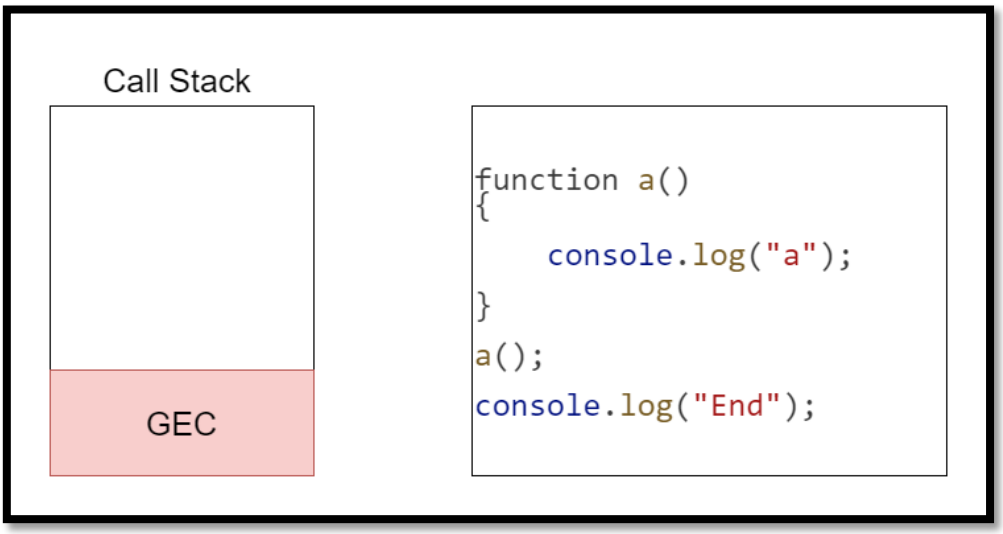
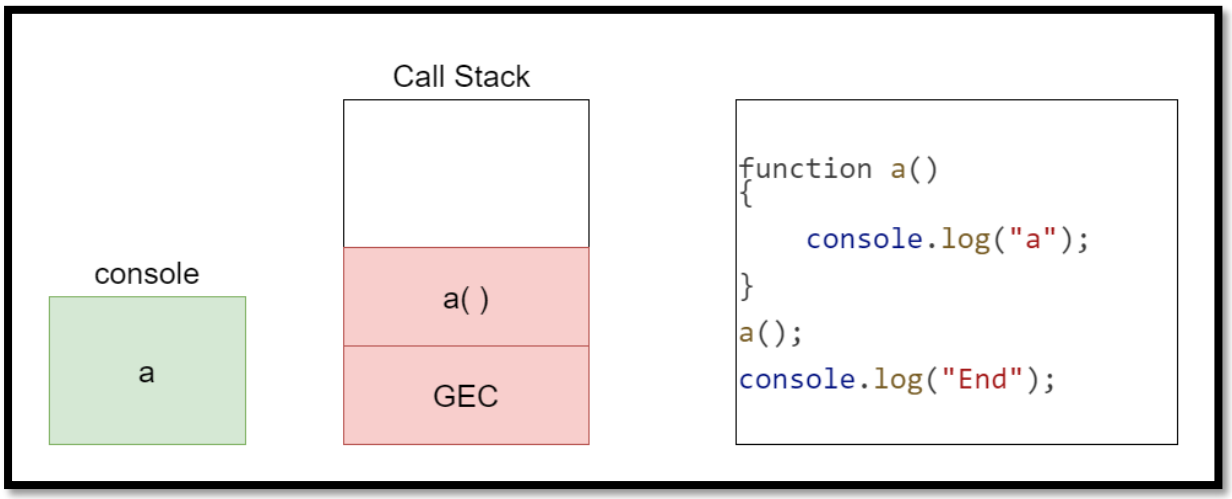
Event Loop

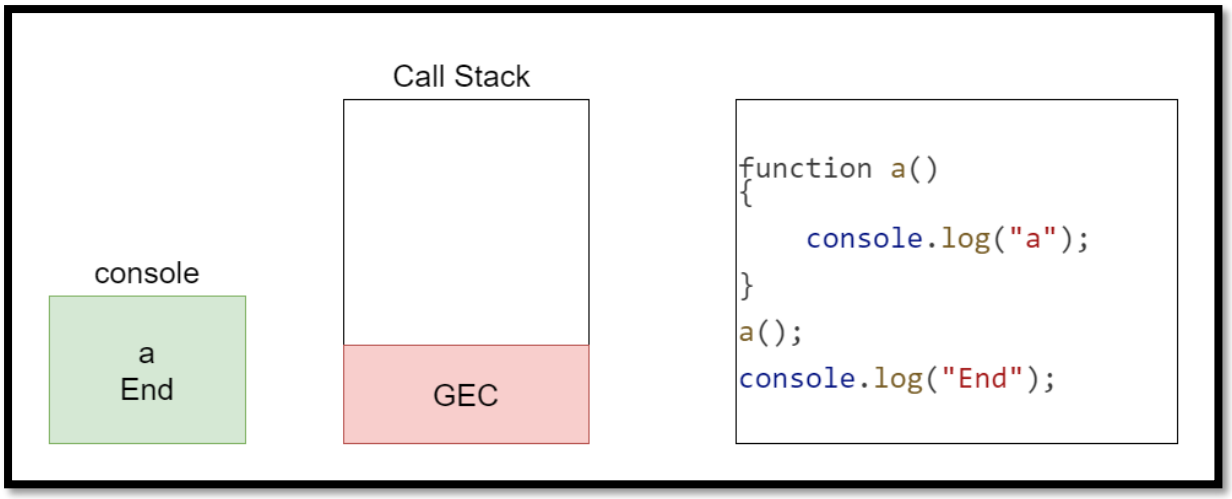
Whenever any JavaScript code is run, a global execution context is created. Global execution context is pushed inside the call stack.

