8.13 Promises

Synchronous and asynchronous functions

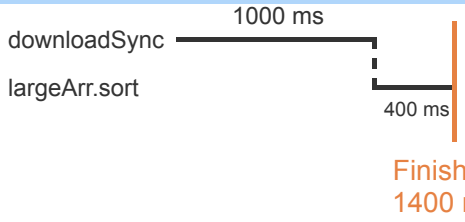
A ***synchronous function*** is a function that completes an operation before returning. Ex: A function to sort an array will return only after the entire array is sorted.

An ***asynchronous function*** is a function that starts an operation and potentially returns before the operation completes. The operation completes in the background, allowing other code to execute in the meantime. Ex: A function to download data is often implemented asynchronously, allowing other code to execute while the download runs in the background. A callback function is commonly passed to an asynchronous function and is called when the operation completes.

PARTICIPATION ACTIVITY | 8.13.1: Asynchronous functions are commonly used to download data in the background.

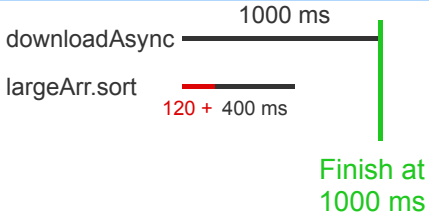
1 2 3 4 5 6 7 2x speed

```
function doWork(downloadURL, largeArr) {
   let data = downloadSync(downloadURL);
   largeArr.sort();
   // do something with data and largeArr
}
```




```
function doWork(downloadURL, largeArr) {
   downloadAsync(downloadURL, listener);
   largeArr.sort();
}

function listener() {
   // do something with data and largeArr
}
```



downloadAsync() calls the listener function when the download completes. The array sort executed concurrently with the download, so only 1000 ms were required for both operations.

Captions

1. Downloading data from the web is often a slow task. The doWork() implementation on the left uses downloadSync() to synchronously download data.
2. The doWork() implementation on the right uses downloadAsync() to download data asynchronously.
3. If the data takes 1000 milliseconds (ms) to download, the downloadSync() call returns after 1000 ms.
4. After the download, sorting the array takes another 400 ms. The entire function finishes at 1400 ms.

5. The asynchronous download function starts the download but returns just after starting. The download still takes 1000 ms.
6. downloadAsync() takes 120ms to start the operation, but then returns. So the array sort can start after about 120 ms. The download continues in the background.
7. downloadAsync() calls the listener function when the download completes. The array sort executed concurrently with the download, so only 1000 ms were required for both operations.

Feedback?

PARTICIPATION ACTIVITY 8.13.2: Asynchronous function result data.

1) In the animation above, the `data` variable is declared in the synchronous `doWork()` function, but not the asynchronous. The data variable in the asynchronous code should be _____.

- declared as a global variable
- passed as an argument to the `listener()` function
- declared as a local variable in `doWork()`, the same as the synchronous version

Correct

Asynchronous functions commonly pass the result of the operation as an argument to the listener function.

Feedback?

PARTICIPATION ACTIVITY 8.13.3: Synchronous and asynchronous functions.

Consider the following code.

```
function doWork2(downloadURL) {
   let dataArray = download(downloadURL);
   dataArray.sort();
}
```

1) Suppose the `doWork2()` function is intended to download an array of data and then sort the downloaded array. For proper functionality, the download function _____.

- (•) must be synchronous
- ( ) must be asynchronous
- ( ) can be either synchronous or asynchronous

Correct

The entire array contents must be downloaded before being sorted. The download completing before the sort begins is guaranteed only if `download` is synchronous.

2) Assume the `download()` function is synchronous but the `sort()` function is asynchronous. If no errors occur, then by the time `doWork2()` finishes, _____.

- ( ) the data array is guaranteed to be downloaded and sorted
- (•) the data array is guaranteed to be downloaded, but may not be sorted
- ( ) the data array is not guaranteed to be downloaded or sorted

Correct

The `download()` function is synchronous and no errors occur, so the data is downloaded after the first line of code completes. But since sorting is asynchronous, `doWork2()` can return before the sort completes.

Feedback?

Promise object

An asynchronous function cannot return the result of the operation since the function may return before the operation completes. So asynchronous functions often return a ***Promise object***: An object representing the eventual completion of the asynchronous operation.

A Promise object can be in one of three states: pending, fulfilled, or rejected.

