1. Synchronous:

Synchronous communication or execution refers to a real-time interaction where the sender and receiver are actively engaged at the same time.
In this mode, there is a direct and immediate exchange of information or execution of tasks.
It follows a "one at a time" approach, meaning that one action or request is completed before moving on to the next.

- Requires immediate participation from all involved parties.
- Offers instant feedback and response.
- Requires coordination and synchronization.
- May require all participants to be available at the same time.

2. Asynchronous:

Asynchronous communication or execution refers to a delayed or non-real-time interaction where the sender and receiver do not need to be engaged simultaneously. In this mode, actions or requests can occur independently and continue regardless of the availability or presence of the other party.

- Does not require immediate participation from all parties.
- Allows for flexible timing and individual pace.
- Responses or actions can be delayed or happen independently.
- Reduces the need for real-time coordination.

Q.2 What are Web Apis ?

Web APIs (Application Programming Interfaces) are sets of rules and protocols that allow different software applications to communicate and interact with each other over the internet. Web APIs enable applications to access and utilize the functionality, data, or services provided by another application, system, or platform.

1. Communication: Web APIs act as intermediaries that facilitate communication between different software systems, allowing them to exchange information and perform specific tasks.
2. Standardization: Web APIs follow standardized protocols and data formats to ensure compatibility and interoperability between applications. The most common protocols used for web APIs are HTTP (Hypertext Transfer Protocol) and HTTPS (HTTP Secure).
3. Resources: Web APIs expose specific resources or functionalities of an application or service, such as retrieving data, performing actions, or accessing specific features.
4. Request-Response Model: Web APIs typically operate on a request-response model, where a client application sends a request to the API, and the API responds with the requested data or performs the requested action.

Q.3 Explain SetTimeOut and setInterval ?

1. setTimeout:

The setTimeout function is used to execute a specified function or a code snippet once after a specified delay. It takes two arguments: a function or code snippet to execute, and the delay (in milliseconds) after which the execution should occur.

Eg. setTimeout(function, delay);

```
setTimeout(() => {
  console.log("Delayed execution after 2000 milliseconds");
}, 2000);
```

setInterval:

The setInterval function is used to repeatedly execute a specified function or a code snippet at a fixed interval. It also takes two arguments: a function or code snippet to execute, and the interval (in milliseconds) at which the execution should repeat.

setInterval(function, interval);
Example:

```
let counter = 0;
let intervalId = setInterval(() => {
  console.log("Interval execution every 1000 milliseconds");
  counter++;
  if (counter === 5) {
    clearInterval(intervalId); // Stop execution after 5 iterations
  }
}, 1000);
```

Q.4 how can you handle Async code in JavaScript ?

Async/Await:
Introduced in ES2017, async/await provides a more synchronous-looking syntax for handling asynchronous code. It is built on top of promises and simplifies the handling of async operations, making the code more readable and easier to write.

```
function delay(a) {
    return new Promise(resolve => setTimeout(resolve, a));
  }
  async function asyncOperation() {
    await delay(2000);
    const result = 'Async operation complete';
    return result;
  }
  async function executeAsync() {
    try {
      const result = await asyncOperation();
      console.log(result);
    } catch (error) {
      console.error(error);
    }
  }
```

```
executeAsync();
```

Q.5 What are Callbacks & Callback Hell ?

Callbacks:
Callbacks are functions that are passed as arguments to other functions and are invoked at a later point in time, often when an asynchronous operation is completed. Callbacks allow you to define what should happen after an asynchronous operation finishes executing.
In JavaScript, callbacks are commonly used to handle asynchronous operations such as fetching data from a server, reading files, or making API calls. By passing a callback function, you can define the behavior that should occur once the async operation completes or encounters an error.

```
function fetchData(callback) {
    // Simulating an asynchronous operation
    setTimeout(() => {
      const data = { id: 1, name: 'John Doe' };
      callback(data);
    }, 2000);
  }

  function processData(data) {
    console.log('Processed data:', data);
  }

  fetchData(processData);
```

Callback Hell:
Callback Hell refers to a situation where multiple asynchronous operations are nested inside each other, resulting in deeply nested callbacks. This can make the code difficult to read, maintain, and reason about, leading to what is often called "callback spaghetti" or "pyramid of doom."

```
asyncOperation1(function(error, result1) {
    if (error) {
      console.error(error);
    } else {
      asyncOperation2(result1, function(error, result2) {
        if (error) {
          console.error(error);
        } else {
          asyncOperation3(result2, function(error, result3) {
            if (error) {
              console.error(error);
            } else {
              // More nested callbacks...
            }
          });
        }
```

```
    });
  }
});
```

```
asyncOperation1()
  .then(result1 => asyncOperation2(result1))
  .then(result2 => asyncOperation3(result2))
  .then(result3 => {
    // Handle final result
  })
  .catch(error => {
    console.error(error);
  });
```

Promises:

Promises are a built-in JavaScript feature introduced in ES6 (ECMAScript 2015) to handle asynchronous operations in a more structured and readable way. Promises represent the eventual completion or failure of an asynchronous operation
and allow you to chain multiple asynchronous operations together.