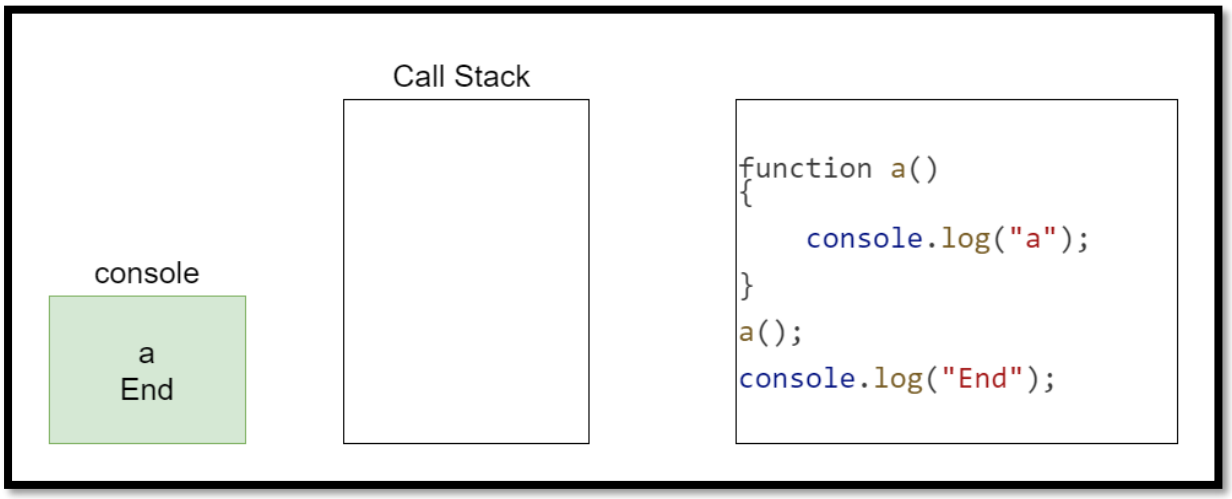


Now, the code will run line by line in global execution context.

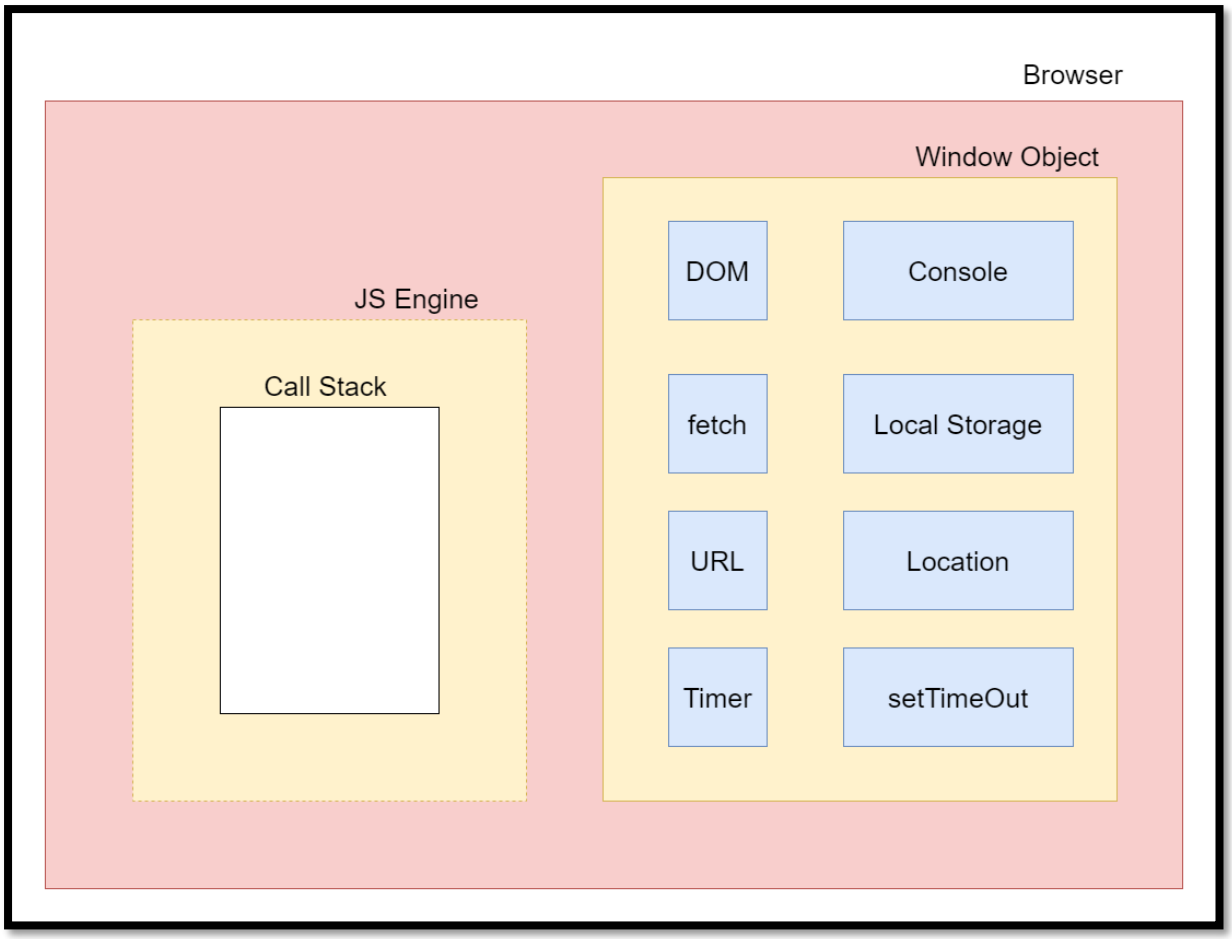


In the execution phase a() will be pushed in the call stack.

