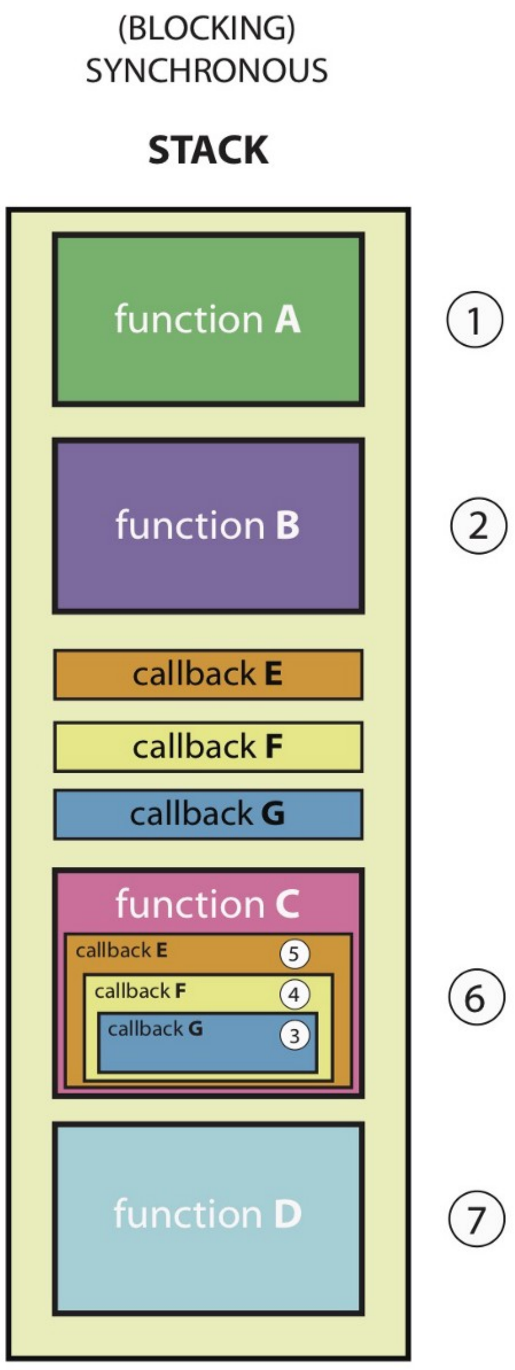
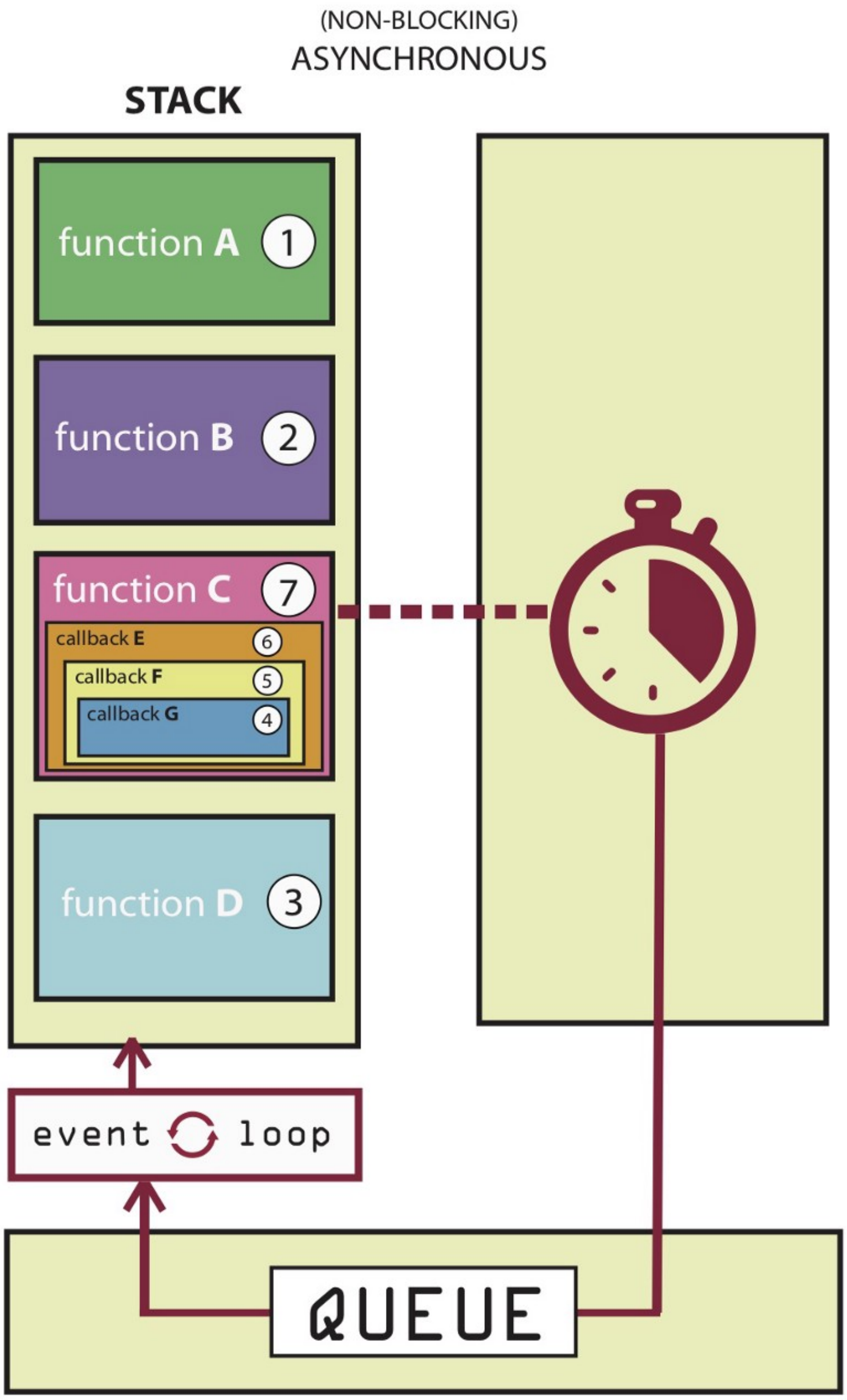
**Session-2**

