**What is a Promise?**

Promises are objects that represent the eventual outcome of an asynchronous operation. A Promise object can be in one of three states:

* **Pending**: The initial state— the operation has not completed yet.
* **Fulfilled**: The operation has completed successfully and the promise now has a *resolved value*. For example, a request’s promise might resolve with a JSON object as its value.
* **Rejected**: The operation has failed and the promise has a reason for the failure. This reason is usually an Error of some kind.

We refer to a promise as *settled* if it is no longer pending— it is either fulfilled or rejected. Let’s think of a dishwasher as having the states of a promise:

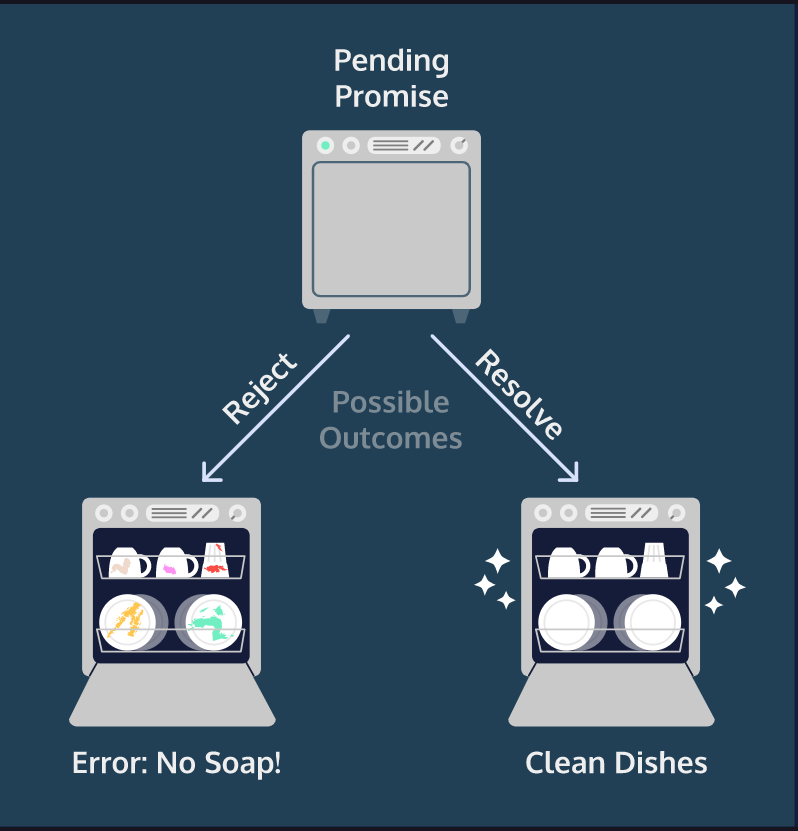
* **Pending**: The dishwasher is running but has not completed the washing cycle.
* **Fulfilled**: The dishwasher has completed the washing cycle and is full of clean dishes.
* **Rejected**: The dishwasher encountered a problem (it didn’t receive soap!) and returns unclean dishes.

If our dishwashing promise is fulfilled, we’ll be able to perform related tasks, such as unloading the clean dishes from the dishwasher. If it’s rejected, we can take alternate steps, such as running it again with soap or washing the dishes by hand.

All promises eventually settle, enabling us to write logic for what to do if the promise fulfills or if it rejects.

**Instructions**

Observe the diagram on the right. Here, we illustrate the different possible states of a dishwashing promise.



**Constructing a Promise Object**

Let’s construct a promise! To create a new Promise object, we use the new keyword and the Promise constructor method:

const executorFunction = (resolve, reject) => { };

const myFirstPromise = new Promise(executorFunction);

The Promise constructor method takes a function parameter called the *executor function* which runs automatically when the constructor is called. The executor function generally starts an asynchronous operation and dictates how the promise should be settled.

The executor function has two function parameters, usually referred to as the resolve() and reject() functions. The resolve() and reject() functions aren’t defined by the programmer. When the Promise constructor runs, JavaScript will pass **its own** resolve() and reject() functions into the executor function.

