Asynchronous Operations

Instructor: Pramod Kumar Jena

---

1. Introduction to Asynchronous Operations

What is Asynchronous Programming?

Asynchronous programming allows tasks to be executed without blocking the execution of the program. In JavaScript, this is essential because operations like fetching data from a server, reading files, or waiting for user input can take time.

Example of Synchronous vs Asynchronous:

- **Synchronous Programming:** Tasks are executed one after another, and each task waits for the previous one to finish.

```
console.log("Task 1");
console.log("Task 2");
console.log("Task 3");
```

- **Asynchronous Programming:** Tasks can start and run in the background while the rest of the program continues executing.

```
console.log("Task 1");

setTimeout(() => {
  console.log("Task 2"); // Delayed execution (asynchronous)
}, 2000);

console.log("Task 3");
```

In the above asynchronous example, Task 3 will execute before Task 2, despite Task 2 being declared earlier. This is because `setTimeout()` is an asynchronous operation that runs in the background and allows other code to continue executing.

Real-Life Example 1: Ordering Food Online

Think about ordering food from a restaurant through an app. You place an order, but while waiting for the food to arrive, you can continue doing other tasks, such as browsing the internet or checking messages. You don't need to stop everything and wait until the food is delivered.

Similarly, in asynchronous programming, other operations can continue while waiting for something (like data or user input).

---

2. What are Promises?

Definition:

A **Promise** is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It acts as a placeholder for the future result of an operation.

States of a Promise:

1. **Pending:** The initial state; neither fulfilled nor rejected.
2. **Fulfilled:** The operation completed successfully.
3. **Rejected:** The operation failed.

Creating a Promise:

Here's the syntax for creating a promise:

```
let myPromise = new Promise((resolve, reject) => {
  // Asynchronous task
  let success = true;

  if (success) {
    resolve("Task completed successfully!");
  } else {
    reject("Task failed.");
  }
});
```

In the above example, `resolve()` is called if the task is successful, and `reject()` is called if the task fails.

Real-Life Example 2: Booking a Movie Ticket

Imagine booking a movie ticket online:

- **Pending:** You submit your request, and the system processes your booking.
- **Fulfilled:** If the seat is available, the system confirms the booking.
- **Rejected:** If the seat is not available, the system rejects the booking.

```
let bookTicket = new Promise((resolve, reject) => {
  let seatAvailable = true;

  if (seatAvailable) {
    resolve("Ticket booked successfully!");
  } else {
    reject("Sorry, no seats available.");
  }
});

bookTicket
  .then((message) => {
    console.log(message); // Ticket booked successfully!
  })
  .catch((error) => {
    console.log(error); // Sorry, no seats available.
  });
```

JavaScript is a **single-threaded** language, meaning it can only perform one task at a time in the main execution thread. However, using asynchronous functions like `setTimeout()` and `setInterval()`, JavaScript can schedule tasks for later while continuing to run other code without blocking the thread.

Example of Asynchronous Behavior:

```
console.log("Start");

setTimeout(() => {
  console.log("This runs after 3 seconds");
}, 3000);
```

```
console.log("End");
```

In this example:

- "Start" is printed first.
- The message "This runs after 3 seconds" is printed after the `setTimeout` delay.
- "End" is printed immediately after "Start" because the `setTimeout` function runs asynchronously.

---

2. What is `setTimeout`?

Definition:

`setTimeout()` is used to execute a function **once** after a specified delay (in milliseconds).

Syntax:

```
setTimeout(function, delay);
```

- **function**: The function to be executed after the delay.
- **delay**: The time in milliseconds before the function is executed.

Real-Life Example 1: Delayed Alert Message

Imagine you're on an e-commerce website, and after adding an item to your cart, a message pops up saying "Item added to cart" and then disappears after 3 seconds. This can be achieved with `setTimeout`.

```
function showAddedToCartMessage() {
  console.log("Item added to cart!");

  setTimeout(() => {
    console.log("Cart message disappears after 3 seconds");
  }, 3000);
}
```

```
showAddedToCartMessage();
```

Here, the message stays on screen and is automatically removed after 3 seconds, simulating a real-world user interface delay.