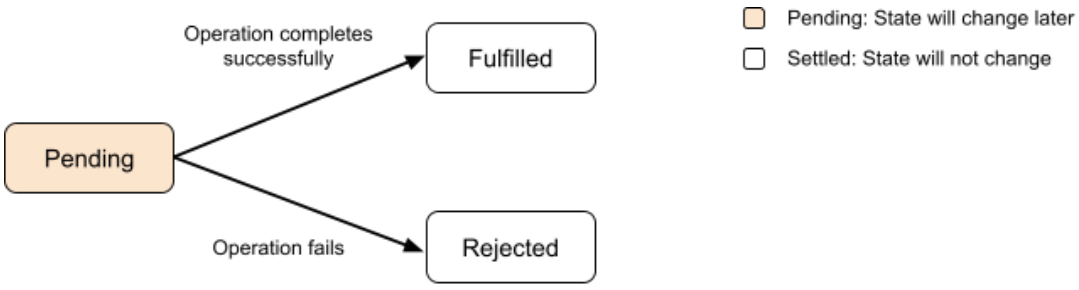
- ***Pending*** means that the asynchronous operation is still running.
- ***Fulfilled*** means that the asynchronous operation has completed successfully.
- ***Rejected*** means that the asynchronous operation has ended in failure to produce the intended result.

Once reaching the fulfilled or rejected state, the Promise object is ***settled***, and the state will not change again.

The Promise constructor has a single parameter, an ***executor function*** that executes in the background. The executor function has two parameters:

- `resolve` - Function to call when the executor function has completed successfully (state becomes fulfilled)
- `reject` - Function to call when the executor function has completed unsuccessfully (state becomes rejected)

Figure 8.13.1: Promise state transition diagram.



Feedback?

PARTICIPATION ACTIVITY 8.13.4: Function that returns a Promise object.

1 2 3 4 5 6 7 2x speed

```
function saveToCloudAsync(userData) {
   return new Promise(function(resolve, reject) {
```

```
        if (!userData) {
           reject();
        }

        // Simulate saving taking 2 seconds
        setTimeout(resolve, 2000);
    });
}
```

```
saveToCloudAsync(null);
```
Promise state: pending rejected

```
saveToCloudAsync({data: 5});
```
Promise state: pending fulfilled

resolve() is called after 2 seconds, which changes the Promise state to fulfilled.

Captions

1. The function saveToCloudAsync() simulates saving user data to a remote server.
2. saveToCloudAsync() returns a Promise object. The Promise constructor's executor function has resolve and reject parameters.
3. When saveToCloudAsync() is called and the Promise object is first created, the Promise is in the pending state.
4. Since userData is null, reject() is called. The Promise moves to the rejected state.
5. When saveToCloudAsync() is called with an object, the new Promise object is in the pending state.
6. Since userData is not null, setTimeout() is called to simulate an executor function that saves userData to a remote server.
7. resolve() is called after 2 seconds, which changes the Promise state to fulfilled.

Feedback?

PARTICIPATION ACTIVITY 8.13.5: Promise object.

1) A Promise object can represent an asynchronous operation that has _____.

- not started
- completed

Correct

A settled Promise object represents a completed asynchronous operation.

2) A Promise may change state if in the _____ state.

- pending
- fulfilled
- rejected

Correct

A Promise in the pending state will eventually change to the fulfilled or rejected state.

3) A settled Promise will not be in the _____ state.

- pending
- fulfilled
- rejected

Correct

A Promise is settled only if the state is fulfilled or rejected, and will not change again.

4) In the animation above, how long does the Promise stay in the pending state when calling `saveToCloudAsync({data: 5})`?

- Less than 1 second
- 2 seconds
- More than 3 seconds

Correct

Since `userData` is not null, the Promise is pending until the `setTimeout()` callback executes in 2 seconds and changes the state to fulfilled. The animation uses `setTimeout()` to simulate a potentially long operation. A function that actually transmits data to a remote server is not likely to use `setTimeout()`.

Feedback?

Promise.then() method

A Promise object's ***then()*** method can be called to request notifications about the state. Two arguments are passed to the `then()` method. The first is a function to be called if the Promise is fulfilled, and the second is a function to be called if the Promise is rejected.

The `then()` method can be called when the Promise object is in any state. The fulfilled callback function will eventually be called if either of the following is true:

- the Promise is already fulfilled when `then()` is called, or
- the Promise is pending when `then()` is called and is eventually fulfilled.

Similarly, the rejected callback function will eventually be called if either of the following is true:

- the Promise is already rejected when `then()` is called, or
- the Promise is pending when `then()` is called and is eventually rejected.

PARTICIPATION ACTIVITY 8.13.6: Calling then() with fulfilled and rejected callbacks.

1 2 3 4 2x speed

```
function saveUserData(data) {
   let promise = saveToCloudAsync(data);
   promise.then(dataSaved, saveFailed);
}

function dataSaved() {
   console.log("Data saved to cloud!");
}
```

Order of function calls:

1. saveUserData (starts)
2. saveToCloudAsync
3. promise.then
4. saveUserData (completes)

```
function saveFailed() {
   console.error("Failed to save data to cloud");
}
```

5. dataSaved
   OR
   saveFailed

If the save attempt fails, saveFailed() is eventually called.

