Q1.What’s difference between Synchronous and Asynchronous?
Synchronous code is executed sequentially, meaning that each line of code is executed one after the other, and the program waits for each line to complete before moving on to the next line. In other words, synchronous operations block the execution of code until they are finished.

Asynchronous code, on the other hand, allows multiple operations to be executed concurrently. It doesn't block the execution of code, and instead, it uses callbacks, promises, or async/await to handle the results of the asynchronous operations once they are completed. Asynchronous operations are typically used for tasks that may take some time to complete, such as network requests or file I/O operations, to avoid blocking the main execution thread.

Q.2 What are Web Apis ?
Web APIs (Application Programming Interfaces) are interfaces provided by web browsers that allow developers to interact with various web features and services. These APIs provide a set of rules and protocols for building web applications and accessing functionalities such as manipulating the DOM (Document Object Model), making HTTP requests, handling storage, and much more.

Some commonly used Web APIs include:

- DOM API: Allows manipulation and interaction with the HTML structure and content of a web page.
- XMLHttpRequest: Enables making HTTP requests from JavaScript.
- Fetch API: A newer and more powerful alternative to XMLHttpRequest for making HTTP requests.
- Geolocation API: Provides access to the user's geographic location.
- Canvas API: Allows dynamic rendering and manipulation of graphics and images.
- Web Storage API: Provides client-side storage options such as localStorage and sessionStorage.
- Web Audio API: Enables audio processing and synthesis within web applications.
- Web Speech API: Provides speech recognition and synthesis capabilities.

These APIs allow developers to extend the capabilities of web applications and create interactive and dynamic experiences for users.

Q.3 Explain SetTimeOut and setInterval ?
`setTimeout` and `setInterval` are functions in JavaScript used for scheduling the execution of code after a certain delay or at regular intervals.

`setTimeout` is used to execute a function or a piece of code once after a specified delay. It takes two arguments: the function or code to be executed and the delay in milliseconds. Here's an example:

```javascript
```

```
setTimeout(() => {
  console.log('This code will run after 2000 milliseconds (2 seconds).');
}, 2000);
```

`setInterval` is used to repeatedly execute a function or code at a specified interval. It also takes two arguments: the function or code to be executed and the interval in milliseconds. Here's an example:

```javascript
setInterval(() => {
  console.log('This code will run every 1000 milliseconds (1 second).');
}, 1000);
```

Both `setTimeout` and `setInterval` return a unique identifier that can be used to cancel the scheduled execution using the `clearTimeout` or `clearInterval` functions, respectively.

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

1. Callbacks: Callbacks are functions passed as arguments to asynchronous functions. They are called once the asynchronous operation is completed. Callbacks can be error-first callbacks (where the first argument is reserved for an error object) or success callbacks. However, using callbacks can lead to callback hell, which makes code difficult to read and maintain.

2. Promises: Promises provide an improved way of dealing with asynchronous operations. They represent a value that may be available now, in the future, or never. Promises have built-in methods like `then()` and `catch()` to handle success and error cases. Promises can be chained together using these methods to form a sequence of asynchronous operations.

3. Async/await: Async/await is a modern approach introduced in ES2017 (ES8) for handling asynchronous code. It allows you to write asynchronous code that looks and behaves like synchronous code, making it more readable and easier to understand. The `async` keyword is used to define an asynchronous function, and the `await` keyword is used to pause the execution of the function until a promise is fulfilled. This way, you can write asynchronous code that appears to be executed in a synchronous manner.

Q.5 What are Callbacks & Callback Hell ?
Callbacks are functions passed as arguments to other functions and are executed once an asynchronous operation is completed. They allow you to specify code that should be executed after a certain task has finished. However, when callbacks are nested within multiple asynchronous operations, it can lead to callback hell or the pyramid of doom. This occurs when code becomes deeply nested and hard to read due to excessive indentation.

Here's an example of callback hell:

```javascript
asyncFunc1(function (error, result1) {
  if (error) {
    console.error(error);
  } else {
    asyncFunc2(result1, function (error, result2) {
      if (error) {
        console.error(error);
      } else {
        asyncFunc3(result2, function (error, result3) {
          if (error) {
            console.error(error);
          } else {
            // More nested callbacks...
          }
        });
      }
    });
  }
});
```

This code becomes harder to understand and maintain as more callbacks are added. To mitigate callback hell, alternative approaches like Promises or async/await can be used.

Q.6 What are Promises & Explain Some Three Methods of Promise

Promises are a feature in JavaScript that provide a better way to deal with asynchronous code compared to callbacks. A promise represents the eventual completion (or failure) of an asynchronous operation and can be in one of three states: pending, fulfilled, or rejected.