1. Is JS asynchronous or synchronous language ?
   JavaScript is a **synchronous**, blocking, **single-threaded language**. That just means that only one operation can be in progress at a time. That's not the entire story, though!.



**Let's use a database request as an example**: asynchronous callbacks allow you to invoke a callback function which sends a database request (and any other nested callbacks) off to your app, where it waits for a response from the database, freeing up the rest of your code to continue running.



Once the database request completes, the results (and any other nested code) are sent to the queue and then processed through the event loop.

In the diagram here, you can see how this differs from the synchronous code. Function C, along with E, F and G are all sent off to the browser, queue and event loop.

So in easy words we can define it in 2 ways:
**Synchronous code:**
Synchronous code runs in sequence. This means that each operation must wait for the previous one to complete before executing.

```
console.log('One');
console.log('Two');
console.log('Three');
// LOGS: 'One', 'Two', 'Three'
```

**Asynchronous code:**
Asynchronous code runs in parallel. This means that an operation can occur while another one is still being processed.

```
console.log('One');
setTimeout(() => console.log('Two'), 100);
console.log('Three');
// LOGS: 'One', 'Three', 'Two'
```

Asynchronous code execution is often preferable in situations where execution can be blocked indefinitely. Some examples of this are **network requests, long-running calculations, file system operations etc**. Using asynchronous code in the browser ensures the page remains responsive and the user experience is mostly unaffected.

