Q.1 What’s the difference between Synchronous and Asynchronous?

Ans:
Synchronous and asynchronous are terms used to describe different modes of communication or interaction in various contexts, including computer programming, telecommunications, and general communication systems. Here's the difference between synchronous and asynchronous:

1. Synchronous:

- Synchronous communication implies that there is a strict timing relationship between the sender and the receiver.

- In synchronous communication, the sender and the receiver must be available and actively engaged at the same time.

- It involves a request-response mechanism, where the sender waits for a response from the receiver before proceeding.

- Examples of synchronous communication include real-time conversations, phone calls, and in-person meetings.

- Synchronous operations are blocking, meaning that the execution of the program or process waits until the operation is completed.

2. Asynchronous:

- Asynchronous communication does not require the sender and the receiver to be active simultaneously.

- In asynchronous communication, the sender initiates a request and does not wait for an immediate response.

- The sender continues with other tasks, and the response can be processed later when it becomes available.

- Examples of asynchronous communication include email, asynchronous messaging systems, and asynchronous programming.

- Asynchronous operations are non-blocking, meaning that the execution of the program or process can continue while waiting for the operation to complete.

Q.2 What are Web Apis?

Ans:

Web APIs, also known as Browser APIs or JavaScript APIs, are sets of functionalities provided by web browsers that allow developers to interact with web features and perform various tasks using JavaScript.

These APIs provide a way to access and control different aspects of web browsers, such as manipulating the DOM (Document Object Model), making HTTP requests, handling user events, and more. Here are some commonly used Web APIs in JavaScript:

1. DOM API (Document Object Model):

- The DOM API provides methods and properties to interact with HTML and XML documents.

- It allows developers to access, modify, and manipulate the structure, content, and style of web pages dynamically.

2. XMLHttpRequest API:

- The XMLHttpRequest API enables making asynchronous HTTP requests to retrieve data from a server without reloading the entire page.

- It is commonly used for AJAX (Asynchronous JavaScript and XML) interactions, allowing for dynamic content updates.

3. Fetch API:

- The Fetch API provides a more modern and flexible way to make HTTP requests, including handling different types of data and managing the request and response objects.

- It simplifies the process of making network requests and handling responses using promises.

4. Geolocation API:

- The Geolocation API allows obtaining the user's current geographical location.

- It provides methods to request and handle location data, enabling developers to build location-based applications.

5. Web Storage API:

- The Web Storage API provides a way to store key-value pairs locally on a user's browser.

- It includes two mechanisms: sessionStorage (stores data for a session) and localStorage (stores data persistently).

6. Canvas API:

- The Canvas API provides a drawing and graphics environment for creating dynamic visual content on web pages.

- It allows drawing shapes, images, and text, applying transformations, and handling user input.

7. Web Audio API:

- The Web Audio API enables audio manipulation and processing, allowing developers to create and control audio content on the web.

Q.3 Explain SetTimeOut and setInterval.

Ans:
Both `setTimeout` and `setInterval` are functions in JavaScript that allow you to schedule the execution of code at specific intervals. Here's an explanation of each:

1. setTimeout:

- The `setTimeout` function is used to execute a piece of code or a function after a specified delay (in milliseconds).

- It takes two parameters: a function or code to be executed and a delay value.

- The delay value determines the time in milliseconds to wait before executing the code.

- After the specified delay, the code is executed once.

- Example usage:

```
setTimeout(function() {
  console.log("This code will execute after a 1-second delay.");
}, 1000);
```

2. setInterval:

- The `setInterval` function is used to repeatedly execute a piece of code or a function at a specified interval.

- It also takes two parameters: a function or code to be executed and an interval value.

- The interval value determines the time in milliseconds between each execution of the code.

- The code is executed repeatedly, with each execution occurring after the specified interval.

- Example usage:

```
var counter = 0;
var intervalId = setInterval(function() {
  console.log("Counter: " + counter);
  counter++;
}, 1000);
```

This code will log the value of the counter variable every second.

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

Ans:

In JavaScript, asynchronous code can be handled using various techniques and patterns.

1. Callbacks: Callbacks are a traditional way to handle asynchronous code in JavaScript. You can pass a function (callback) as an argument to an asynchronous function, and that function will be invoked once the asynchronous operation is complete. However, nesting callbacks can lead to callback hell and make the code difficult to read and maintain.