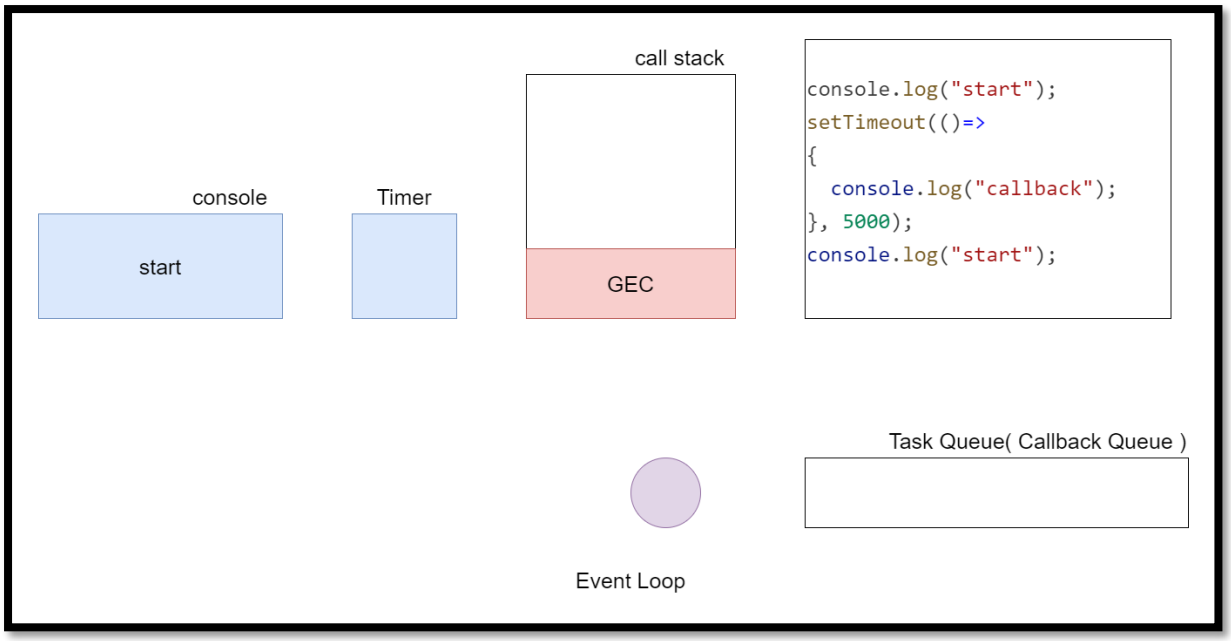


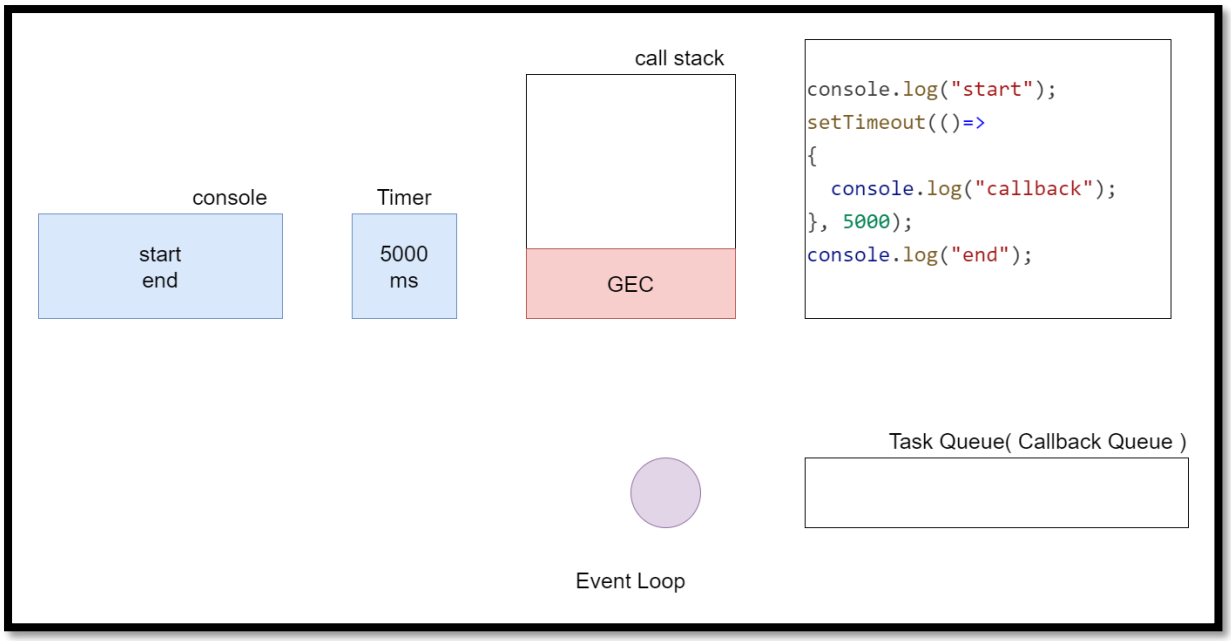


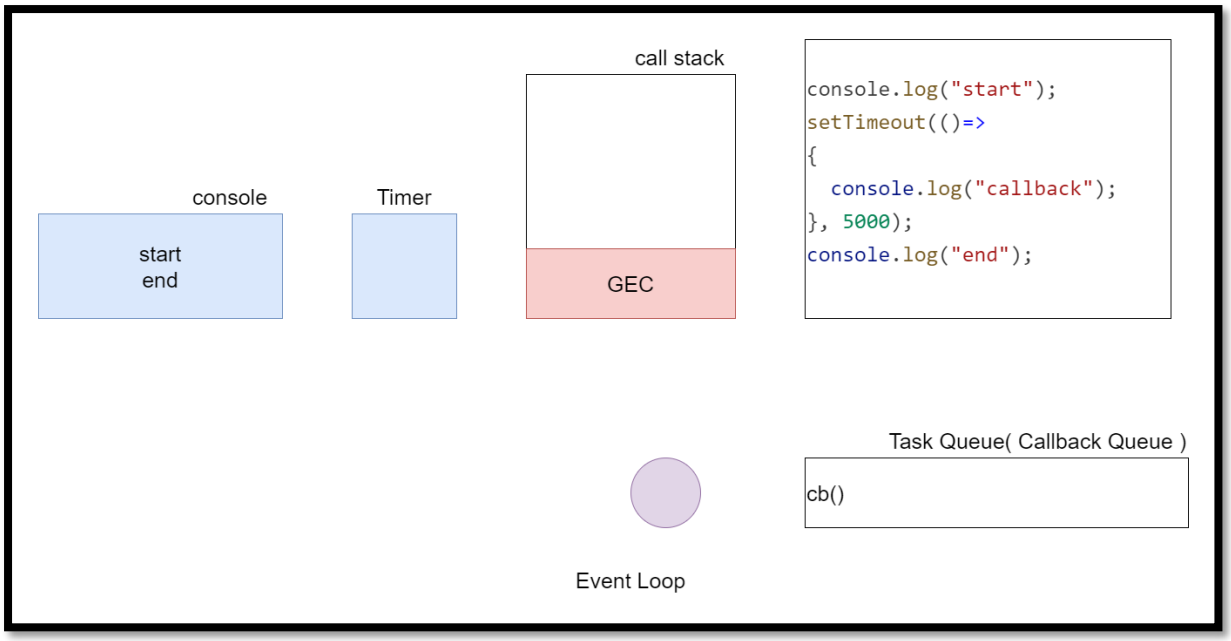
The main job of the call stack is to execute whatever comes inside it.