2. **Difference between var let and const:**
   In JavaScript, users can **declare a variable** using 3 keywords that are var, let, and const.
   So let's discuss each one:
   a. **var:** the var is the oldest keyword to declare a variable in JavaScript.
      **Scope**: Global scoped or function scoped. **The scope of the var keyword is the global or function scope**. It means variables defined outside the function can be accessed globally, and variables defined inside a particular function can be accessed within the function.
      **Example 1**: Variable 'x' is declared globally. So, the scope of the variable 'x' is global, and it can be accessible everywhere in the program. The output shown is in the console.

```
var x=10
function hello(){
   console.log(x);
}
hello();
console.log(x);
```

      **output:**

```
10
10
```

**Example 2:** The variable 'a' is declared inside the function. If the user tries to access it outside the function, it will display the error. Users can declare the 2 variables with the same name using the var keyword. Also, the user can reassign the value into the var variable. The output shown in the console.

```
function f() {

   // It can be accessible any
   // where within this function
   var a = 10;
   console.log(a)
}
f();
// A cannot be accessible
// outside of function
console.log(a);
output:
```
**10**
**ReferenceError: a is not defined**

**Example 3:** If users use the var variable before the declaration, it initializes with the undefined value. The output is shown in the console.

```
   console.log(a);
   var a = 10;
   Output:
```
**undefined**

b. **let:** The let keyword is an improved version of the var keyword.
**Scope: block scoped:** The scope of a let variable is only block scoped. It can't be accessible outside the particular block ({block}). Let's see the below example.
**Example 1:** The output is shown in the console.

```
   let a = 10;
   function f() {
      let b = 9
      console.log(b);
      console.log(a);
   }
   f();
```
**Output:**
**9**
**10**

**Example 2:** The code returns an error because we are accessing the let variable outside the function block. The output is shown in the console.

```
   let a = 10;
   function f() {
      if (true) {
            let b = 9

            // It prints 9
            console.log(b);
        }

        // It gives error as it
        // defined in if block
        console.log(b);
    }
    f()

    // It prints 10
    console.log(a)
```

Output:
9
ReferenceError: b is not defined

**Example 3**: **Users cannot re-declare the variable defined with the let keyword but can update it.**

```
    let a = 10

    // It is not allowed
    let a = 10

    // It is allowed
    a = 10
```

Output:
Uncaught SyntaxError: Identifier 'a' has already been declared

**Example 4:** Users can **declare the variable with the same name in different blocks** using the let keyword.

```
  let a = 10
  if (true) {
    let a=9
    console.log(a) // It prints 9
  }
  console.log(a) // It prints 10
```

Output:
9
10

Example 5: If users use the let variable before the declaration, **it does not initialize with undefined just like a var variable** and return an error.

```
console.log(a);
let a = 10;
```

Output:
Uncaught ReferenceError: Cannot access 'a' before initialization

c. **const:** The const keyword has all the properties that are the **same as the let keyword, except the user cannot update it**
**Scope: block scoped:** When users declare a const variable, they need to initialize it, otherwise, it returns an error. **The user cannot update the const variable once it is declared.**

**Example 1:** We are changing the value of the const variable so that it returns an error. The output is shown in the console.

```
const a = 10;
function f() {
    a = 9
    console.log(a)
}
f();
```

Output:
a=9
**TypeError:Assignment to constant variable.**

**Example 2:** Users **cannot change the properties of the const object, but they can change the value of properties** of the const object.

```
const a = {
    prop1: 10,
    prop2: 9
}

// It is allowed
a.prop1 = 3

// It is not allowed
a = {
    b: 10,
    prop2: 9
}
```

Output:
Uncaught SyntaxError:Unexpected identifier

| **var** | **let** | **const** |
|---|---|---|
| The scope of a var variable is functional scope. | The scope of a let variable is block scope. | The scope of a const variable is block scope. |
| It can be updated and re-declared into the scope. | It can be updated but cannot be re-declared into the scope. | It cannot be updated or re-declared into the scope. |
| It can be declared without initialization. | It can be declared without initialization. | It cannot be declared without initialization. |
| It can be accessed without initialization as its default value is “undefined”. | It cannot be accessed without initialization, as it returns an error. | It cannot be accessed without initialization, as it cannot be declared without initialization. |

