**Synchronous vs. Asynchronous JavaScript**

Synchronous behavior is analogous to sequential processor instructions in memory. Each instruction is executed strictly in the order in which it appears, and each is also capable of immediately retrieving information that is stored locally within the system (for example: in a processor register or in system memory). As a result, it is easy to reason about the program state (for example, the value of a variable) at any given point in code.

A trivial example of this would be performing a simple arithmetic operation:

let x = 3;

x = x + 4;

At each step in this program, it is possible to reason about the state of the program because execution will not proceed until the previous instruction is completed. When the last instruction completes, the computed value of x is immediately available for use.

Conversely, asynchronous behavior is analogous to interrupts, where an entity external to the current process is able to trigger code execution. An asynchronous operation is often required because it is infeasible to force the process to wait a long time for an operation to complete (which is the case with a synchronous operation). This long wait might occur because the code is accessing a high-latency resource, such as sending a request to a remote server and awaiting a response.

A trivial JavaScript example of this would be performing an arithmetic operation inside a timeout:

let x = 3;

setTimeout(() => x = x + 4, 1000);

For the value of x to become useful, this asynchronously executed function would need to signal to the rest of the program that it has updated the value of x. However, if the program does not need this value, then it is free to proceed and do other work instead of waiting for the result.

Returning Asynchronous Values

function double(value, success, failure) {

setTimeout(() => {

**try {**

**if (typeof value !== 'number') {**

**throw 'Must provide number as first argument';**

**}**

**success(2 \* value);**

**} catch (e) {**

**failure(e);**

**}**

}, 1000);

}

**const successCallback = (x) => console.log(`Success: ${x}`);**

**const failureCallback = (e) => console.log(`Failure: ${e}`);**

double(3, successCallback, failureCallback);

double('b', successCallback, failureCallback);

**Promise Basics**

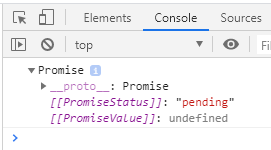
As of ECMAScript 6, Promise is a supported reference type and can be instantiated with the new operator. Doing so requires passing an executor function parameter which here is an empty function object to please the interpreter:

let p = new Promise(() => {});

setTimeout(console.log, 0, p); // Promise <pending>

If an executor function is not provided, a SyntaxError will be thrown.

When passing a promise instance to console.log, the console output (which may vary between browsers) indicates that this promise instance is pending.



A promise is a stateful object that can exist in one of three states:

➤➤ Pending

➤➤ Fulfilled (sometimes also referred to as resolved)

➤➤ Rejected

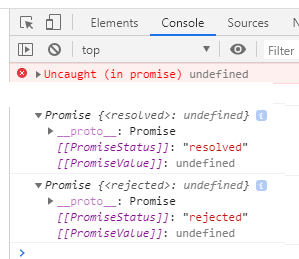
A pending state is the initial state a promise begins in. From a pending state, a promise can become settled by transitioning to a fulfilled state to indicate success, or a rejected state to indicate failure. This transition to a settled state is irreversible; once a transition to either fulfilled or rejected occurs, the state of the promise can never change

let p1 = new Promise((resolve, reject) => resolve());

setTimeout(console.log, 0, p1); // Promise <resolved>

let p2 = new Promise((resolve, reject) => reject());

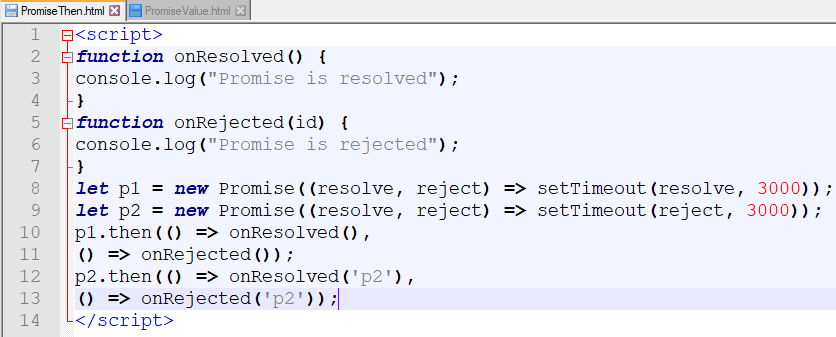
setTimeout(console.log, 0, p2); // Promise <rejected>