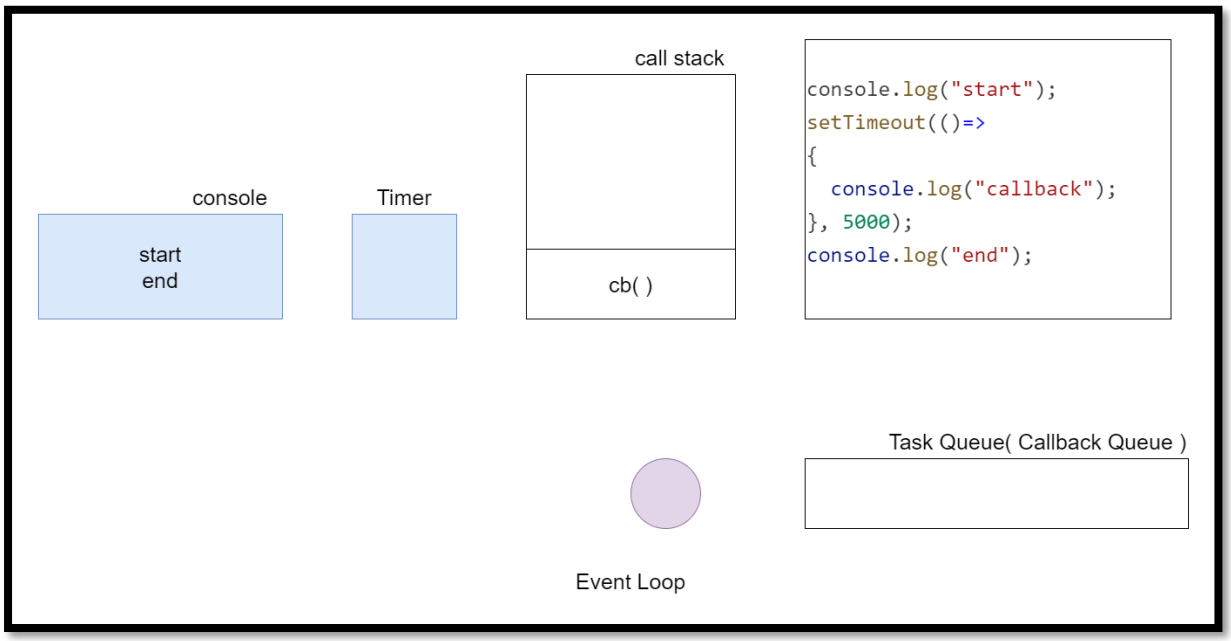
Browser gives access of global object (window object) in call stack in global execution context.


call stack
console
Timer
start
GEC
console.log("start");
setTimeout(()=>
{
console.log("callback");
}, 5000);
console.log("start");
Task Queue(Callback Queue)
Event Loop

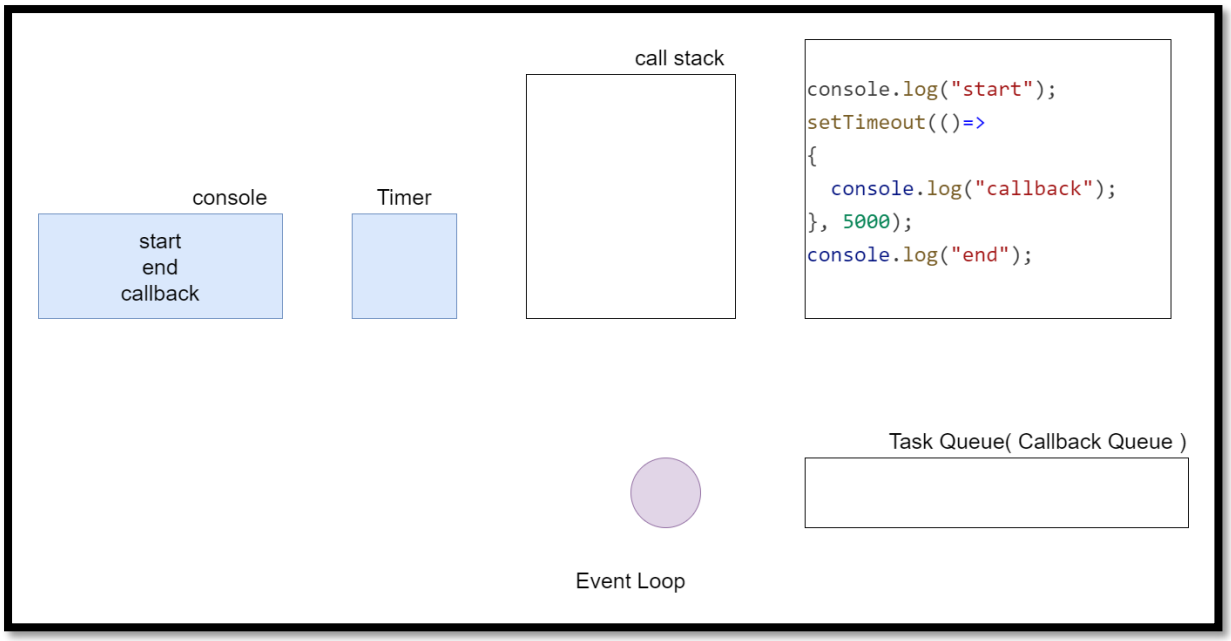

call stack
console
Timer
start
end
5000
ms
GEC
console.log("start");
setTimeout(()=>
{
console.log("callback");
}, 5000);
console.log("end");
Task Queue(Callback Queue)
Event Loop



