1. Asynchronous JavaScript:

Asynchronous JavaScript refers to the ability of JavaScript to execute multiple tasks concurrently without blocking the main execution thread. It allows the program to continue running while waiting for long-running operations, such as network requests or file operations, to complete. This is crucial for building responsive and efficient web applications.

In JavaScript, asynchronous behavior is achieved through callbacks, promises, and async/await. By utilizing these mechanisms, you can initiate an asynchronous operation and provide a callback function or handle the result later when it becomes available.

2. Callbacks:

Callbacks are functions passed as arguments to other functions and are executed after a specific operation or event completes. They are commonly used in asynchronous programming to handle the result of an asynchronous operation.

Here's an example of using a callback to handle the result of an asynchronous operation, such as making an API request using the `fetch` function:

```
function fetchData(url, callback) {
  fetch(url)
    .then(response => response.json())
    .then(data => callback(null, data))
    .catch(error => callback(error, null));
}

fetchData('https://api.example.com/data', (error, data) => {
  if (error) {
    console.error('Error:', error);
  } else {
    console.log('Data:', data);
  }
});
```

In this example, the `fetchData` function takes a URL and a callback function. It performs an asynchronous `fetch` request and calls the callback with the error and data once the request is complete.

3. Promises:

Promises provide an alternative approach to handling asynchronous operations. A promise represents the eventual completion or failure of an asynchronous operation and allows you to attach callbacks to handle the result when it becomes available.

Here's an example of using promises to handle an asynchronous operation:

```
function fetchData(url) {
  return new Promise((resolve, reject) => {
    fetch(url)
      .then(response => response.json())
      .then(data => resolve(data))
      .catch(error => reject(error));
  });
}

fetchData('https://api.example.com/data')
  .then(data => console.log('Data:', data))
  .catch(error => console.error('Error:', error));
```

In this example, the `fetchData` function returns a promise that resolves with the data fetched from the URL. The `.then()` method is used to handle the resolved value, and the `.catch()` method is used to handle any errors.

4. Async-Await:

Async/await is a syntactic sugar built on top of promises, making asynchronous code look more like synchronous code. It allows you to write asynchronous code in a more sequential and readable manner.

Here's an example of using async/await to handle an asynchronous operation:

```
async function fetchData(url) {
  try {
    const response = await fetch(url);
    const data = await response.json();
    return data;
  } catch (error) {
    throw new Error(error);
  }
}

async function fetchDataExample() {
  try {
    const data = await fetchData('https://api.example.com/data');
    console.log('Data:', data);
  } catch (error) {
    console.error('Error:', error);
  }
}

fetchDataExample();
```

In this example, the `fetchData` function is defined as an async function. It uses the `await` keyword to pause the execution until the promise is resolved or rejected. The `fetchDataExample` function demonstrates how to call the async function and handle any errors using a try/catch block.

5. setTimeout:

`setTimeout` is a built-in JavaScript function that allows you to execute a callback function after a specified delay (in milliseconds). It is commonly used for introducing delays in code execution or scheduling a function to run once.

Here's an example of using `setTimeout`:

```
function delayedLog(message, delay) {
  setTimeout(() => {
    console.log(message);
  }, delay);
}

delayedLog('Delayed log after 2 seconds', 2000);
```

In this example, the `delayedLog` function logs a message after the specified delay of 2 seconds using `setTimeout`.

6. setInterval:

`setInterval` is a built-in JavaScript function that repeatedly executes a callback function with a fixed time interval between each execution. It is commonly used for creating timers or periodically performing tasks.

Here's an example of using `setInterval`:

```
let counter = 0;

function incrementCounter() {
  counter++;
  console.log('Counter:', counter);
}

setInterval(incrementCounter, 1000);
```

In this example, the `incrementCounter` function increments a counter variable and logs its value every second using `setInterval`.

7. Polyfill of setInterval:

A polyfill is a piece of code that provides the implementation of a feature or API that is not natively supported in a particular environment. To polyfill `setInterval`, you can use `setTimeout` recursively to achieve the same functionality.

Here's an example of a polyfill for `setInterval`:

```
function setIntervalPolyfill(callback, delay) {
  let timerId = setTimeout(function tick() {
    callback();
    timerId = setTimeout(tick, delay);
  }, delay);

  return {
    clear: function() {
      clearTimeout(timerId);
    }
  };
}

// Usage:
const interval = setIntervalPolyfill(() => {
  console.log('Interval callback');
}, 1000);

// To clear the interval:
interval.clear();
```

In this example, `setIntervalPolyfill` emulates the behavior of `setInterval` by using `setTimeout` recursively. It returns an object with a `clear` method to cancel the interval.