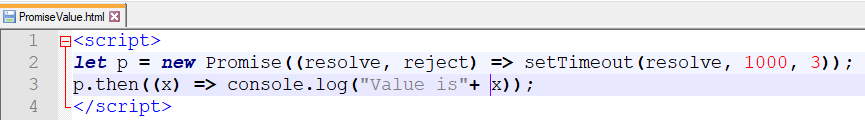


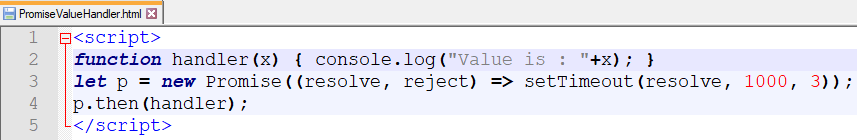
**Promise Instance Methods**

These methods can be used to access data returned from an asynchronous operation, handle success and failure outcomes of the promise, serially evaluate promises, or add functions that only execute once the promise enters a terminal state.

The method Promise.prototype.**then() is the primary method that is used to attach handlers to a promise instance.** The then() method accepts up to two arguments: an optional onResolved handler function, and an optional onRejected handler function. Each will execute only when the promise upon which they are defined reaches its respective "fulfilled" or "rejected" state.

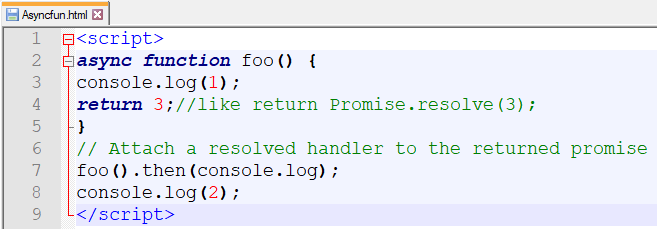




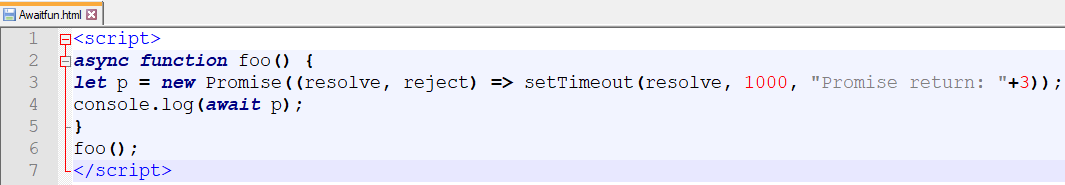


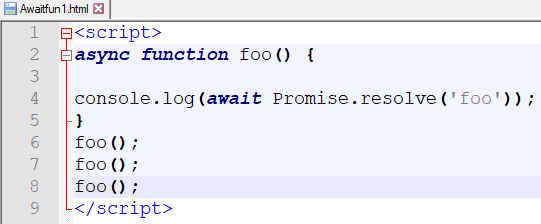
**Async Function**

An async function can be declared by prepending the async keyword. In an async function, whatever value is returned with the return keyword (or undefined if there is no return) will be effectively converted into a promise object with Promise.resolve(). An async function will always return a promise object.



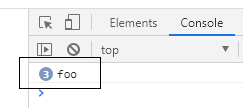
**The await keyword**, which is used to pause execution while waiting for a promise to resolve.





Here in async fun we have retrun promise obect with resolve status and value is “foo”

If u see output of above in console u will get



It indicates foo is returned 3 times.

The await keyword must be used inside an async function.

**Push vs. Pull**:

If you are familiar with the Iterator pattern, you know that in this case, you are in charge. When you want new value, you just call the next method to pull the value.

var it = makeIterator(['yo', 'ya']);

console.log(it.next().value); // 'yo'

With Observable it’s like, don’t call us we call you.

The Observable is the boss. When he has a new value, he will push the value to you. Your job is just to “listen.”

A key thing to understand when using observables is that observables push. Push and pull are two different ways that describe how a *data producer*communicates with the *data consumer*.

**Pull**When pulling, the data consumer decides when it get’s data from the data producer. The producer is unaware of when data will be delivered to the consumer.

Every javascript function uses the pull. The function is a Producer of data, and the code that calls the function is consuming it by “pulling” out a *single* return value from its call.

**Push**When pushing, it works the other way around. The data producer (the creator of the newsletter) decides when the consumer (the subscriber to the newsletter) gets the data.

Promises are the most common way of push in JavaScript today. A promise (the producer) delivers a resolved value to registered callbacks (the consumers), but unlike functions, it is the promise which is in charge of determining precisely when that value is “pushed” to the callbacks.

**Promises**

A Promise is a data structure that represents a value that might not be immediately available. This is very useful when you want to perform some action as soon as that value does become available. An example use case is when you have made an HTTP request: at a certain moment, the response is going to come in, at which point you will want to perform some action. A Promise can then represent that eventual response.