Javascript prioritizes the task and executes it through an endlessly running loop. This single-threaded loop is called an event loop. This loop waits for the task, Executes the task and then sleeps waiting for more tasks.
The execution of tasks mainly depends on the type of operation-Synchronous and Asynchronous. Javascript executes all the synchronous tasks at first and pushes the asynchronous task to the queue. When the engine's stack becomes empty, the event loop again pushes the operation from the queue to the stack.
It is clear that, event-loop selects the synchronous task at first, But when selecting the asynchronous task, It again prioritizes the task according to **Macro** and **Micro.**
Microtask are the async operation with the higher priority whereas macro are the async operation with lesser priority.

So, the event-loop works in the following way.

1) executes all the tasks from the standard stack which are synchronous ones. And if the stack is empty, then only moves to check the queue.
2) Executes all the operations from the microtask queue and if the microtask queue is empty, It moves to check the Macrotask queue.
3) Finally, If all the stacks are empty, It executes the tasks from the Macrotask queue.

Example:

```
 console.log(1);
//Macrotask
 const a = setTimeout(function () { console.log(2); }, 1000);
 const b = setTimeout(function () { console.log(3); }, 0);
 console.log(4);

//microtask
const promise = ()=>{
 return new Promise((resolve,reject)=>{
   resolve(console.log("resolved"))
 })
}

promise().then(()=>console.log("true"))

console.log("I am from top stack")
```

The output of the above code is

```
1
4
resolved
I am from top stack
true
3
```

So, From the output it is clear that Javascript adds the async operation in the queue and executes when the normal stack gets empty. Here, Promise gets resolved and then it consoles “I am from top stack” . It is because, whenever the normal stack gets empty it immediately executes the tasks from the microtask queue. But before executing .then, it needs to execute a task from the normal stack.

Question 2:

There are different ways of converting normal synchronous operation to asynchronous.

1) By wrapping the code inside the setTimeout or setInterval.
2) By wrapping the code inside the promise and using the callback to either resolve or reject. “resolve” and “reject” are the builtin async operations provided by js.
3) Declaring function as an async and awaiting the results.