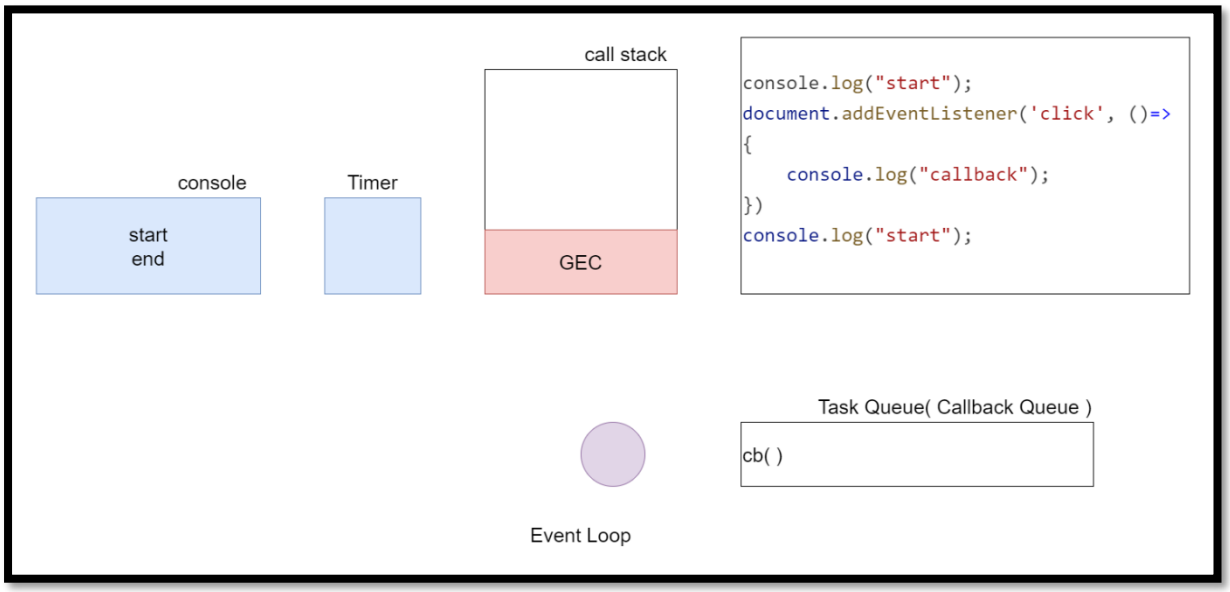
When the timer expires, the callback goes into task queue(callback queue).