Here's an example of creating and using a promise:

```javascript
const fetchData = new Promise((resolve, reject) => {
  // Simulating an asynchronous operation
  setTimeout(() => {
    const data = 'Some fetched data';
    if (data) {
      resolve(data); // Operation succeeded
    } else {
      reject('Data not found'); // Operation failed
    }
```

```
  }, 2000);
});

fetchData
  .then((data) => {
    console.log(data); // Use the fetched data
  })
  .catch((error) => {
    console.error(error); // Handle the error
  });
```

In the example above, the promise `fetchData` represents the asynchronous operation of fetching some data. When the operation completes successfully, the `resolve` function is called with the data as an argument. If an error occurs, the `reject` function is called with an error message.

Promises provide three essential methods:

1. `then()`: Attaches a callback function to be executed when the promise is fulfilled. It takes two optional arguments: a success callback and a failure callback.

2. `catch()`: Attaches a callback function to be executed when the promise is rejected. It is used for error handling.

3. `finally()`: Attaches a callback function to be executed when the promise is settled, whether it's fulfilled or rejected. It is useful for cleanup operations.

Q.7 What’s async & await Keyword in JavaScript
The `async` and `await` keywords are part of the ES2017 (ES8) update to JavaScript and provide a more readable and synchronous-like way of working with promises.

The `async` keyword is used to define an asynchronous function. It allows the function to use the `await` keyword inside it, indicating that the function should pause its execution until the awaited promise is fulfilled.

Here's an example:

```javascript
async function fetchData() {


  const response = await fetch('https://api.example.com/data');
  const data = await response.json();
```

```
  return data;
}

fetchData()
  .then((data) => {
    console.log(data); // Process the fetched data
  })
  .catch((error) => {
    console.error(error); // Handle any errors
  });
```

In the example above, the `fetchData` function is defined with the `async` keyword. It uses the `await` keyword to pause the execution of the function until the `fetch` request is completed and the response is obtained. The `await` keyword ensures that the value of `response` is assigned only after the promise is fulfilled.

By using `async/await`, you can write asynchronous code that looks and behaves like synchronous code, making it easier to understand and maintain.

Q.8 Explain Purpose of Try and Catch Block & Why do we need it?
The purpose of the `try` and `catch` blocks in JavaScript is to handle errors and provide a mechanism for graceful error handling in your code.

The `try` block is used to enclose the code that might throw an exception. If an exception occurs within the `try` block, the execution is immediately transferred to the `catch` block.

The `catch` block is used to catch and handle the exception thrown in the `try` block. It takes an error parameter that represents the caught exception. You can use this parameter to handle the error, log it, or perform any necessary error recovery operations.

Here's an example:

```javascript
try {
  // Code that might throw an exception
  const result = someFunction();
  console.log(result);
} catch (error) {
  // Handle the error
  console.error('An error occurred:', error);
}
```

In the example above, if an exception is thrown within the `try` block, the code execution jumps directly to the `catch` block, allowing you to handle the error gracefully. Without the `try/catch` block, an unhandled exception would cause the program to terminate and potentially disrupt the user experience.

By using `try/catch` blocks, you can capture and handle errors, preventing them from crashing your application and enabling you to provide fallback strategies or alternative actions.

Q.9 Explain fetch
`fetch` is a modern API in JavaScript that provides an easy way to make asynchronous HTTP requests to fetch resources (like JSON data) from a network. It is built into most modern web browsers and can be used both in the browser and in Node.js with additional libraries or polyfills.

Here's an example of using `fetch` to make a GET request:

```javascript
fetch('https://api.example.com/data')
  .then((response) => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then((data) => {
    console.log(data); // Process the fetched data
  })
  .catch((error) => {
    console.error(error); // Handle any errors
  });
```

In the example above, `fetch` is called with the URL of the resource to fetch. It returns a promise that resolves to the `Response` object representing the response to the request. The response can be checked for errors using the `ok` property. If the response is not successful (e.g., for a 404 status), an error is thrown.

The response can then be parsed using the `json()` method, which returns a promise that resolves to the JSON data. The fetched data can be further processed in the subsequent `then()` block.

The use of promises and chaining allows for more concise and readable code when dealing with asynchronous requests.

Q.10 How do you define an asynchronous function in JavaScript using async/await?
To define an asynchronous function in JavaScript using `async/await`, you need to use the `async` keyword before the function declaration. This tells JavaScript that the function will contain asynchronous code, and it will return a promise.

Here's an example:

```javascript
async function asyncFunction() {
  // Asynchronous code
  const result = await someAsyncOperation();
  return result;
}
```

In the example above, the `asyncFunction` is defined with the `async` keyword. Inside the function, you can use the `await` keyword to pause the execution until an awaited promise is fulfilled. The `await` expression will pause the function and wait for the promise to resolve, and then it will resume the execution and assign the resolved value to the `result` variable.

The function can be called like any other function, and it will return a promise. You can use the `then()` method to handle the result or the `catch()` method to handle any errors.

```javascript
asyncFunction()
  .then((result) => {
    console.log(result);
  })
  .catch((error) => {
    console.error(error);
  });
```

Using `async/await` allows you to write asynchronous code that appears more synchronous and is easier to read and understand.