Callback Functions

- In JavaScript, a callback is a function that is passed as an argument to another function and is executed after some operation has been completed.
- The use of callbacks is particularly important in asynchronous programming, where certain tasks (like fetching data from a server or reading a file) take time to complete, and you don’t want the rest of the program to be blocked while waiting.

Why, Callbacks Introduced?

JavaScript is single-threaded, meaning it can only do one thing at a time. Early in JavaScript’s development, this single-threaded nature created a challenge: long-running operations (like network requests, file system access, or database queries) could freeze the browser or delay other important tasks.

Callbacks were introduced to solve the problem of blocking execution in JavaScript. They enable asynchronous programming, where some tasks are started and continue to run in the background, allowing other code to execute while waiting for a response.

Callbacks allow JavaScript to be non-blocking. For example, instead of waiting for an API request to finish before executing further code, a callback allows you to continue execution and process the result once the request is done.

Callbacks were introduced to allow non-blocking operations and concurrent execution without requiring the entire program to stop for the result of one task.

Synchronous and Asynchronous Callbacks in JavaScript

The primary difference between synchronous and asynchronous callbacks lies in when the callback is executed in relation to the main code.

a) Synchronous Callbacks

A synchronous callback is executed immediately as part of the main program flow. In other words, the callback is called and completed before the next line of code in the function is executed. Synchronous callbacks are typically used in operations that don't require waiting, such as mathematical calculations, sorting, or immediate result returns.

Example

```
function message(name) {
                    console.log(`Hello, ${name}!`);
}
function initialize(callback) {
                     const name = "Neha Mittal";
                     callback(name);  // Synchronous callback
}
initialize(message);
console.log("This statement gets executed after displaying the message.");
```

b) Asynchronous Callbacks

Asynchronous callbacks are executed after the current operation completes or at some point in the future. These are commonly used in operations that take time, such as network requests, reading files, or waiting for user input. Asynchronous callbacks do not block the execution of the remaining code, meaning the program can continue running other tasks while waiting for the callback to execute.

Example

```
function fetchData(callback) {
          console.log("Fetching data...");
           setTimeout(() => {
                      const data = "Welcome to Learn2Earn Labs";
                      callback(data);  // Asynchronous callback
           }, 2000); // delay of 2 seconds
}
function execute(data) {
                    console.log(data);
}
fetchData(execute);
console.log("This statement gets executed before the data is processed.");
```

note: The callback itself doesn't necessarily need to be asynchronous; the asynchronicity in this case comes from the setTimeout function, which simulates an asynchronous operation (like fetching data from an API).

Use Cases for Callbacks

a) **Asynchronous Programming:** Callbacks are most commonly used in asynchronous tasks, such as:
   - API calls (fetching data from a server)
   - Database queries
   - File system operations (in Node.js)
   - Event-driven programming (handling user input)
   - Timers (setTimeout, setInterval)

   Example

```
console.log('Hello Students');

setTimeout(function callback() {
  console.log('The future is unpredictable');
}, 2000);

console.log('You all need to know that');
```

b) **Event Handling:** JavaScript is widely used for event-driven applications (especially in the browser). Callbacks are used in event listeners to react to user actions like clicks, key presses, or scrolling.

   Example

```
document.getElementById('myButton').addEventListener('click',     function
handleClick() {
            console.log('Button clicked!');
});
```

   In above program, the callback is executed only when the button is clicked, allowing the program to remain responsive to other tasks.

c) **Array Methods:** Array methods like map(), filter(), and forEach() use callbacks to process each item in an array.

   Example with map()

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map((num) => num * 2);
console.log(doubled);
```

d) **Custom Iterations:** Callbacks are useful when you need to implement a custom iteration behaviour for functions.

Example

```
function display(n, callback) {
        for (let i = 0; i < n; i++) {
                callback(i);
        }
}

display(3, (i) => {
        console.log('Iteration:', i);
});
```

e) **Handling Multiple Operations:** Callbacks help manage complex workflows where multiple tasks depend on the results of other tasks.

Example Syntax (Nested Callbacks)