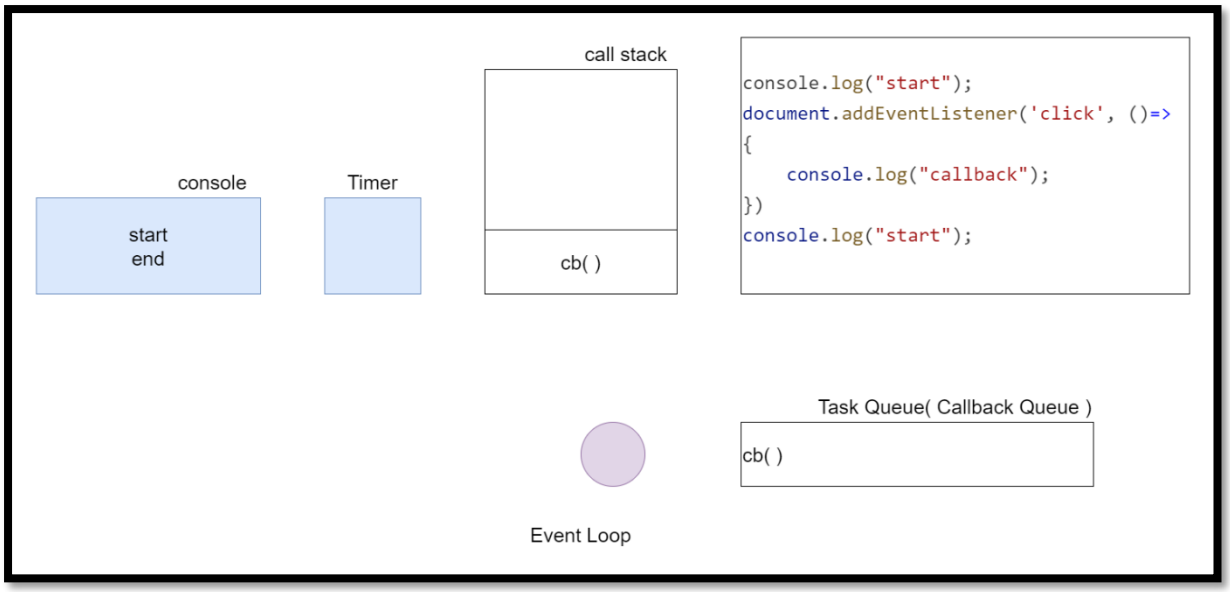


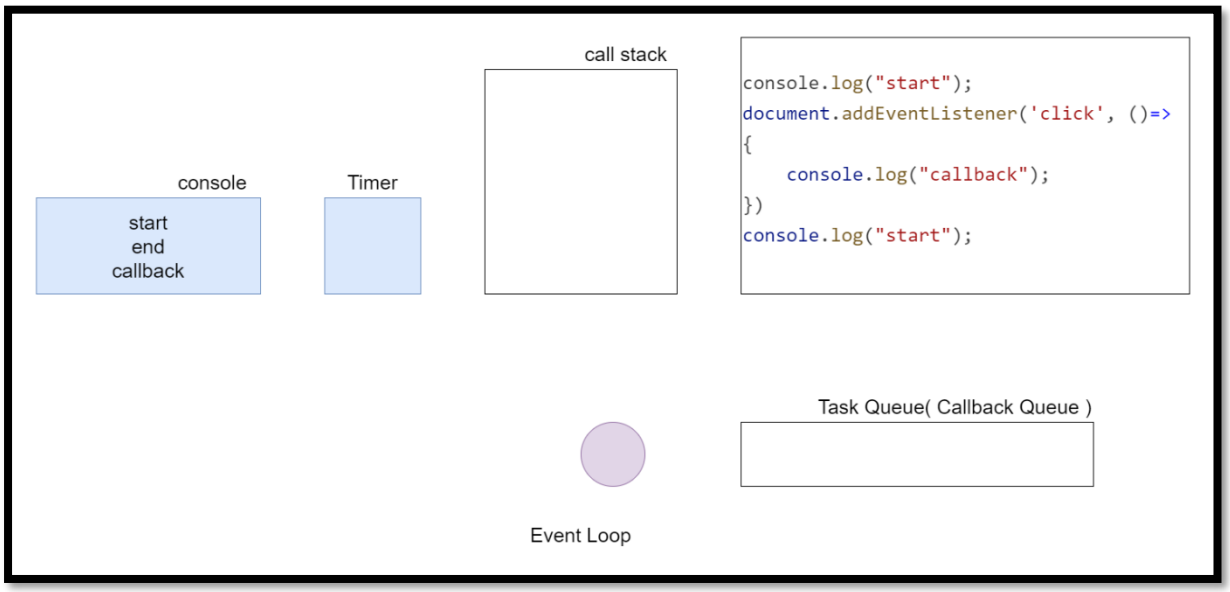
The event loop transfers the callback into call stack from task queue.



