Time Intervals:

In JavaScript, time intervals refer to the ability to execute code or functions repeatedly at specified intervals of time. This functionality is commonly used for tasks like animations, periodic updates, or timed actions in web applications.

1. **setTimeout():**

- Executes a function or evaluates an expression once after a specified delay.
- Syntax: **setTimeout(function, delay, arg1, arg2, ...)**

```
let batch  = (a,b)=>{
   console.log(`${a} ${b}`)
}
setTimeout(()=>{
   console.log("This is Two")

},4000,"Hema","Coding")
```

1. **setInterval():**

- Executes a function or evaluates an expression repeatedly at specified intervals.
- Syntax: **setInterval(function, interval, arg1, arg2, ...)**

```
let stopInterval = setInterval(()=>{
   console.log("This is set Interval")

},1000,"Hema","Coding")
```

1. **clearInterval():**

- Stops the execution of code specified by **setInterval()**.
- Syntax: **clearInterval(intervalId)**

```
setTimeout(()=>{
   console.log("stopped Interval")

   clearInterval(stopInterval)
},3000)
```

Callback Hell

Definition: Callback hell, also known as the "Pyramid of Doom," refers to a situation in JavaScript where multiple nested callback functions make the code difficult to read and maintain. This typically happens when you have several asynchronous operations that depend on each other, leading to deeply nested callback functions.

```
let callBackHell = (callback) => {
  setTimeout(() => {
    callback();
  }, 1000);
};
callBackHell(() => {
  console.log("Function 1");
  callBackHell(() => {
    console.log("Function 2");
    callBackHell(() => {
      console.log("Function 3 ");
    });
  });
});
```

Problems with Callback Hell:

1. **Readability:** Deeply nested callbacks are hard to read and understand.
2. **Maintainability:** The code becomes difficult to maintain and modify.
3. **Error Handling:** Managing errors in nested callbacks can be complex and cumbersome.
4. **Scalability:** As the number of nested callbacks grows, the code becomes increasingly unmanageable.

Solutions to Callback Hell:

Promises

Async/Await

Promises

Definition: A promise is an object representing the eventual completion or failure of an asynchronous operation. It provides a way to handle asynchronous operations more gracefully compared to traditional callback-based approaches.

States of a Promise:

1. **Pending:** Initial state, neither fulfilled nor rejected.
2. **Fulfilled:** Operation completed successfully.
3. **Rejected:** Operation failed.

Creating a Promise:

A promise is created using the Promise constructor, which takes a function (executor) with two parameters: resolve and reject.

```
let getPromise = new Promise((resolve,reject)=>{
   console.log("This is Promise")
   resolve("Success")
   // reject("This is rejected")
})
```

Using Promises:

Promises are used with then, catch, and finally methods to handle the resolved or rejected state.

```
let newPromise = (number) => {
  return new Promise((resolve, reject) => {
    console.log(`This promise ${number}`)
    setTimeout(() => {
        resolve(`This is resolve ${number}`);
    }, 1000);
    // reject(`This is reject ${number}`)
  });
};
newPromise(1).then((res)=>{
   console.log(res)
}).catch((rej)=>{
   console.log(rej)
})
```

Chaining Promises:

You can chain multiple then methods to handle subsequent asynchronous operations.

```
let newPromise = (number) => {
  return new Promise((resolve, reject) => {
    console.log(`This promise ${number}`)
    setTimeout(() => {
        resolve(`This is resolve ${number}`);
    }, 1000);
    // reject(`This is reject ${number}`)
  });
};
newPromise(1).then((res) => {
  console.log(res);
  newPromise(2).then((res) => {
    console.log(res);
    newPromise(3).then((res) => {
      console.log(res);
    });
  });
});
```

Async/Await

Definition: async and await are syntactic sugar built on top of promises, making asynchronous code easier to write and read.

Async Functions:

An async function is a function that returns a promise. It allows the use of the await keyword to pause the execution of the function until a promise is resolved or rejected.

```
async function myFunction() {
    // Function body
}
```

Await:

The await keyword can only be used inside an async function. It pauses the execution of the function until the promise is resolved or rejected.

```
let newPromise = (number) => {
  return new Promise((resolve, reject) => {
    console.log(`This promise ${number}`)
    setTimeout(() => {
        resolve(`This is resolve ${number}`);
    }, 1000);
    // reject(`This is reject ${number}`)
  });
};

let asyncFunction = async function () {
  console.log("This is Async function 1");
  await newPromise(1);
}
asyncFunction();
```

Interview Questions:

Callback Hell

What is callback hell?

Answer: Callback hell, also known as the "Pyramid of Doom," refers to a situation in JavaScript where multiple nested callback functions make the code difficult to read, understand, and maintain. This typically occurs when you have several asynchronous operations that depend on each other, leading to deeply nested callbacks.

Why is callback hell considered a problem?

Answer: Callback hell is problematic because it makes code difficult to read, understand, and maintain. It increases the complexity of error handling and debugging, and it leads to deeply nested code that is hard to follow and modify.

How can you avoid callback hell?

Answer: There are several ways to avoid callback hell, including:

- **Named Functions:** Breaking down callbacks into separate named functions.
- **Promises:** Using promises to handle asynchronous operations in a more manageable way.
- **Async/Await:** Using async/await to write asynchronous code that looks synchronous.
- **Modularization:** Splitting code into smaller, reusable modules.

Promises

What is a Promise in JavaScript?

Answer: A promise is an object representing the eventual completion or failure of an asynchronous operation. It has three states: pending, fulfilled, and rejected.

How do you create a new Promise?

Answer: You create a new promise using the Promise constructor, which takes an executor function with resolve and reject parameters.

```
const myPromise = new Promise((resolve, reject) => {
  // Asynchronous operation
  if (success) {
    resolve('Operation successful');
  } else {
    reject('Operation failed');
  }
});
```

What are the methods available on a Promise object?

Answer: The primary methods are then, catch, and finally.

- then: Handles fulfillment.
- catch: Handles rejection.
- finally: Executes code regardless of the promise's outcome.

Explain promise chaining with an example.

Answer: Promise chaining allows you to perform a series of asynchronous operations in sequence.

```
let newPromise = (number) => {
  return new Promise((resolve, reject) => {
    console.log(`This promise ${number}`)
    setTimeout(() => {
        resolve(`This is resolve ${number}`);
    }, 1000);
    // reject(`This is reject ${number}`)
  });
};
newPromise(1).then((res) => {
  console.log(res);
  newPromise(2).then((res) => {
    console.log(res);
    newPromise(3).then((res) => {
      console.log(res);
    });
  });
});
```

Async/Await

What is the purpose of async and await in JavaScript?

Answer: async and await provide a way to work with promises more comfortably, allowing asynchronous code to be written in a synchronous-looking manner. An async function returns a promise, and await pauses the function execution until the promise is resolved or rejected.

How do you define an async function?

Answer: An async function is defined using the async keyword before the function declaration.

What does the await keyword do?

Answer: The await keyword pauses the execution of an async function until the promise is resolved or rejected, then returns the resolved value or throws the rejected error.

Explain how you would handle errors using async/await.

Answer: Errors in async/await can be handled using try...catch blocks.

```
let fetchData = async () {
   try {
      const response = await fetch('API');
      if (!response.ok) {
         throw new Error('Network response was not ok');
      }
      const data = await response.json();
      console.log(data);
   } catch (error) {
      console.error('Error fetching data:', error);
   }
}

fetchData();
```

Can you use await outside of an async function?

Answer: No, await can only be used inside an async function. Using it outside an async function will result in a syntax error.