```
fetchData(function(result1) {
        execute(result1, function(result2) {
                display(result2, function() {
                        console.log('All tasks completed');
                });
        });
});
```

The above approach can lead to **callback hell** (we'll discuss later).

Where to Use Callbacks

- **Asynchronous Code:** Callbacks are best used in scenarios where you have to deal with asynchronous operations like Network requests (fetching data from an API), File system I/O (in Node.js), Time-based operations (e.g., setTimeout(), setInterval())
- **Event Handling:** When working with user interactions, such as clicks, keypresses, or form submissions, callbacks are perfect for processing user-driven events.
- **Custom Iterations and Array Methods:** If you need to perform custom logic during array processing (e.g., map(), filter()), callbacks are used to handle each item's transformation or filtering logic.

Where Not to Use Callbacks

- **Complex Chains of Callbacks (Callback Hell):** While callbacks are powerful, when they are nested too deeply (as shown in the handling multiple operations example in previous page), they create, callback hell, which leads to unreadable and hard-to-maintain code. This is a major drawback of using callbacks in JavaScript.
- **When Using Promises:** JavaScript's modern features such as Promises and async/await provide a cleaner alternative to callbacks, especially for handling asynchronous code. If you find your code is becoming cluttered with callbacks, switching to promises or async/await is recommended.

  Example Syntax
  ```
  fetchData()
    .then((result1) => processData(result1))
    .then((result2) => displayData(result2))
    .catch((error) => console.error(error));
  ```
  Promises flatten the callback structure and make it easier to handle asynchronous workflows.
- **Handling Errors:** Callbacks that only handle success paths are prone to missing error handling. Error-first callbacks help manage this issue, but modern patterns (like Promises) provide better mechanisms for managing errors.

Key Issues with Callbacks

- **Callback Hell (Pyramid of Doom):** This occurs when multiple callbacks are nested within one another, creating deeply nested structures that are hard to follow and debug.

  Solution: Promises and async/await can help eliminate callback hell by flattening code structure and improving readability.
- **Error Handling:** In callbacks, errors are often managed using the error-first callback pattern, where the first argument of the callback function is reserved for an error object.

  Example
  ```
  function fetchData(callback) {
                          const error = null;
                          const data = { id: 1, name: 'Tushar Gupta' };
                          if (error) {
                              callback(error, null);
  ```

```
        } else {
            callback(null, data);
          }
        }

fetchData((err, data) => {
          if (err) {
            console.error('Error:', err);
          } else {
            console.log('Data:', data);
          }
});
```

Using Promises or async/await simplifies error handling by using try/catch blocks.

- **Inversion of Control:** Callbacks can lead to inversion of control, where you hand over control of part of your program to a third party. This can create unpredictability and make debugging difficult.

Error-First Callbacks

- Error-first callbacks are a common pattern in JavaScript used primarily in asynchronous programming.
- This pattern allows functions to handle errors in a consistent manner while processing results.
- An error-first callback is defined as a function that receives the error as the first argument, followed by the result (if any). In this case, the function can easily determine whether an error occurred.
- Error-first callbacks are essential in asynchronous programming in JavaScript, allowing developers to handle errors effectively.
- The pattern provides a straightforward way to manage errors, ensuring that functions are robust and maintainable.

Key Characteristics of Error-First Callbacks

- **First Argument:** The first argument is always an error object (or null if there is no error).
- **Subsequent Arguments:** The remaining arguments contain the result(s) of the operation.
- **Consistency:** This pattern provides a consistent interface for handling errors across asynchronous operations.

Example

```
function fetchData(callback) {
                    const error = null;
                    const data = { id: 1, name: 'Neha Mittal' };
                    if (error) {
                              callback(error, null);
                    } else {
                              callback(null, data);
                    }  }
 fetchData((err, data) => {
                    if (err) {
                              console.error('Error:', err);
                    } else {
                              console.log('Data:', data);
                    }  });
```

Example: Error-First Callback

```
function execute(callback) {
            const isError = true
            if (isError) {
               return callback(new Error("generating an error"), null);    }
            callback(null, "Code Executing without any error.");
}
execute((error, result) => {
  if (error) {
          console.error("Error:", error.message);
  } else {
          console.log("Result:", result);
  }});
```

Callback Timers

- In JavaScript, timers are a powerful feature used to execute code after a specified period.
- They utilize callbacks, which are functions passed as arguments to be executed later.
- The primary timer functions in JavaScript are setTimeout() and setInterval().
- Using callbacks with timers is fundamental in JavaScript, enabling developers to create dynamic and responsive applications.

Key Timer Functions

a) setTimeout()

- Executes a function after a specified delay (in milliseconds).
- The callback is executed once after the delay.
- Additional arguments can be passed to the callback.

Syntax

