Lesson:

Timer functions - setTimeout, clearTimeout, setInterval, clearInterval

PW SKILLS

Topics

1. **setInterval()**
   - Introduction
   - Usecase Example
2. **clearInterval()**
   - Introduction
   - Usecase Example
3. **setTimeout()**
   - Introduction
   - Usecase Example
4. **clearTimeout()**
   - Introduction
   - Usecase Example

setInterval()

Introduction

This method repeats a given function at every given time interval and returns an interval ID that uniquely identifies the interval.

Syntax:

```
setInterval(function, delay_milliseconds);
```

function: the first parameter is the function to be executed
delay_milliseconds: indicates the length of the time interval between each execution.

A function to be executed every delay milliseconds. The first execution happens after a delay of the specified milliseconds.

There is also an option of passing arguments to the callback function in setInterval, for that the syntax would be:

```
setInterval(function,delay milliseconds, arg1, arg2……argN)
```

As you can see, you may pass as many arguments as required.

Let us look at a few examples for a better understanding

Example 1: Displaying messages with the help of setInterval()

```
const exSetInterval = setInterval(myRepeatedMessage, 300);
function myRepeatedMessage(){
 console.log('Hi');
 console.log('Hi Again !');
}

// OUTPUT
/*
Hi
Hi Again !
Hi
Hi Again !
Hi
Hi Again !
Hi
Hi Again !

... this continues
*/
```

Here, the function myRepeatedMessage() will keep on printing the messages 'Hi' and 'Hi Again' after every 300 milliseconds.

Example 2: Modify the above example by passing arguments to the function.

```
const exSetInterval = setInterval(myRepeatedMessage, 300, 'Hi', 'Hi
Again');
function myRepeatedMessage(a,b)
{
 console.log(a);
 console.log(b);
}

// OUTPUT
/*
Hi
Hi Again !
Hi
Hi Again !
Hi
Hi Again !
Hi
Hi Again !

... this continues
*/
```

Example 3: Displaying time with the help of setInterval and another helping function.

```
setInterval(timer, 1000);
function timer() {
  const date = new Date();
  const newTime = date.toLocaleTimeString();
  console.log(newTime);
}
// OUTPUT
/*
11:33:15 AM
11:33:16 AM
11:33:17 AM
11:33:18 AM
11:33:19 AM
11:33:20 AM
11:33:21 AM
11:33:22 AM
```

```
11:33:23 AM
11:33:24 AM
11:33:25 AM
11:33:26 AM
11:33:27 AM
11:33:28 AM
11:33:29 AM
11:33:30 AM
11:33:31 AM
11:33:32 AM
11:33:33 AM
11:33:34 AM
11:33:35 AM
... this continues
*/
```

Usecase Example: Asynchronously Fetch Data

Let's consider a scenario where you are working on a Node.js application that needs to perform a background task at regular intervals. For this example, let's say you want to simulate fetching data from an external API every 3 seconds.

```
// Node.js script example

// Simulated function to fetch data from an external API
function fetchDataFromAPI() {
  console.log('Fetching data from the API at:', new Date());
  // Simulated API call or data fetching logic
}

// Set an interval to fetch data from the API every 3 seconds
const dataFetchInterval = setInterval(fetchDataFromAPI, 3000);

// Stop fetching data after 15 seconds (just as an example)
setTimeout(function() {
  clearInterval(dataFetchInterval);
  console.log('Data fetching stopped after 15 seconds.');
}, 15000);
```