Captions

1. The saveUserData() function calls saveToCloudAsync(), which returns a Promise object.
2. The Promise object's then() method is called with 2 functions provided as arguments. saveUserData() returns while the async save attempt continues in the background.
3. If the save attempt succeeds, the dataSaved() function is eventually called.
4. If the save attempt fails, saveFailed() is eventually called.

Feedback?

PARTICIPATION ACTIVITY 8.13.7: Promise.then() method.

Refer to the animation above.

1) If the Promise object returned from `saveToCloudAsync()` was rejected before calling `then()`, then neither `dataSaved()` nor `saveFailed()` would be called.

   - True
   - (selected) False

Correct

Calling `then()` on an already rejected Promise results in the rejected callback being called. So `saveFailed()` would be called.

2) A partially successful save would result in both `dataSaved()` and `saveFailed()` being called.

   - True
   - (selected) False

Correct

Promise objects are either fulfilled or rejected. No state exists for a partially successful operation. So only one of the two callback functions passed to `then()` will ever be called.

3) Reversing argument order and calling `promise.then(saveFailed,`

Correct

`dataSaved()` would result in `saveFailed()` being called on fulfillment, and `dataSaved()` being called on rejection.

- True
- False

The `then()` method takes 2 functions and calls the first on fulfillment and the second on rejection. The function names have no bearing on behavior, so the programmer must take care to pass the callback functions in the proper order.

4) The `saveUserDataNew()` function below works like `saveUserData()` from the animation, except `saveUserDataNew()` uses anonymous functions.

```
function
saveUserDataNew(data) {
   let promise =
saveToCloudAsync(data);
   promise.then(
      function() {

console.log("Data saved
to cloud!");
      },
      function() {

console.error("Failed
to save data to
cloud");
      }
   );
}
```

- True
- False

Correct

The `then()` arguments can be anonymous functions or arrow functions:

```
function saveUserData(data) {
   let promise =
saveToCloudAsync(data);
   promise.then(
      () => console.log("Data saved
to cloud!"),
      () => console.error("Failed
to save data")
   );
}
```

5) The `promise` variable is not necessary to call `then()`.

```
function saveUserData(data)
{

saveToCloudAsync(data).then(
      function() {
         console.log("Data
saved to cloud!");
      },
      function() {

console.error("Failed to
save data to cloud");
      }
   );
}
```

- (●) True
- ( ) False

Correct

The code can be simplified to avoid assigning the Promise to a `promise` variable. The code can be simplified even further with arrow functions:

```
function saveUserData(data) {

saveToCloudAsync(data).then(
      () => console.log("Data
saved to cloud!"),
      () =>
console.error("Failed to save
data")
   );
}
```

Feedback?

Promise fulfillment values and rejection reasons

A pending Promise is either fulfilled with a value or rejected with a reason. A fulfilled Promise passes the fulfillment value as an argument to the fulfilled callback function. A rejected Promise passes the rejection reason as an argument to the rejected callback function.

A rejection reason is commonly an Error object. The type of a fulfillment value varies based on the type of Promise. Ex: An asynchronous function that downloads data may return a Promise that passes the downloaded data string as the fulfillment value.

PARTICIPATION ACTIVITY 8.13.8: Promise fulfillment values and rejection reasons.

1 2 3 4 5 6 2x speed

```
function loadFromCloudAsync(filename) {
   return new Promise(function(resolve, reject) {
      if (filename === "Hello.txt") {
         // Simulate loading taking 2 seconds
         setTimeout(() => resolve("Hello World!"), 2000);
      }
      else {
         reject(new Error("File does not exist"));
      }
   });
}

function loadString(filename) {
   let promise = loadFromCloudAsync(filename);
   promise.then(dataLoaded, loadFailed);
}

function dataLoaded(value) {
   console.log("Data loaded from cloud: " + value);
}
```

Cloud storage:

Hello.txt file conten

Hello World!

(OtherFile.txt does exist)

```
function loadFailed(reason) {
   console.error(reason.toString());
}
```

```
loadString("Hello.txt");
```

Console (on success):

Data loaded from cloud: Hello World!

```
loadString("OtherFile.txt");
```

Console (on failure):

Error: File does not exist

loadFailed() is called with the Error object as an argument.
loadFailed() logs the error message to the console.

Captions

1. The loadFromCloudAsync() function simulates loading a string from a file in cloud storage. loadFromCloudAsync() returns a Promise object.
2. loadString() calls loadFromCloudAsync() and then() on the returned Promise.
3. Since filename is Hello.txt, loadFromCloudAsync() calls resolve() with a string argument after 2 seconds.
4. dataLoaded() is called with the argument "Hello World!". dataLoaded() logs the string to the console.
5. An attempt to load a non-existent file results in a reject() call with an Error object.
6. loadFailed() is called with the Error object as an argument. loadFailed() logs the error message to the console.