Use Case: Auto-Logout Warning

Many applications (like banking or financial services) automatically log you out after inactivity. They might show a warning message just before auto-logout, giving you 10 seconds to interact. If you don't, you're logged out.

```
function showLogoutWarning() {
  console.log("You will be logged out in 10 seconds due to
inactivity.");

  setTimeout(() => {
    console.log("Logging out now...");
    // Simulate logout functionality
  }, 10000); // 10 seconds
}

showLogoutWarning();
```

In this case, the `setTimeout` helps delay the auto-logout message, giving the user time to take action.

---

3. What is `setInterval`?

Definition:

`setInterval()` is used to repeatedly execute a function at specified time intervals. It keeps running until explicitly stopped with `clearInterval()`.

Syntax:

```
setInterval(function, interval);
```

- **function**: The function to execute repeatedly.
- **interval**: The time (in milliseconds) between each function call.

Real-Life Example 2: Updating a Clock

Consider the clock on your computer or phone. It continuously updates every second. You can implement a simple clock using `setInterval`.

```
function showTime() {
  let date = new Date();
  let time = date.toLocaleTimeString();
  console.log(time);
}

setInterval(showTime, 1000); // Update time every 1 second
```

Here, `setInterval` executes the `showTime` function every 1000 milliseconds (1 second) to display the current time.

Use Case: Stock Price Updates

Imagine a real-time stock trading application where prices need to be updated every few seconds. You can use `setInterval` to fetch and display the updated stock prices regularly.

```
function updateStockPrice() {
  // Simulate getting stock price from an API
  let stockPrice = (Math.random() * 1000).toFixed(2);
  console.log("Updated Stock Price: $" + stockPrice);
}

setInterval(updateStockPrice, 5000); // Fetch new stock price every 5
seconds
```

In this example, the stock price is fetched and displayed every 5 seconds, simulating real-time stock market updates.

---

4. Stopping the Timer

Stopping `setTimeout` with `clearTimeout`:

You can stop a `setTimeout` execution before the delay finishes using `clearTimeout()`.

```
let timeoutId = setTimeout(() => {
  console.log("This will not run");
}, 5000);

// Cancel the timeout
clearTimeout(timeoutId);
```

Stopping `setInterval` with `clearInterval`:

Similarly, you can stop a repeating `setInterval` function using `clearInterval()`.

```
let intervalId = setInterval(() => {
  console.log("This will not repeat");
}, 2000);

// Stop the interval after 6 seconds
setTimeout(() => {
  clearInterval(intervalId);
}, 6000);
```

In this example, the `setInterval` would normally print a message every 2 seconds, but after 6 seconds, it is stopped using `clearInterval()`.

---

5. Real-Life Example 3: Countdown Timer

Imagine creating a countdown timer for an event, such as a sale that ends in 10 seconds. After each second, the remaining time is updated.

```
function startCountdown(seconds) {
  let remainingTime = seconds;

  let countdown = setInterval(() => {
    if (remainingTime > 0) {
      console.log(`Time remaining: ${remainingTime} seconds`);
      remainingTime--;
    } else {
      console.log("Time's up!");
      clearInterval(countdown); // Stop the countdown
    }
  }, 1000); // Update every second
}

startCountdown(10);
```

Here, `setInterval` reduces the remaining time by 1 every second, and `clearInterval` stops the timer when the countdown reaches zero.

---

6. Nested `setTimeout` vs `setInterval`

You can create recurring actions with both `setTimeout` and `setInterval`, but there's a difference in their behavior.

Using `setInterval`:

```
setInterval(() => {
  console.log("Repeating task every 2 seconds");
}, 2000);
```

This will execute the task every 2 seconds, regardless of how long the task takes to complete.

Using `setTimeout` Recursively (for better control):

```
function repeatTask() {
  console.log("Repeating task every 2 seconds");

  setTimeout(repeatTask, 2000); // Call the function again after 2
seconds
}

repeatTask();
```

With a recursive `setTimeout`, you have better control over when the next execution happens, especially if the task takes time to complete. The next iteration won't start until the current one finishes, which ensures tasks don't overlap.

---

7. Exercises for Students

1. **Exercise 1:** Create a countdown timer that counts down from 30 seconds. When the countdown reaches 0, display a message saying "Countdown complete!".
2. **Exercise 2:** Create a function that prints a random motivational quote every 10 seconds using `setInterval()`. Stop the interval after 1 minute.
3. **Exercise 3:** Create an application that mimics an online food delivery tracker. Every 5 seconds, display the current status of the food (e.g., "Order received", "Food being prepared", "Out for delivery", "Delivered"). Stop updating after the food is delivered.