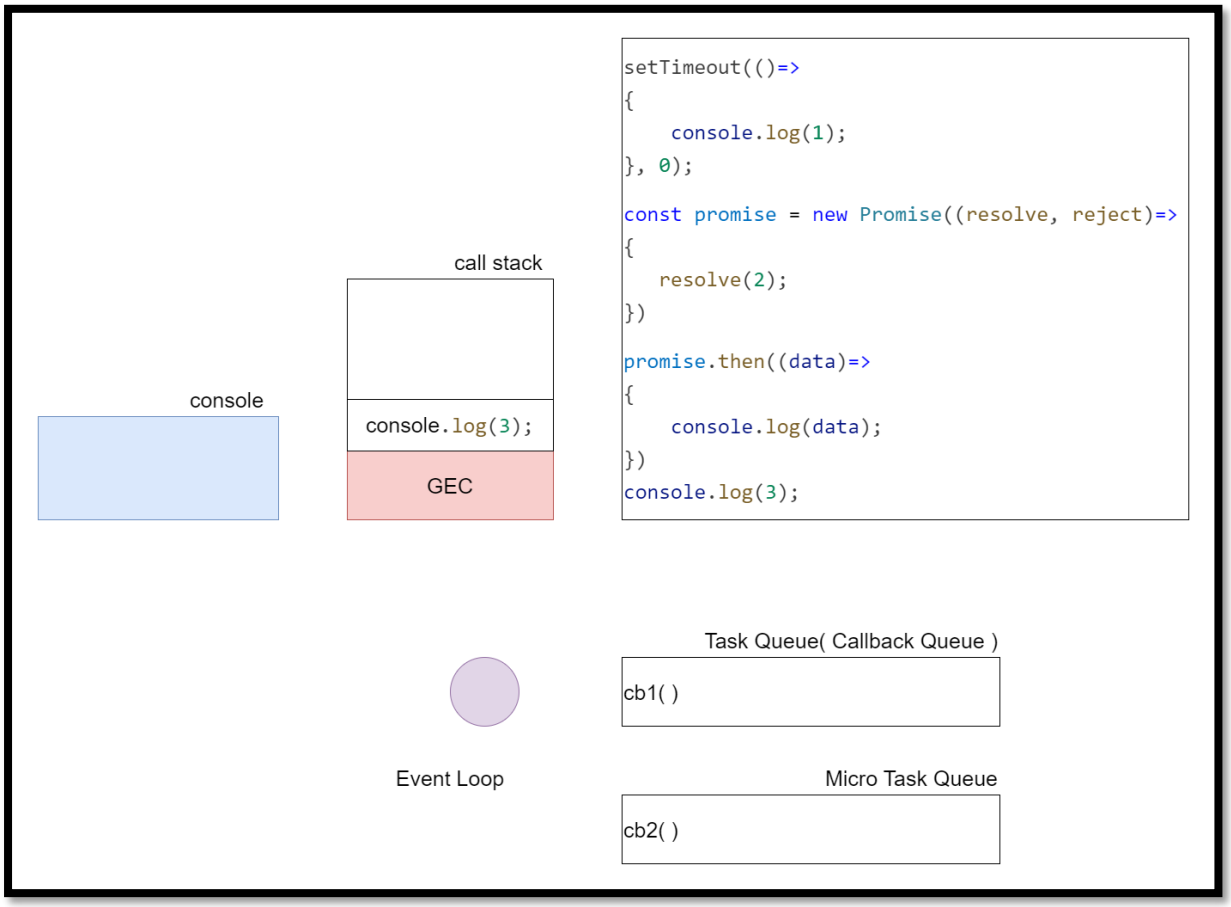
DOM and Event Loop

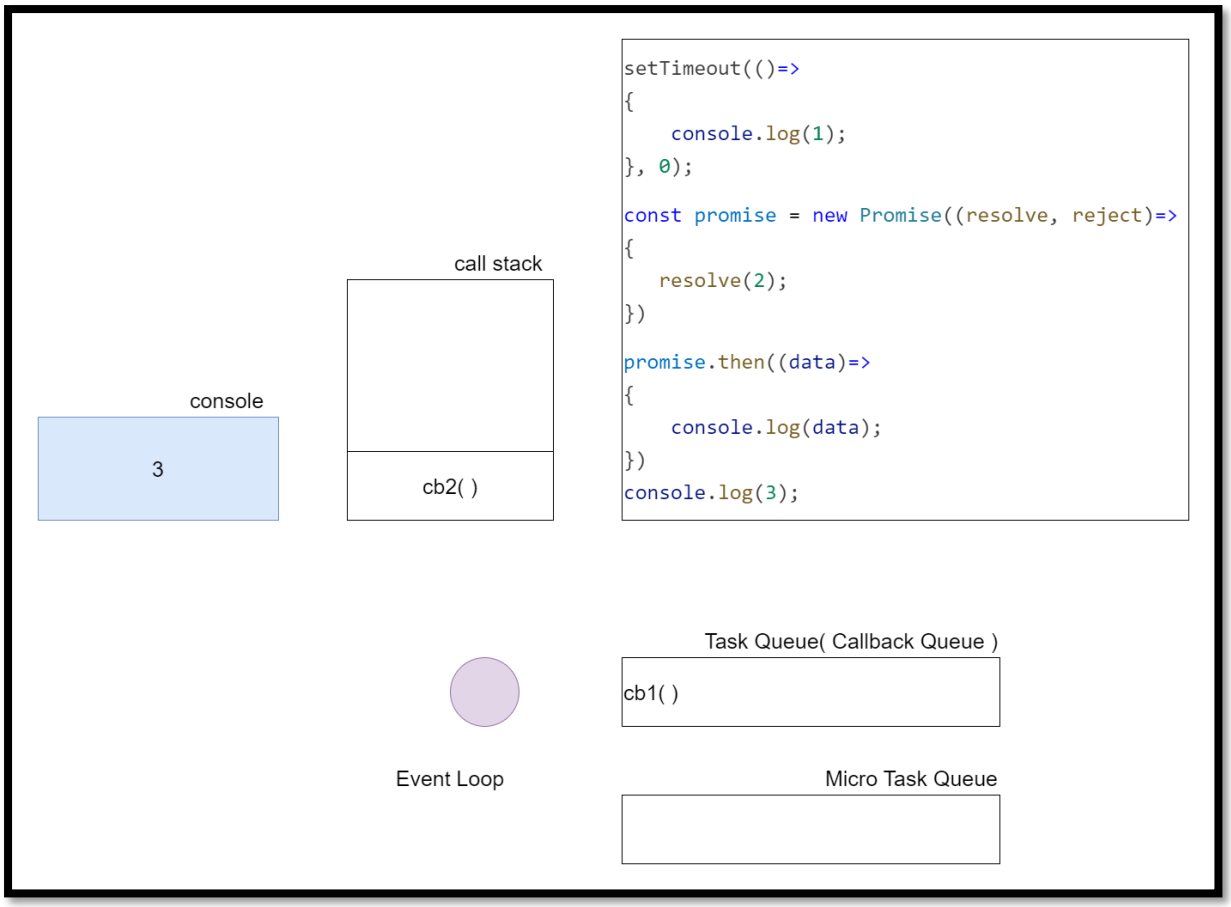


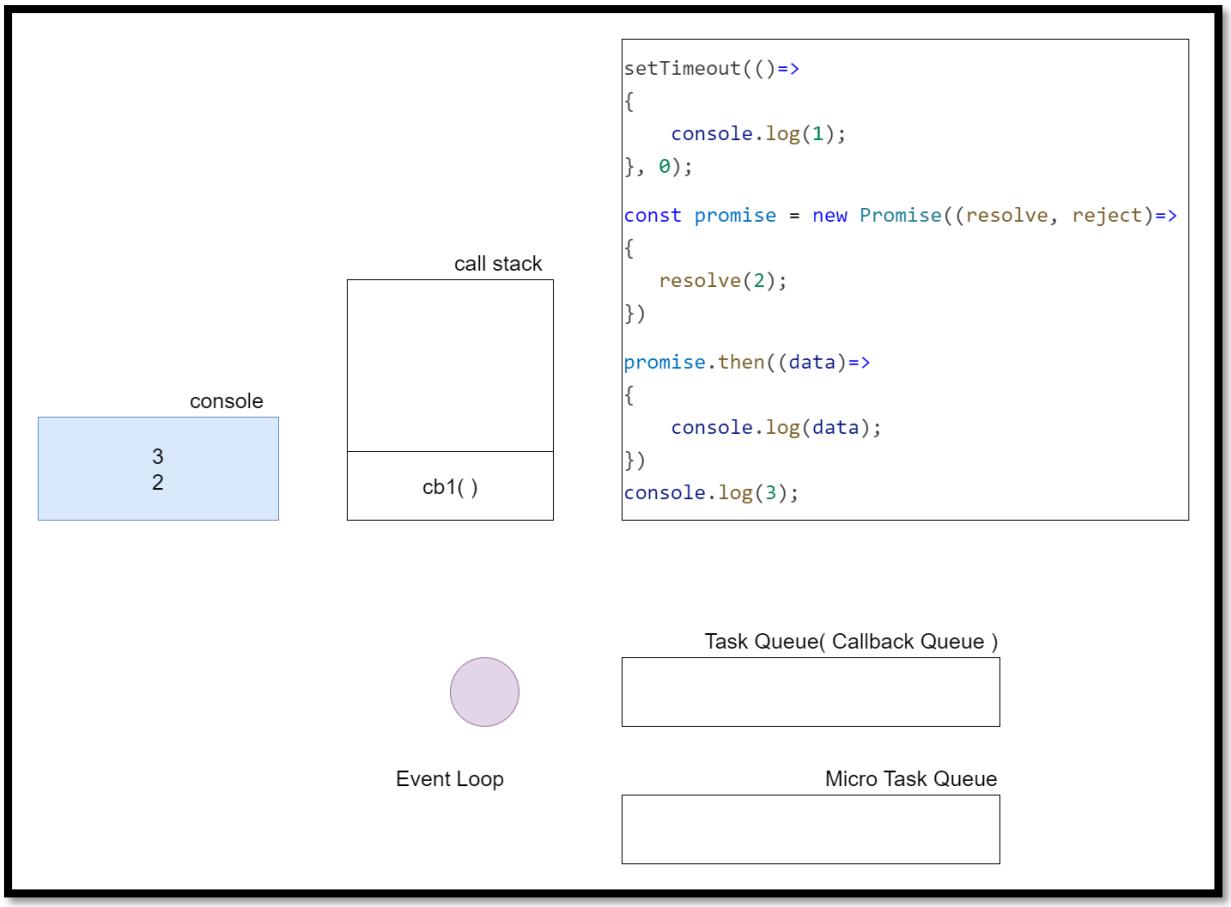

call stack
console
Timer
start
end
cb( )
console.log("start");
document.addEventListener('click', ()=>
{
console.log("callback");
})
console.log("start");
Task Queue(Callback Queue)
cb( )
Event Loop