Example:

```
function fetchData(callback) {
  // Simulating asynchronous operation
  setTimeout(function() {
    const data = "Some data";
    callback(data);
  }, 1000);
}

fetchData(function(result) {
  console.log(result); // Handle the result
});
```

2. Promises: Promises were introduced to solve the callback hell problem and provide a more structured way of handling asynchronous code. A Promise represents the eventual completion (or failure) of an asynchronous operation and allows you to chain operations using `then()` and `catch()`.

Example:

```
function fetchData() {
  return new Promise(function(resolve, reject) {
    // Simulating asynchronous operation
    setTimeout(function() {
      const data = "Some data";
      resolve(data); // Success
```

```
    // reject(new Error("Failed")); // Error
  }, 1000);
 });
}

fetchData()
 .then(function(result) {
  console.log(result); // Handle the result
 })
 .catch(function(error) {
  console.error(error); // Handle errors
 });
```

3. Async/await: Async/await is a syntax built on top of Promises, which provides a more synchronous style of writing asynchronous code. The `async` keyword is used to declare an asynchronous function, and the `await` keyword is used to pause the execution until a Promise is resolved or rejected.

Example:

```
function fetchData() {
 return new Promise(function(resolve, reject) {
  // Simulating asynchronous operation
  setTimeout(function() {
   const data = "Some data";
   resolve(data); // Success
   // reject(new Error("Failed")); // Error
  }, 1000);
 });
}

async function handleData() {
 try {
  const result = await fetchData();
  console.log(result); // Handle the result
 } catch (error) {
  console.error(error); // Handle errors
 }
}

handleData();
```

Q.5 What are Callbacks & Callback Hell?

Ans:

Callbacks:

- Callbacks are functions passed as arguments to another function.

- They are commonly used to handle asynchronous operations or events.

- Callback functions are invoked at a later time or when a specific condition is met.

- They allow you to execute code after an asynchronous operation completes and pass the result to the callback function.

Callback Hell:

- Callback Hell refers to the situation when dealing with multiple nested callbacks in asynchronous code.

- It occurs when there are many dependent asynchronous operations, leading to deeply nested and hard-to-read code.

- Callback Hell makes the code difficult to understand, maintain, and debug.

- The indentation and nesting of callbacks can become increasingly complex as the number of dependent operations grows.

Example of Callback Hell:

```
asyncOperation1(function(result1) {
  asyncOperation2(result1, function(result2) {
    asyncOperation3(result2, function(result3) {
      // ...more nested callbacks
    });
```

```
  });

});
```

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

Ans:

Promises are a built-in JavaScript feature introduced to handle asynchronous operations in a more structured and manageable way.
A Promise represents the eventual completion (or failure) of an asynchronous operation and allows you to chain operations together. Promises have three states: pending, fulfilled, or rejected.

Here are three important methods associated with Promises:

1. `then()`:

The `then()` method is used to handle the successful completion of a Promise. It takes two optional arguments: `onFulfilled` and `onRejected`. The `onFulfilled` function is called when the Promise is fulfilled (resolved successfully), and it receives the resolved value as an argument. The `onRejected` function is called when the Promise is rejected, and it receives the rejection reason as an argument.

Example:

```
fetchData()
  .then(function(result) {
    console.log(result); // Handle the result
  })
  .catch(function(error) {
    console.error(error); // Handle errors
  });
```

2. `catch()`:

The `catch()` method is used to handle the rejection of a Promise. It takes a single argument, which is the `onRejected` function. This function is called when the Promise is rejected, and it receives the rejection reason as an argument. It is used for error handling in Promises.

Example:

```
fetchData()
  .then(function(result) {
    console.log(result); // Handle the result
  })
  .catch(function(error) {
    console.error(error); // Handle errors
  });
```

3. `finally()`:

The `finally()` method is used to specify a callback function that will be executed regardless of whether the Promise is fulfilled or rejected. It is often used to perform cleanup tasks or actions that should be done in both success and error cases.

Example:

```
fetchData()
  .then(function(result) {
    console.log(result); // Handle the result
  })
  .catch(function(error) {
    console.error(error); // Handle errors
  })
  .finally(function() {
    console.log("Cleanup"); // Execute regardless of fulfillment or rejection
  });
```

Q.7 What’s the async & await Keyword in JavaScrip

Ans:

`async` and `await`:

- `async` is a keyword used to declare an asynchronous function, indicating that the function will perform asynchronous operations.