```
setTimeout(callback, delay, arg1, arg2, ...)
```

b) setInterval()

- Repeatedly executes a function at specified intervals.
- The callback is executed repeatedly every interval milliseconds.
- Additional arguments can be passed to the callback.

Syntax

```
setInterval(callback, interval, arg1, arg2, ...)
```

c) clearTimeout()

- Cancels a timeout previously established by setTimeout().
- timeoutId is the identifier returned by setTimeout().

Syntax

```
clearTimeout(timeoutId)
```

d) clearInterval()

- Cancels an interval previously established by setInterval().
- intervalId (Id of interval) is the identifier returned by setInterval().

Syntax

```
clearInterval(intervalId)
```

Example: Using setTimeout()

```
function message() {
        console.log("Hello, Students!, welcome to learn2earn labs");
}
// Call function after 2 seconds
setTimeout(message, 2000);
```

Example: Passing Arguments to setTimeout()

```
function getDetails(name) {
        console.log(`Hello, ${name}!`);
}
setTimeout(getDetails, 3000, "Neha Mittal");
```

Example: Using setInterval()

```
let value = 1;
const intervalId = setInterval(() => {
                    console.log(`Value: ${value}`);
                    value++;
  // Stop the interval after 10 counts
  if (value >10) {
            clearInterval(intervalId);
            console.log("Interval cleared.");
  }
}, 1000);
```

Example: Chaining Timers

```
console.log("Executing...");
setTimeout(() => {
            console.log("First Statement");
            setTimeout(() => {
                    console.log("Second Statement");
            }, 5000);

}, 2000);
```

Example: Using setTimeout() with Anonymous Functions

```
let value = 0;
// Increment value every second
const intervalId = setInterval(() => {
                    value++;
                    console.log(`value: ${value}`);

                    // Stop after 5 seconds
                    if (value === 5) {
                            clearInterval(intervalId);
                            console.log("Exiting Iteration.");
                    }
                }, 1000);
```

Example: Delaying a Function Call with setTimeout()

```
function execute() {
                console.log("Welcome to Learn2Earn Labs, Agra");
}
const timeoutId = setTimeout(execute, 5000);
```

Callback with array functions

- In JavaScript, array methods often accept callback functions as arguments, allowing developers to perform operations on each element of an array.
- These callbacks provide a powerful and flexible way to manipulate and transform data within arrays.
- Using callbacks with array methods is a fundamental concept in JavaScript that allows for elegant and expressive manipulation of data structures.

Example: Using forEach()

The forEach() method executes a provided function once for each array element.

```
const numbers = [1, 2, 3, 4, 5];
numbers.forEach((number, index) => {
                console.log(`Index: ${index}, Value: ${number}`);
});
```

Example: Using map()

The map() method creates a new array populated with the results of calling a provided function on every element in the calling array.

```
const numbers = [1, 2, 3, 4, 5];
const result = numbers.map(number => number * 2);
console.log(result); // Output: [2, 4, 6, 8, 10]
```

Example: Using filter()

The filter() method creates a new array with all elements that pass the test implemented by the provided function.

```
const numbers = [1, 2, 3, 4, 5];
const result = numbers.filter(number => number % 2 === 0);
console.log(result); // Output: [2, 4]
```

Example: Using reduce()

The reduce() method executes a reducer function (that you provide) on each element of the array, resulting in a single output value.

```
const numbers = [1, 2, 3, 4, 5];
const result = numbers.reduce((accumulator, currentValue) => accumulator +
currentValue, 0);
console.log(result); // Output: 15
```

Example: Using find()

The find() method returns the value of the first element in the array that satisfies the provided testing function. Otherwise, it returns undefined.

```
const numbers = [1, 2, 3, 4, 5];
const result = numbers.find(number => number > 3);
console.log(result); // Output: 4
```

Example: Using some()

The some() method tests whether at least one element in the array passes the test implemented by the provided function. It returns a Boolean value.

```
const numbers = [1, 2, 3, 4, 5];
const result = numbers.some(number => number % 2 === 0);
console.log(result); // Output: true
```

Example: Using every()

The every() method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value.

```
const numbers = [2, 4, 6, 8, 10];
const result = numbers.every(number => number % 2 === 0);
console.log(result); // Output: true
```

Points to remember

- **Callbacks:** Array methods use callbacks to operate on elements, providing a functional approach to manipulate and process arrays.
- **Immutability:** Methods like map(), filter(), and reduce() do not modify the original array, promoting immutability.
- **Versatility:** These methods can be combined to perform complex operations, enhancing code readability and maintainability.