Promises have three states:

1. Pending: The initial state when a promise is created, and the asynchronous operation is still ongoing.
2. Fulfilled: The state when the asynchronous operation is successfully completed, and the promise is resolved with a value.
3. Rejected: The state when an error occurs during the asynchronous operation, and the promise is rejected with a reason or error message.

1. then:

The then method is used to handle the fulfillment of a promise. It takes two optional callback functions as arguments:
onFulfilled and onRejected. These functions are executed when the promise is successfully fulfilled or rejected, respectively.
The then method returns a new promise, allowing chaining of multiple async operations.

Example:

```
asyncOperation()
  .then(result => {
    console.log('Async operation fulfilled with result:', result);
  })
  .catch(error => {
    console.error('Async operation rejected with error:', error);
  });
```

2. catch:

The catch method is used to handle the rejection of a promise. It is similar to the onRejected callback in the then method, but specifically handles errors or rejections. If any error occurs in the promise chain, the catch method allows you to catch and handle that error.

```
asyncOperation()
  .then(result => {
    console.log('Async operation fulfilled with result:', result);
  })
  .catch(error => {
    console.error('Async operation rejected with error:', error);
  });
```

3. finally:

The finally method is used to specify a callback that should be executed regardless of whether the promise is fulfilled or rejected. It is commonly used to perform cleanup tasks or execute code that should always run, regardless of the outcome of the promise.

```
asyncOperation()
  .then(result => {
    console.log('Async operation fulfilled with result:', result);
  })
  .catch(error => {
    console.error('Async operation rejected with error:', error);
  })
  .finally(() => {
    console.log('Cleanup or final actions');
  });
```

Q.7 What's async & await Keyword in JavaScript

1. async Keyword:

The async keyword is used to define an asynchronous function. An asynchronous function returns a promise implicitly, allowing you to use await within the function to pause execution until a promise is fulfilled or rejected.

2. await Keyword:

The await keyword is used inside an async function to pause the execution until a promise is resolved.
It can only be used within an async function and allows you to write asynchronous code in a more sequential manner, as if it were synchronous.

Q.8 Explain Purpose of Try and Catch Block & Why do we need it?

Try/Catch:

No, it is not a replacement for an if, then block, it serves an entirely different purpose.
The objective of a try, catch block is to try and do something which could fail and raise an exception (e.g., read a file from disk, but the file might not be there, etc.). After catching an exception, you can handle it.

```
try {
    // Code that may throw an error or raise an exception
  } catch (error) {
    // Error handling logic
  }
```

Q.9 Explain fetch

The fetch() method in JavaScript is used to request data from a server. The request can be of any type of API that returns the data in JSON or XML. The fetch() method requires one parameter, the URL to request, and returns a promise.

```
const url = 'https://api.ex.com/data';


fetch(url)
  .then(response => {
    if (!response.ok) {
      throw new Error('Request failed with status ' + response.status);
    }
    return response.json();
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error(error);
  });
const postData = { name: 'John Doe', age: 25 };
fetch(url, {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(postData)
})
  .then(response => {
    if (!response.ok) {
      throw new Error('Request failed with status ' + response.status);
    }
    return response.json();
```

```
  })
  .then(data => {

    console.log(data);
  })
  .catch(error => {

    console.error(error);
  });
```

Q.10 How do you define an asynchronous function in JavaScript using async/await?

The word "**async**" before a function means one simple thing: a function always returns a promise.
Other values are wrapped in a resolved promise automatically.

```
async function f() {
    return 1;
  }

  f().then(alert); // 1
```

The keyword "**await"** makes JavaScript wait until that promise settles and returns its result.

```
async function f() {

    let promise = new Promise((resolve, reject) => {
      setTimeout(() => resolve("done!"), 1000)
    });

    let result = await promise; // wait until the promise resolves (*)

    alert(result); // "done!"
  }

  f();
```