- An `async` function automatically returns a Promise, allowing the use of `await` inside the function.

- `await` is a keyword used to pause the execution of an `async` function until a Promise is resolved or rejected.

- `await` can only be used inside an `async` function.

- When encountering an `await` expression, the function execution pauses until the Promise settles.

- The result of the awaited Promise is returned, allowing synchronous-style coding with asynchronous behavior.

Example:

```
async function fetchData() {

  const result = await fetch(url);

  // Code execution waits until the Promise from `fetch` resolves

  // `result` contains the resolved value of the Promise

  return result;

}
```

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

Ans:

The purpose of the `try` and `catch` blocks in JavaScript is to handle exceptions (or errors) that may occur during the execution of a block of code.

They provide a mechanism for capturing and responding to errors in a controlled manner. Here's an explanation of their purpose and why we need them:

1. Error Handling:

The primary purpose of the `try` and `catch` blocks is to handle errors that can occur during the execution of code. When an error occurs within the `try` block, it gets caught by the corresponding `catch` block, allowing you to handle the error gracefully.

2. Graceful Degradation:

By using `try` and `catch`, you can prevent your program from crashing or abruptly terminating when an error occurs. Instead, you can gracefully degrade the application's behavior and provide fallback logic or error messages to the user.

3. Debugging and Logging:

Using `try` and `catch` enables you to capture and log detailed information about the error, including the error message, stack trace, and other relevant data. This information can be invaluable for debugging purposes, as it helps identify the source and cause of the error.

4. Error Recovery:

In some cases, you might want to recover from specific types of errors and continue the execution of your code. By catching the error with `catch`, you have the opportunity to perform error recovery actions, such as retrying the operation, providing default values, or attempting an alternative approach.

Example:

```
try {

  // Code that may throw an error
```

```
} catch (error) {

  // Error handling logic

}
```

Q.9 Explain fetch

Ans:

1. Web API for HTTP Requests:

   - `fetch` is a built-in web API in JavaScript for making HTTP requests, typically used to retrieve data from a server.

   - It provides a modern alternative to the older XMLHttpRequest (XHR) object.

2. Promise-Based:

   - The `fetch` function returns a Promise that resolves to the response from the server.

   - This allows you to handle the asynchronous nature of the HTTP request using `then()` and `catch()` or `async/await`.

3. Syntax:

   - The `fetch` function is invoked by passing a URL as the first parameter.

   - Additional optional parameters can be passed to configure the request, such as headers, request method, body, etc.

4. Response Handling:

   - The Promise returned by `fetch` resolves to a `Response` object representing the response from the server.

   - You can use methods like `json()`, `text()`, `blob()`, `arrayBuffer()`, etc., on the `Response` object to extract the response data.

5. Error Handling:

- The Promise rejects if the network request fails, such as due to network connectivity issues or server errors.

- You can handle such errors using the `catch()` method or by using `try/catch` when using `async/await`.

6. Cross-Origin Requests:

- The `fetch` function follows the same-origin policy, which means it is subject to cross-origin restrictions.

- Cross-origin requests may require server-side CORS (Cross-Origin Resource Sharing) configuration to allow access from different domains.

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

Ans:
To define an asynchronous function in JavaScript using `async/await`, We need to follow these steps:

1. Declare the function using the `async` keyword:

   - Prepend the `async` keyword before the function declaration.

   - This indicates that the function will contain asynchronous operations and will return a Promise.

2. Use the `await` keyword to pause the execution:

   - Inside the `async` function, use the `await` keyword followed by a Promise.

   - This pauses the execution of the function until the Promise is resolved or rejected.

   - The `await` expression returns the resolved value of the Promise.

3. Handle errors using `try/catch`:

   - Wrap the code inside the `async` function with a `try` block to catch any potential errors.

   - Use the `catch` block to handle and manage the caught errors.

Here's an example that demonstrates the definition of an asynchronous function using `async/await`:

```
async function fetchData() {
  try {
    // Asynchronous operations
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
```

```
    console.log(data);
  } catch (error) {
    console.error('Error:', error);
  }
}
```