8. Implementing Promises:

You can implement a basic version of promises using native JavaScript constructs. Here's an example of a simplified implementation:

```
class MyPromise {
  constructor(executor) {
    this.status = 'pending';
    this.value = undefined;
    this.error = undefined;
    this.onResolveCallbacks = [];
    this.onRejectCallbacks = [];

    const resolve = (value) => {
      if (this.status === 'pending') {
        this.status = 'fulfilled';
        this.value = value;
        this.onResolveCallbacks.forEach(callback => callback(this.value));
      }
    };

    const reject = (error) => {
      if (this.status === 'pending') {
        this.status = 'rejected';
        this.error = error;
        this.onRejectCallbacks.forEach(callback => callback(this.error));
      }
    };

    try {
      executor(resolve, reject);
    } catch (error) {
      reject(error);
    }
  }

  then(onResolve, onReject) {
    const promise = new MyPromise((resolve, reject) => {
```

```
    if (this.status === 'fulfilled') {
      try {
        const result = onResolve(this.value);
        resolve(result);
      } catch (error) {
        reject(error);
      }
    }

    if (this.status === 'rejected') {
      try {
        const result = onReject(this.error);
        resolve(result);
      } catch (error) {
        reject(error);
      }
    }

    if (this.status === 'pending') {
      this.onResolveCallbacks.push((value) => {
        try {
          const result = onResolve(value);
          resolve(result);
        } catch (error) {
          reject(error);
        }
      });

      this.onRejectCallbacks.push((

error) => {
        try {
          const result = onReject(error);
          resolve(result);
        } catch (error) {
          reject(error);
        }
      });
    }
```

```
    });

    return promise;
  }

  catch(onReject) {
    return this.then(null, onReject);
  }
}
```

This is a simplified implementation of the `MyPromise` class, which supports resolving and rejecting a promise, chaining `then` and `catch` methods, and handling synchronous resolutions and rejections.

9. Promises and its functions:

Promises provide several built-in functions for working with asynchronous operations. Here are some commonly used promise functions:

- **`Promise.resolve(value)`:** Returns a resolved promise with the given value.

- **`Promise.reject(error)`:** Returns a rejected promise with the given error.

- **`Promise.all(promises)`:** Returns a promise that resolves when all the promises in the iterable argument have resolved, or rejects if any of the promises reject.

- **`Promise.race(promises)`:** Returns a promise that resolves or rejects as soon as any of the promises in the iterable argument resolves or rejects.

- **`Promise.allSettled(promises)`:** Returns a promise that resolves after all the promises in the iterable argument have settled (resolved or rejected), providing an array of objects with the outcome of each promise.

These functions allow you to handle multiple promises simultaneously and deal with their combined outcomes.

10. Fulfillment and Rejection of Promises:

Promises can be fulfilled or rejected based on the result of an asynchronous operation. The state of a promise can be one of the following:

- **Pending**: The initial state of a promise. It hasn't been fulfilled or rejected yet.

- **Fulfilled**: The state of a promise when it has been successfully fulfilled with a value.

- **Rejected**: The state of a promise when it has been rejected with an error.

Once a promise is fulfilled or rejected, it is considered settled and its state cannot be changed.

11. Microtask Queue:

The microtask queue is a queue that holds microtasks, which are tasks that need to be executed asynchronously in a separate phase of the event loop. Microtasks have higher priority than regular tasks and are executed immediately after the current task but before rendering.

Promises and `async/await` functions are examples of operations that enqueue microtasks. When a promise is resolved or rejected, the associated callbacks registered with `.then()` or `.catch()` are added to the microtask queue.

The microtask queue is processed completely before moving on to the next task in the event loop. This ensures that all microtasks are executed before any additional rendering or user interactions occur, providing a smoother user experience.

12. Callbacks vs Promises:

Callbacks and promises are both used to handle asynchronous operations, but they have different approaches and characteristics.

Callbacks require passing a function as an argument to another function to handle the asynchronous result. They can quickly lead to callback hell (discussed later) when dealing with multiple asynchronous operations or error handling. Additionally, error handling in callbacks can become cumbersome, as each callback needs to check for potential errors.

Promises, on the other hand, provide a more structured and cleaner approach. They represent the eventual completion or failure of an asynchronous operation and allow you to chain multiple operations together using `.then()` and `.catch()`. Promises also provide better error handling through the use of `.catch()`.

Promises offer a more readable and maintainable code structure compared to callbacks, especially when dealing with complex asynchronous operations or a large number of asynchronous calls.

13. Event Loop:

The event loop is a critical part of JavaScript's concurrency model. It manages the execution of code and handles asynchronous operations, ensuring that JavaScript remains responsive and non-blocking.

The event loop continuously checks for new tasks in a cycle. It consists of two main components: the call stack and the task queue.

- Call Stack: The call stack is a data structure that keeps track of function calls. It follows a Last-In-First-Out (LIFO) order. When a function is called, it is pushed onto the stack. When a function returns, it is popped off the stack.

- Task Queue: The task queue holds tasks (also known as "macrotasks") that are ready to be executed. These tasks usually include events like user input, network requests, and timers.

The event loop continuously checks the call stack and task queue. If the call stack is empty, it takes the first task from the task queue and pushes it onto the call stack for execution. This process continues in a loop, allowing asynchronous tasks to be processed while keeping the main execution thread available.

14. Callback Hell:

Callback hell, also known as the pyramid of doom, is a situation that arises when using multiple nested callbacks for asynchronous operations. It makes the code difficult to read, understand, and maintain due to excessive indentation and callback nesting.

Here's an example of callback hell:

```
asyncOperation1((error1, result1) => {
  if (error1) {
    console.error('Error:', error1);
  } else {
    asyncOperation2(result1, (error2, result2) => {
      if (error2) {
        console.error('Error:', error2);
      } else {
        asyncOperation3(result2, (error3, result3) => {
          if (error3) {
            console.error('Error:', error3);
          } else {
            // More nested callbacks...
          }
        });
      }
    });
  }
});
```

As you can see, each asynchronous operation requires a callback that triggers the next operation, leading to deeply nested code. This can make code harder to follow, debug, and maintain.

To avoid callback hell, you can use promises or async/await, which provide a more structured and readable way to handle asynchronous operations.

Prepared By :
Divyansh Singh (LinkedIn : rgndunes)