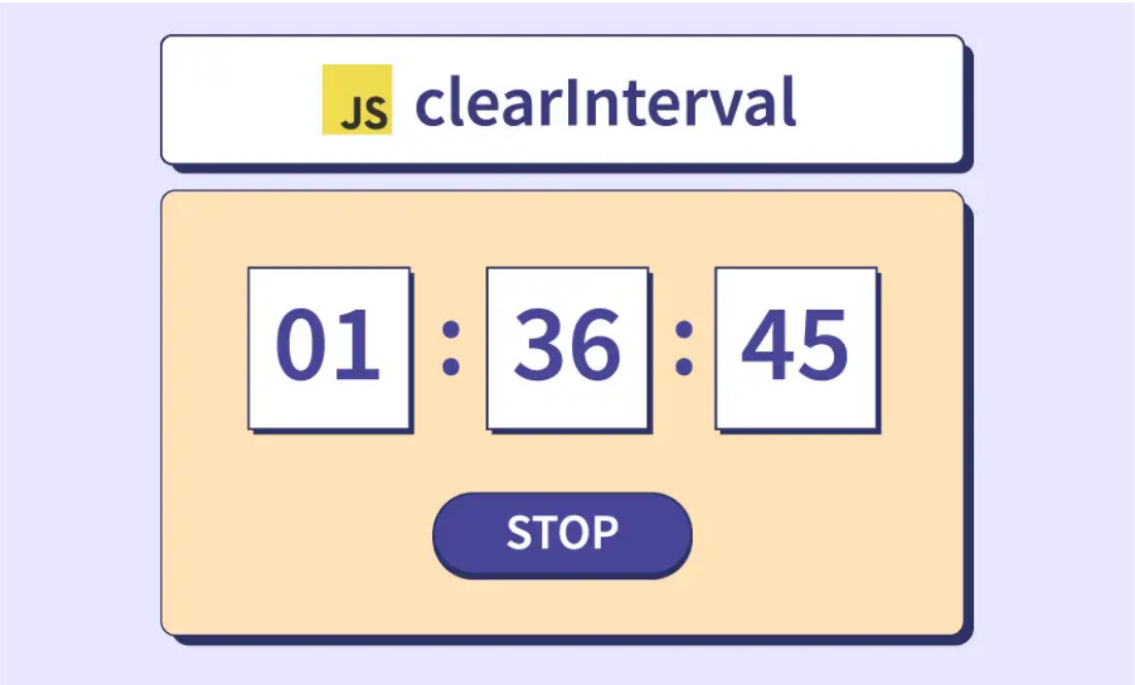
Output

Fetching data from the API at: 2023-11-16T08:25:16.226Z
Fetching data from the API at: 2023-11-16T08:25:19.241Z
Fetching data from the API at: 2023-11-16T08:25:22.243Z

In this example, the fetchDataFromAPI (simulated for this example). function is scheduled to run every 3 seconds using setInterval. It logs a message indicating the time of the API call.

clearInterval()

Introduction



The clearInterval() method cancels a timed, repeating action which was previously established by a call to setInterval().
If the parameter does not identify a previously established action, this method does nothing.

Syntax:

```
clearInterval(intervalID)
```

intervalID: The identifier of the repeated action you want to cancel. This ID was returned by the corresponding call to setInterval().

Example 4: How clearInterval works on setInterval()

```
var interval = setInterval(warning, 3000);
function warning() {
  console.log('3 second warning');
  clearInterval(interval);
}

// OUTPUT:
3 second warning
```

Usecase Example : Stop Countdown

Let's consider a Node.js script where you have a countdown timer, and you want to stop the countdown when a certain condition is met. For example, let's create a simple countdown that stops when the countdown reaches zero.

```
// Node.js script example

let countdown;
let secondsRemaining = 10;

// Function to update the countdown
function updateCountdown() {
  console.log(`Countdown: ${secondsRemaining} seconds remaining`);

  // Stop the countdown when it reaches zero
  if (secondsRemaining === 0) {
    stopCountdown();
  } else {
    secondsRemaining--;
  }
}

// Set an interval to update the countdown every second
countdown = setInterval(updateCountdown, 1000);

// Function to stop the countdown
function stopCountdown() {
  clearInterval(countdown);
  console.log("Countdown stopped.");
}
```

Output
Countdown: 10 seconds remaining
Countdown: 9 seconds remaining
Countdown: 8 seconds remaining
Countdown: 7 seconds remaining
Countdown: 6 seconds remaining
Countdown: 5 seconds remaining
Countdown: 4 seconds remaining
Countdown: 3 seconds remaining
Countdown: 2 seconds remaining
Countdown: 1 seconds remaining
Countdown: 0 seconds remaining
Countdown stopped.

In this example, the **updateCountdown** function is scheduled to run every second using **setInterval**. It logs the remaining time in the countdown. When the countdown reaches zero, the **stopCountdown** function is called, which uses **clearInterval** to stop the interval.

