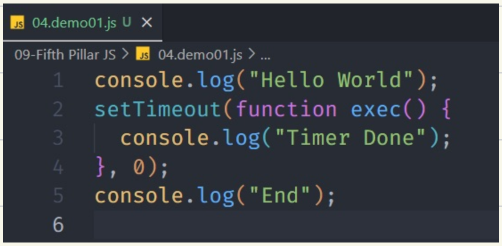
#27 completed timer 2

#12 Completed timer 0



```
[Running] node "e:\Backend-Development\09-Fif
Hi
Loop start
Loop ends
Started timer 0
Started timer 1
Started timer 2
Loop start
Loop ends
Bye
Completed the timer1
Completed the timer2
Completed the timer0

[Done] exited with code=0 in 5.586 seconds
```

Example 3 :



04.demo01.js U ×

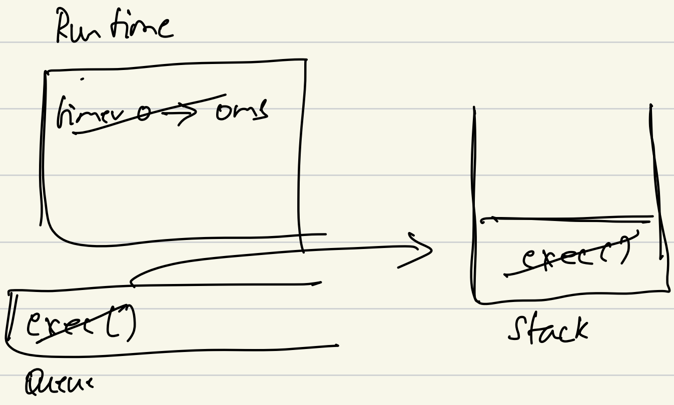
09-Fifth Pillar JS > 04.demo01.js > ...

```javascript
console.log("Hello World");
setTimeout(function exec() {
  console.log("Timer Done");
}, 0);
console.log("End");
```

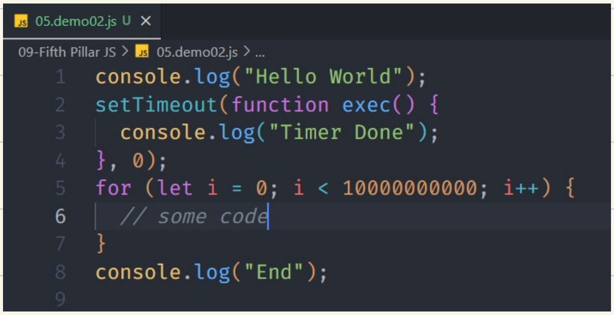
#1 Hello World

#5 End

#3 Timer Done



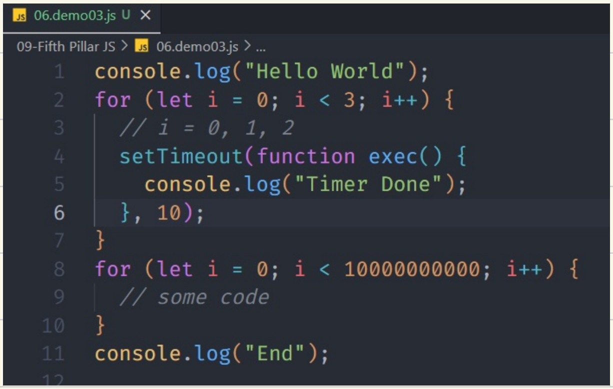
Example 4 :


05.demo02.js U ×
09-Fifth Pillar JS > 05.demo02.js > ...

```javascript
console.log("Hello World");
setTimeout(function exec() {
  console.log("Timer Done");
}, 0);
for (let i = 0; i < 10000000000; i++) {
  // some code
}
console.log("End");
```

Hello World
End
Timer End

Example 5 :


06.demo03.js U ×
09-Fifth Pillar JS > 06.demo03.js > ...

```javascript
console.log("Hello World");
for (let i = 0; i < 3; i++) {
  // i = 0, 1, 2
  setTimeout(function exec() {
    console.log("Timer Done");
  }, 10);
}
for (let i = 0; i < 10000000000; i++) {
  // some code
}
console.log("End");
```

Hello World
End
Timer Done
Timer Done
Timer Done

is console.log also an async feature ?

dependent on how runtime handle it.
(node runtime)

Console.log() → print $\overset{\text{stdout}}{}$ with new line
n

Whenever answering for the output you should say considering console.log() → work Sync.

Set Interval :

```
setInterval( function () {
    console.log (" another one")
}, 1000)
```

⇒ You get unique id. // 1
After every 1 sec it will print to stop it you can refersh.

↳ you can store this unique id, with the id you can stop the interval

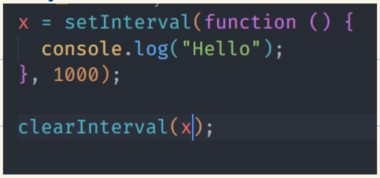
```
x = setInterval( function () {
        console.log (" another one")
    }, 1000)
```

clear Interval (x); // it will clear

In browser set Interval return id (number) (chrome)

In node set Interval return an object

Syntax



```
x = setInterval(function () {
  console.log("Hello");
}, 1000);

clearInterval(x);
```

Behaviour is same but return type is different based on runtime / browser.