Feedback?

PARTICIPATION ACTIVITY 8.13.9: Promise fulfillment values and rejection reasons.

Refer to the animation above.

1) If the cloud storage operation loaded only part of the file and then got disconnected, the `dataLoaded()` function would likely be called.

   - True
   - False (selected)

Correct

Storage operations that get disconnected before completing will likely consider the operation a failure and thus reject the Promise. `dataLoaded()` is the fulfilled callback, not the rejected callback.

2) If `loadFromCloudAsync()` calls `resolve()` with an array, `dataLoaded()`'s

Correct

The `resolve()` argument is always passed to the Promise's fulfilled callback. Therefore, the data type

`value` parameter would be an array.

- True
- False

of `resolve()`'s argument always matches the data type of `dataLoaded()`'s `value` parameter.

3) If `loadFromCloudAsync()` calls `reject()` with a string, `loadFailed()`'s `reason` parameter is still an Error object.

- True
- False

Correct

The `reject()` argument is always passed to the Promise's rejected callback. Therefore, the data type of `reject()`'s argument always matches the data type of `loadFailed()`'s `reason` parameter.

4) A function that actually loads data from the cloud might result in `loadFailed()` being called, even when the filename exists.

- True
- False

Correct

A function that loads data from the cloud could fail for various reasons other than non-existent files. Ex: If the user doesn't have appropriate permissions to access the file, the load may fail, causing `loadFailed()` to be called.

Feedback?

Promise.catch() method

The second parameter to the `then()` method is optional. If omitted, the parameter defaults to a function that throws an exception.

A Promise object's ***catch()*** method takes a single argument that is a function to call if the Promise is rejected or if the fulfilled handler throws an exception. Consider the two statements:

1. `promiseObj.then(okFunc, failFunc);`
2. `promiseObj.then(okFunc).catch(failFunc);`

While having some similarity, the two statements are not equivalent. The first statement will call either `okFunc()` or `failFunc()`, but not both. The second statement will call `okFunc()` if `promiseObj` is fulfilled, and then also call `failFunc()` if `okFunc()` throws an exception. Both will call only `failFunc()` if `promiseObj` is rejected.

Statements of the form `promiseObj.then(okFunc).catch(failFunc);` are commonly used when working with Promises.

Table 8.13.1: Comparison of Promise object's then() and catch() usage scenarios.

Code	Scenario	Function(s) called	Uncaught exception?
`promise1.then(okFunc, failFunc);`	promise1 fulfilled, okFunc() does NOT throw an exception	okFunc() only	No
	promise1 fulfilled, okFunc() throws an exception	okFunc() only	Yes
	promise1 rejected, failFunc() does NOT throw an exception	failFunc() only	No
	promise1 rejected, failFunc() throws an exception	failFunc() only	Yes
`promise1.then(okFunc).catch(failFunc);`	promise1 fulfilled, okFunc() does NOT throw an exception	okFunc() only	No

	promise1 fulfilled, okFunc() throws an exception	okFunc() first, then failFunc()	No
	promise1 rejected, failFunc() does NOT throw an exception	failFunc() only	No
	promise1 rejected, failFunc() throws an exception	failFunc() only	Yes

Feedback?

PARTICIPATION ACTIVITY 8.13.10: Promise.catch() method.

Suppose `promise2` is a Promise object in the pending state. Consider the following functions:

```
function noThrow(arg) {
   return arg;
}
function yesThrow() {
   throw new Error("Error message");
}
```

If unable to drag and drop, refresh the page.

Results in an uncaught exception if `promise2` is fulfilled.

Correct

`promise2.then(yesThrow, noTh`	When `promise2` is fulfilled, `yesThrow()` is called. Since the Promise's `catch` method is not used, and `yesThrow()` throws an exception, the exception is uncaught.	
`promise2.then(noThrow).catch`	Results in an uncaught exception if `promise2` is rejected. Since `promise2`'s `then` function is called without a second argument, a default, exception-throwing function is used for the missing second argument. The default function is called and throws an exception, which is caught by `catch`, which then calls `yesThrow()`. But `yesThrow()` throws an uncaught exception.	Correct
`promise2.then(yesThrow).catc`	Does not result in an uncaught exception, regardless of whether `promise2` is fulfilled or rejected. If `promise2` is fulfilled, `yesThrow()` is called, and the thrown exception is caught by the `catch` method. If `promise2` is rejected, `noThrow()` is called, and no exceptions are uncaught.	Correct

Reset

Feedback?

Exploring further:

- Promise (MDN)
- Using promises (MDN)

How was this section? Provide section feedback