This pattern allows you to create a countdown timer that stops when a specific condition is met, providing flexibility for various use cases.

setTimeout()

Introduction

The setTimeout() method sets a timer that executes a function or specified piece of code once the timer expires.

Syntax:

```
setTimeout(function,delay,argument1, argument2…….argumentN)
```

function: is executed after the timer expires

delay: in milliseconds is the time for which the timer should wait before the specified function or code is executed. If this parameter is omitted, a value of 0 is used, meaning execute immediately.

arguments: Additional arguments which can be passed through to the function specified here.

The returned timeoutID is a positive integer value which is an identifier for the timer created by the call to setTimeout. We can use this identifier to cancel the timer using clearTimeout.

setTimeout() is an asynchronous function which simply means that the timer function will not pause the execution of other functions while the timer is running.

Example 1 : Display a set of messages using setTimeout()

```
setTimeout(() => {console.log("First message")}, 1000);
setTimeout(() => {console.log("Second message")}, 2000);
setTimeout(() => {console.log("Third message")}, 3000);
```

Output:

```
First message
Second message
Third message
```

In the above code, the first time-out function after a 1-second delay will be executed first, the second timeout function will be executed after 2 second delay and the third function will execute after 3 second delay.

Example 2 :

```
console.log("First message without timeout")
setTimeout(() => {console.log("First message with timeout")},
1000);
console.log("Second message without timeout")
```

```
setTimeout(() ⇒ {console.log("Second message with timeout")},
2000);
console.log("Third message without timeout")
setTimeout(() ⇒ {console.log("Third message with time out")},
3000);
```

Output:

```
First message without timeout
Second message without timeout
Third message without timeout
First message with timeout
Second message with timeout
Third message with time out
```

Here in the above example,

- The line console.log("First message without timeout"); is executed first, and it immediately logs the message "First message without timeout" to the console.
- The line setTimeout(() => {console.log("First message with timeout")}, 1000); schedules the execution of a callback function after a delay of 1000 milliseconds (1 second). This means that the callback function, which logs the message "First message with timeout", will be executed after a 1-second delay.
- The line console.log("Second message without timeout"); is executed next and logs the message "Second message without timeout" to the console immediately.
- The line setTimeout(() => {console.log("Second message with timeout")}, 2000); schedules the execution of a callback function after a delay of 2000 milliseconds (2 seconds). This callback function logs the message "Second message with timeout".
- The line console.log("Third message without timeout"); is executed and logs the message "Third message without timeout" to the console immediately.
- The line setTimeout(() => {console.log("Third message with time out")}, 3000); schedules the execution of a callback function after a delay of 3000 milliseconds (3 seconds). The callback function logs the message "Third message with timeout".

Example 3 : setTimeout() with 0 delay

```
console.log("First message");
```

```
setTimeout(() => {
  console.log("Second message");
}, 0);

console.log("Third message");
```

In this code, we have three console.log statements and a setTimeout call with a delay of 0 milliseconds.