**3. What is hoisting in javascript:**
Hoisting is JavaScript's default behavior of **moving all declarations to the top** of the current scope (to the top of the current script or the current function).
Hoisting means that you can **define a variable before its declaration.**
**1. Variable hoisting**
Variable hoisting means the JavaScript engine moves the variable declarations to the top of the script. For example, the following example declares the counter variable and initialize its value to 1:

```
console.log(counter); // undefined
var counter = 1;
```

In this example, we reference the counter variable before the declaration.

However, the first line of code doesn’t cause an error. The reason is that the JavaScript engine moves the variable declaration to the top of the script.

**Technically, the code looks like the following in the execution phase:**

```
var counter;
console.log(counter); // undefined
counter = 1;
```

the following declares the variable counter with the let keyword:

```
console.log(counter);
let counter = 1;
```

The JavaScript issues the following error:
"ReferenceError: Cannot access 'counter' before initialization
The error message explains that the counter variable is already in the heap memory. However, it hasn't been initialized.

**Behind the scenes, the JavaScript engine hoists the variable declarations that use the let keyword. However, it doesn't initialize the let variables.**

**2.Function hoisting**
Like variables, t**he JavaScript engine also hoists the function declarations**. This means that the JavaScript engine also moves the function declarations to the top of the script.

For example:

```
let x = 20,
  y = 10;
let result = add(x, y);
console.log(result);
function add(a, b) {
  return a + b;
}
```

In this example, we called the add() function before defining it. The above code is equivalent to the following:

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

let x = 20,
    y = 10;

let result = add(x,y);
console.log(result);
```

**4. what is a promise in javascript?**
A promise is an object which can be returned synchronously from an asynchronous function. It will be in one of 3 possible states:
**Fulfilled:** onFulfilled() will be called (e.g., resolve() was called)
**Rejected:** onRejected() will be called (e.g., reject() was called)
**Pending:** not yet fulfilled or rejected
Here is a function that returns a promise which will resolve after a specified time delay:

```
const wait = time => new Promise((resolve) => setTimeout(resolve, time));
wait(3000).then(() => console.log('Hello!')); // 'Hello!'
```

Our wait(3000) call will wait 3000ms (3 seconds), and then log 'Hello!'. All spec-compatible promises define a .then() method which you use to pass handlers which can take the resolved or rejected value.

example:

```
let promise = new Promise(function(resolve, reject) {
  // the function is executed automatically when the promise is constructed

  // after 1 second signal that the job is done with the result "done"
  setTimeout(() => resolve("done"), 1000);
});
```

Promises are used to **handle asynchronous operations in JavaScript.** They are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code.

Multiple **callback functions would create callback hell that leads** to unmanageable code. Also it is not easy for any user to handle multiple callbacks at the same time.

Promises are the ideal choice for **handling asynchronous operations in the simplest manner. They can handle multiple asynchronous operations easily** and provide better error handling than callbacks and events.

**Benefits of Promises**

- Improves Code Readability
- Better handling of asynchronous operations
- Better flow of control definition in asynchronous logic
- Better Error Handling

what is promisify in javascript:
Promisify is used when you want to **convert a callback function into a promise based function**. Nowadays, is used promises because let the developers to write more structured code. With promises you can use then to call another function.
Example:
Callback Example

```
 a (function (data1) {
  b (function (data2) {
    c (function (data3) {
      d (function (data4) {
        e (function (data5) {
          f (function (data6) {
            // some code
          })
        }
      })
    })
  })
})
```

Its called as callback hell we can solve it using promise
Example of promise to solve it:

```
a(data1)
.then(return b(data2))
.then(return c(data3))
.then(return d(data4))
.then(return e(data5))
```