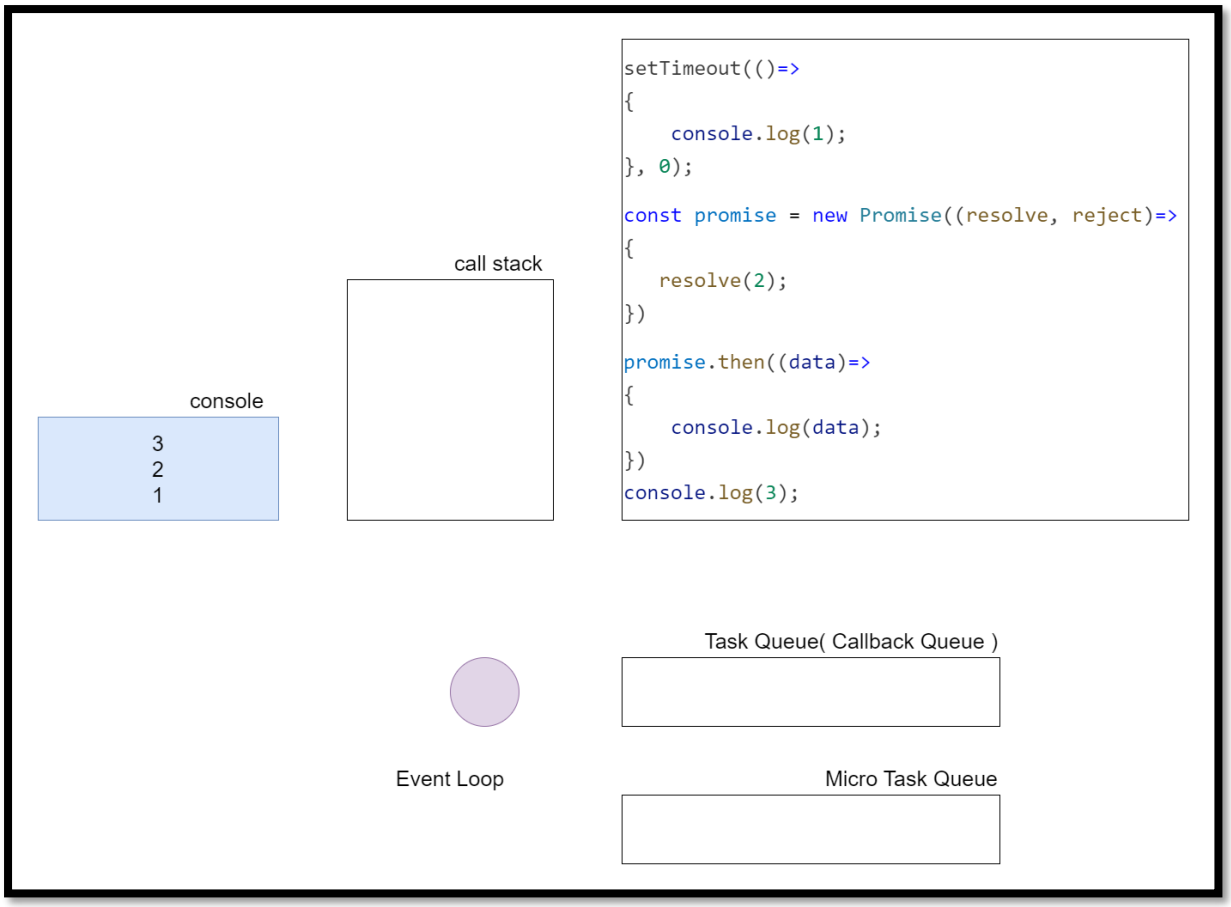

call stack
console
Timer
start
end
callback
console.log("start");
document.addEventListener('click', ()=>
{
console.log("callback");
})
console.log("start");
Task Queue(Callback Queue)
Event Loop

Promises and Event Loop




setTimeout(()=>
{
console.log(1);
}, 0);
const promise = new Promise((resolve, reject)=>
{
resolve(2);
})
promise.then((data)=>
{
console.log(data);
})
console.log(3);
call stack
cb2( )
console
3
Task Queue(Callback Queue)
cb1( )
Event Loop
Micro Task Queue


setTimeout(()=>
{
console.log(1);
}, 0);
const promise = new Promise((resolve, reject)=>
{
resolve(2);
})
promise.then((data)=>
{
console.log(data);
})
console.log(3);
call stack
cb1( )
console
3
2
Task Queue(Callback Queue)
Event Loop
Micro Task Queue



Event Loop

Event loop is the key to the asynchronous nature of javascript. The main task of the event loop is to take callbacks from the task queue and microtask queue and put them in the call stack. This operation is performed only when the call stack is empty. If there are tasks present in both the queues then it gives priority to the microtask queue. It pulls out callbacks from the microtask queue until it is empty and then starts with the task queue. This is the reason promises have a higher priority than other asynchronous methods.

Task Queue

All the callbacks from setTimeout() and setInterval() are pushed into this queue after the timer ends and then when the call stack is empty and microtask queue is cleared, the event loop will take the callbacks from here and push it to the call stack.

Microtask Queue

All the callbacks from the promises are pushed into this queue and then when the call stack is empty, the event loop will take the callbacks from here and push it to the call stack.

Async/Await Syntax Sugaring

An async function is a function declared with the async keyword, and the await keyword is permitted within them. The async and await keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

```
function getData()
{
    return Promise.resolve(1);
}

async function demo()
{
    const data = await getData();
    console.log(data);
}

demo();

console.log(2);
console.log(3);

//2
//3
//1
```

When the demo function is called, following things will happen.

1. Execution context will be created for demo.
2. getData function will be called which returns a promise.
3. When the JavaScript encounters the await keyword following things will happen.
   - 3.1. JavaScript engine suspends the demo function and the demo function will be thrown into the micro task queue.
   - 3.2. In the background, it will wait for the getData task to be completed.
   - 3.3. JavaScript will move on to the synchronous code i.e. console.log(1) and console.log(2).
   - 3.4. After the call stack becomes empty, the event loop transfers the demo function from micro task queue to call stack.
   - 3.5. In the demo function we are waiting for the getData task to completed. When it gets complete, then we can perform the particular action.