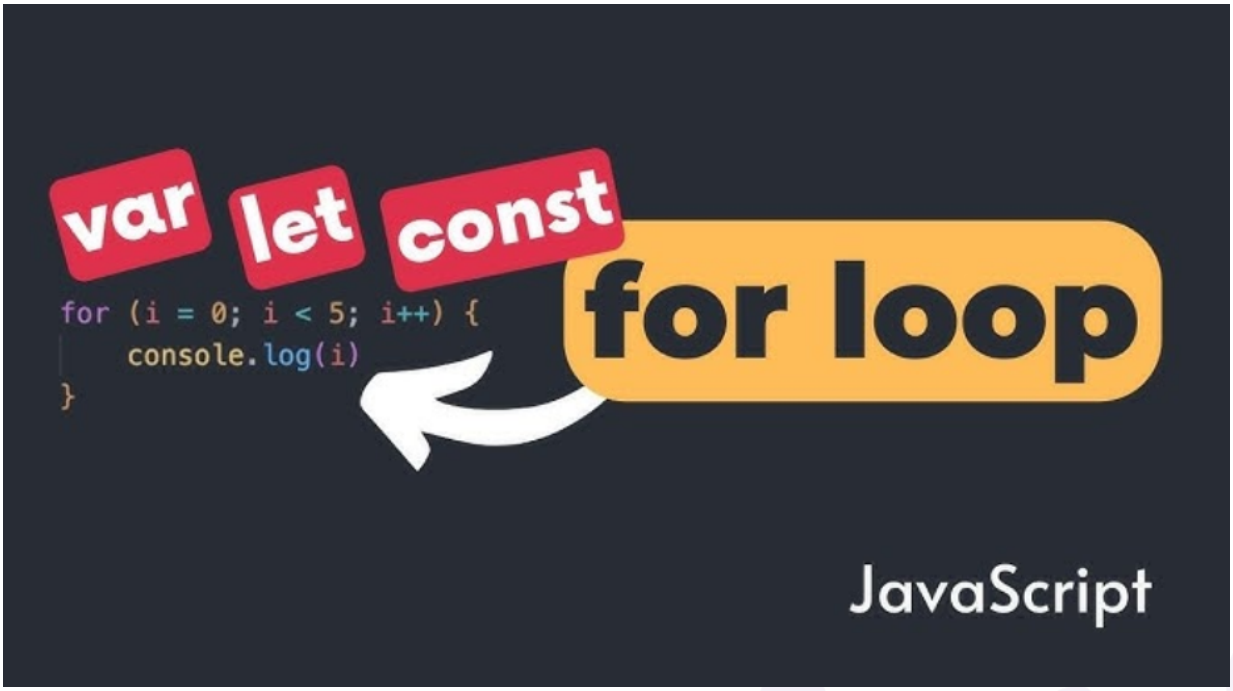
Output:

```
First message
Third message
Second message
```

Here in the above example,

- The first **console.log("First message");** statement is executed, and it immediately logs the message "First message" to the console.
- The **setTimeout** function is then called with a callback function and a delay of 0 milliseconds. Despite the 0-millisecond delay, the callback function is not executed immediately. Instead, it is queued to be executed by the event loop as soon as possible after the current task is completed.
- The execution continues to the next line, which logs the message "Third message" to the console.
- After the current task is completed, the event loop picks up the queued callback function from the setTimeout call with a delay of 0 milliseconds. It then executes the callback, logging the message "Second message" to the console.

It's important to note that even though the delay is set to 0 milliseconds, the callback function is not executed immediately. Instead, it is placed in the event loop and scheduled to be executed in the next available task. This demonstrates the non-blocking nature of asynchronous operations in JavaScript and how they are managed by the event loop.



Example 4: Tricky loop

```
for (var i = 1; i ⩽ 5; i++) {
  setTimeout(() ⇒ {
    console.log(i);
  }, 3000);
}
```

Output:

```
6
6
6
6
6
```

The intention of the code appears to be printing the numbers 1 to 5 with a delay of 3 seconds (3000 milliseconds) between each number. However, due to the behavior of closures and the **"var"** keyword having functional scope, the output will not be as expected.

The issue lies in the fact that the arrow function passed to **setTimeout** retains a reference to the variable **i**, and by the time the function is executed after the delay time which is 3 seconds here, the loop has already completed, leaving **i** with a value of 6. As a result, when the arrow function is invoked, it will print 6 for all iterations.

So how we can fix this issue?

Method 1: We can use “let” instead of “var”.

```
for (let i = 1; i ⩽ 5; i++) {
  setTimeout(() ⇒ {
    console.log(i);
  }, 3000);
}
```

OUTPUT:

```
1
2
3
4
5
```

By using let instead of var, each iteration of the loop will have its own block-scoped variable i, ensuring that the correct value is captured and logged after the delay. With this modification, the expected output will be the numbers 1 to 5, each printed after a delay of 3 seconds.

Method 2: We can pass the parameter to the function inside setTimeout instead of relying on the **"var"** keyword. This can be achieved by creating a separate scope using an Immediately Invoked Function Expression **(IIFE)**

```
for (var i = 1; i ⩽ 5; i++) {
  (function(num) {
    setTimeout(() ⇒ {
      console.log(num);
    }, 3000);
  })(i);
}
```

OUTPUT:

```
1
2
3
4
5
```

In this updated code, an IIFE is used to create a new scope for each iteration of the loop. The current value of i is passed as an argument **num** to the IIFE, ensuring that each iteration has its own copy of num. Within the arrow function passed to **setTimeout**, the correct value of num is logged after the specified delay.

