function findWords(str) {

console.log("Number of words"); // This line executes immediately.

let cnt = 0; // Initializes the count to 0.

setTimeout(() => {

cnt = str.split("\\s").length; // This line will execute after a delay of 2000 milliseconds (2 seconds).

}, 2000);

console.log(cnt); // This line executes immediately after the setTimeout is set up, before the asynchronous task completes.

}

findWords("welcome to Javascript");

**Notes:**

1. **Synchronous Execution**:
   * The lines console.log("Number of words");, let cnt = 0;, and console.log(cnt); are executed synchronously, meaning they run in the order they appear without waiting for any other operations to complete.
   * console.log(cnt); will log 0 because the asynchronous setTimeout has not yet completed.
2. **Asynchronous Execution**:
   * The setTimeout function is used to schedule a task to be executed after a specified delay (2000 milliseconds in this case).
   * The code inside the setTimeout callback (cnt = str.split("\\s").length;) will run after the delay, independently of the main execution flow.
   * This means that the console.log(cnt); outside the setTimeout will execute before the setTimeout callback, logging 0 instead of the actual word count.
3. **Impact of Asynchronous Code**:
   * Asynchronous code like setTimeout does not block the execution of subsequent lines of code. This is why console.log(cnt); logs 0 before the word count is updated.
   * To see the correct word count, you should log cnt inside the setTimeout callback.

**Corrected Example:**

To ensure the word count is logged correctly after the asynchronous operation, you can move the console.log(cnt); inside the setTimeout callback:

function findWords(str) {

console.log("Number of words");

let cnt = 0;

setTimeout(() => {

cnt = str.split(/\s+/).length; // Corrected the regex to properly split by whitespace

console.log(cnt); // Logs the correct word count after 2 seconds

}, 2000);

}

findWords("welcome to Javascript");

This way, the word count will be logged after the asynchronous task completes.

function findWords(sentence,callback){

  let cnt=0;

  console.log("Number of words");

   setTimeout(()=>{

     cnt=sentence.split(" ").length;

   },2000)

   callback(cnt);

}

function displayCount(cnt){

  console.log(cnt);

}

findWords("Welcometo Javascript",displayCount);

/\* We can do asynchronous tasks using promises in a better way \*/

Promises in JavaScript are objects that represent the eventual completion (or failure) of an asynchronous operation and its resulting value. They provide a way to handle asynchronous tasks more efficiently by using then for resolved values and catch for errors. Callbacks are functions passed as arguments to other functions, allowing asynchronous code to execute after a certain task is completed. They are often used in older code to handle asynchronous operations before promises were introduced.

// 1. Creating a Promise object by passing a function as an argument.

// This function implicitly receives 'resolve' and 'reject' arguments, which are methods used to send the output.

let p = new Promise((res, rej) => {

// Implementation of asynchronous task will go here

});

// 2. Within the arrow function, we can implement asynchronous tasks using setTimeout.

let p = new Promise((res, rej) => {

setTimeout(() => {

res("msa"); // 'res' method is called to resolve the promise with the value "msa"

}, 2000); // The task will be executed after a delay of 2000 milliseconds (2 seconds)

});

// 3. How to call the promise

p.then((msg) => {

// 'msg' is the argument passed by the 'resolve' or 'reject' methods

console.log(msg); // Logs the resolved value

}).catch((msg) => {

// 'msg' is the argument passed by the 'reject' method

console.log(msg); // Logs the rejected value

});

/\* Example with rejection \*/

let p = new Promise((res, rej) => {

setTimeout(() => {

// res("welcome"); // Uncomment this line to resolve the promise

rej("Welcome mn"); // 'rej' method is called to reject the promise with the value "Welcome mn"

}, 2000);

});

p.then((m) => {

console.log(m); // This will log the resolved value if the promise is resolved

}).catch((m) => {

console.log(m); // This will log the rejected value if the promise is rejected

});

**Notes:**

1. **Creating a Promise**:
   * A Promise object is created by passing a function to the Promise constructor. This function receives two arguments: resolve and reject.
   * resolve is used to mark the promise as successfully completed and pass a value.
   * reject is used to mark the promise as failed and pass an error or rejection reason.
2. **Implementing Asynchronous Tasks**:
   * Inside the function passed to the Promise constructor, you can implement asynchronous tasks using methods like setTimeout.
   * In the example, setTimeout is used to simulate an asynchronous task that completes after 2000 milliseconds (2 seconds).
   * Depending on the outcome of the task, you can call resolve or reject to complete the promise.
3. **Calling the Promise**:
   * The then method is used to handle the resolved value of the promise. It takes a callback function that receives the resolved value as an argument.
   * The catch method is used to handle the rejected value of the promise. It takes a callback function that receives the rejection reason as an argument.
   * In the example, p.then((m) => { console.log(m); }).catch((m) => { console.log(m); }); will log the resolved or rejected value based on the outcome of the promise.

This should help you understand how to use promises for asynchronous tasks in JavaScript.

function findWords(sentence){

  return new Promise((res,rej)=>{

    setTimeout(()=>{

      if(sentence.length!=0)

        res(sentence.split(" ").length);

      else

        rej("Invalid String");

    },1000);

  });

}

findWords("").then((cnt)=>{console.log(cnt)})

.catch((msg)=>{

  console.log(msg);

})

**Explanation and Notes**

**Async Function Example**

async function display() {

// Task 1

await getCities();

await getHotel();

// Task 2

}

**Explanation**:

* **Async Function**: The async keyword is used to declare an asynchronous function. Inside this function, you can use the await keyword to wait for promises to resolve.
* **Await**: The await keyword pauses the execution of the display function until the promise returned by getCities and getHotel is resolved. This allows you to write asynchronous code that looks and behaves like synchronous code.

**Cohort Details and Count Example**

function cohortDetails() {

let cohorts = [];

return new Promise((res, rej) => {

setTimeout(() => {

cohorts.push({ id: "QEA001", name: "Cypress" });

cohorts.push({ id: "QEA002", name: "Selenium" });

res(cohorts);

}, 1000);

});

}

function getCount(cohortId) {

let cohortsCount = [];

return new Promise((res, rej) => {

setTimeout(() => {

cohortsCount.push({ id: "QEA001", cnt: 30 });

cohortsCount.push({ id: "QEA002", cnt: 27 });

res(cohortsCount.find((c) => c.id === cohortId));

}, 2000);

});

}

async function cohortsDisplay() {

let c = await cohortDetails();

console.log(c);

let cnt = await getCount("QEA001");

console.log(cnt);

}

cohortsDisplay();

**Explanation**:

* **Promises**: Both cohortDetails and getCount functions return promises. These promises simulate asynchronous operations using setTimeout.
* **cohortDetails Function**: This function creates a promise that resolves with an array of cohort objects after 1 second.
* **getCount Function**: This function creates a promise that resolves with the count of a specific cohort based on the cohortId after 2 seconds.
* **Async/Await in cohortsDisplay**:
  + await cohortDetails(): Waits for the cohortDetails promise to resolve and assigns the result to c.
  + await getCount("QEA001"): Waits for the getCount promise to resolve and assigns the result to cnt.
  + The results are then logged to the console.

**Key Points:**

* **Async/Await**: Simplifies working with promises, making asynchronous code easier to read and write.
* **Promises**: Used to handle asynchronous operations, allowing you to perform tasks like fetching data or waiting for a timeout.
* **Error Handling**: In a real-world scenario, you should include error handling using try...catch blocks to manage any errors that occur during the asynchronous operations.