Usecase Example: Schedule Download Operation with delay

Let's consider a Node.js script where you want to simulate a scenario where a user initiates a download, and you want to show a message indicating that the download will start in a few seconds. You can use **setTimeout** to delay the display of the message.

```
// Node.js script example

// Function to simulate starting a download
function startDownload() {
  console.log('Download started. Please wait...');
  // Simulated download logic
}

// Function to display a message and start the download after a
delay
function initiateDownload() {
  console.log('Initiating download. The download will start in 3
seconds.');

  // Introduce a delay of 3 seconds before starting the download
  setTimeout(() ⇒ startDownload(), 3000);
}

// Call the function to initiate the download
initiateDownload();
```

OUTPUT:

Initiating download. The download will start in 3 seconds.
Download started. Please wait...

In this example, the **initiateDownload** function displays a message indicating that the download will start in 3 seconds. The actual download is then initiated using **startDownload** after a delay of 3 seconds using setTimeout.

This pattern can be useful in scenarios where you want to provide feedback or notifications to users and introduce delays before performing certain actions in a Node.js script or server-side application.


```
clearTimeout(timeoutID)
```

```
const timeoutId = setTimeout(function(){
    console.log("Hi");
}, 2000);
```

```
clearTimeout(timeoutId);
console.log(`Timeout ID ${timeoutId} has been cleared`);
```

```
Timeout ID 2 has been cleared
```

```
// Node.js script example

let callCount = 0;
let rateLimitTimeout;

// Function to simulate an operation (in this case, logging a
message)
function performOperation() {
  console.log('Operation performed at:', new Date());
}

// Function to implement rate limiting logic
function rateLimitOperation() {
  // Increment the call count
  callCount++;

  // Check if the call count exceeds the limit (e.g., limit to 3
calls within 5 seconds)
  if (callCount > 3) {
    console.log('Rate limit exceeded. Please wait...');
    return;
  }

  // Clear the previous timeout (if any) to reset the delay
  clearTimeout(rateLimitTimeout);

  // Set a new timeout for 5 seconds to reset the call count
  rateLimitTimeout = setTimeout(() => {
    callCount = 0;
  }, 5000);
```

```
  // Execute the operation
  performOperation();
}

// Simulate calling the rate-limited function multiple times
rateLimitOperation();
rateLimitOperation();
rateLimitOperation();
rateLimitOperation();
rateLimitOperation();

// Only three operations will be executed within 5 seconds due to
rate limiting
```

Output:

Operation performed at: 2023-11-16T08:43:31.891Z
Operation performed at: 2023-11-16T08:43:31.895Z
Operation performed at: 2023-11-16T08:43:31.895Z
Rate limit exceeded. Please wait...
Rate limit exceeded. Please wait...

In this example, the **rateLimitOperation** function simulates a rate-limited operation (e.g., logging a message) with a limit of 3 calls within a 5-second window. If the function is called more frequently than allowed, it logs a rate limit exceeded message. The **setTimeout** and **clearTimeout** combination is used to reset the call count after a specified interval. This pattern is useful for scenarios where you want to control the rate of certain actions to avoid resource exhaustion or API rate limits.

Interview Point

1. Find the output of the following code

```
for (var i = 1; i ⩽ 16; i=i*2) {
  (function(num) {
    setTimeout(() ⇒ {
      console.log(num);
    }, 3000);
  })(i);
}
```

Ans:
1
2
4
8
16

2. Find the output of the following code

```
let count = 0;

function printCount() {
  console.log("Interval Count:", count);
  count++;
}

console.log("Start");

const intervalId = setInterval(printCount, 500);

setTimeout(function() {
      clearInterval(intervalId);
      console.log("Interval stopped");
}, 2000);

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

Ans:
Start
End
Interval Count: 0
Interval Count: 1
Interval Count: 2
Interval Count: 3
Interval stopped