* resolve is a function with one argument. Under the hood, if invoked, resolve() will change the promise’s status from pending to fulfilled, and the promise’s resolved value will be set to the argument passed into resolve().
* reject is a function that takes a reason or error as an argument. Under the hood, if invoked, reject() will change the promise’s status from pending to rejected, and the promise’s rejection reason will be set to the argument passed into reject().

Let’s look at an example executor function in a Promise constructor:

const executorFunction = (resolve, reject) => {

if (someCondition) {

resolve('I resolved!');

} else {

reject('I rejected!');

}

}

const myFirstPromise = new Promise(executorFunction);

Let’s break down what’s happening above:

* We declare a variable myFirstPromise
* myFirstPromise is constructed using new Promise() which is the Promise constructor method.
* executorFunction() is passed to the constructor and has two functions as parameters: resolve and reject.
* If someCondition evaluates to true, we invoke resolve() with the string 'I resolved!'
* If not, we invoke reject() with the string 'I rejected!'

In our example, myFirstPromise resolves or rejects based on a simple condition, but, in practice, promises settle based on the results of asynchronous operations. For example, a database request may fulfill with the data from a query or reject with an error thrown. In this exercise, we’ll construct promises which resolve synchronously to more easily understand how they work.

**Instructions**

**1.**

You’ll be writing your code in the code-editor, but we won’t be running it until the final step. To check your code for a step, you can press the “Check Work” button.

We’re going to create a promise representing ordering sunglasses from an online store. First, create the function, myExecutor(). Later on, you’ll pass this function into the Promise constructor.

myExecutor() should:

* Have two parameters: resolve and reject
* Check if the sunglasses property on the inventory object has a value greater than zero
* If it does, myExecutor() should invoke resolve() with the string 'Sunglasses order processed.'
* If it does not, myExecutor() should invoke reject() with the string 'That item is sold out.'

When you’re ready, press the “Check Work” button.

Hint

Here’s an example executor function expression:

const anExampleExecutor = (resolve, reject) => {

if (someCondition) {

resolve('I resolved!');

} else {

reject('I rejected!');

}

}

**2.**

Create a function, orderSunglasses(). This function should have no parameters. It should return a new promise constructed by passing your myExecutor() function into the Promise constructor.

Hint

You’ll need to use the new keyword and the Promise constructor method to create a new promise object:

new Promise(anExecutorFunction);

Remember to pass the function in without invoking it and make sure your orderSunglasses() function returns the promise.

**3.**

Create a variable orderPromise assigned to the returned value of your orderSunglasses() function.

Hint

Your code should look similar to this:

const variableName = functionName();

**4.**

At the bottom of your **app.js** file, log orderPromise to the console.

**5.**

In this exercise and throughout the lesson, we’ll provide you with a bash terminal to execute your code. To run the **app.js** program, you’ll type node app.js in the terminal and hit enter (or return). You’ll be able to see the output of the program in the terminal.

Let’s try it! Type node app.js in the terminal and hit enter.

If you’d like, you can see an alternate output by changing the sunglasses property in the inventory object to 0 and executing app.js from the terminal again.

When you’re ready to move on, press the “Check Work” button.

Hint

Hint: If you type just node you’ll open the Node.js REPL:

$ node

>

You can quit out of this by typing in .exit and pressing enter.

const inventory = {

  sunglasses: 1900,

  pants: 1088,

  bags: 1344

};

// Write your code below:

const myExecutor = (resolve, reject) => {

    if (inventory.sunglasses > 0) {

        resolve('Sunglasses order processed.');

    } else {

        reject('That item is sold out.');

    }

};

const orderSunglasses = () => {

    return new Promise(myExecutor);

};

const orderPromise = orderSunglasses();

console.log(orderPromise);

# Consuming Promises

The initial state of an asynchronous promise is pending, but we have a guarantee that it will settle. How do we tell the computer what should happen then? Promise objects come with an aptly named .then() method. It allows us to say, “I have a promise, when it settles, **then** here’s what I want to happen…”

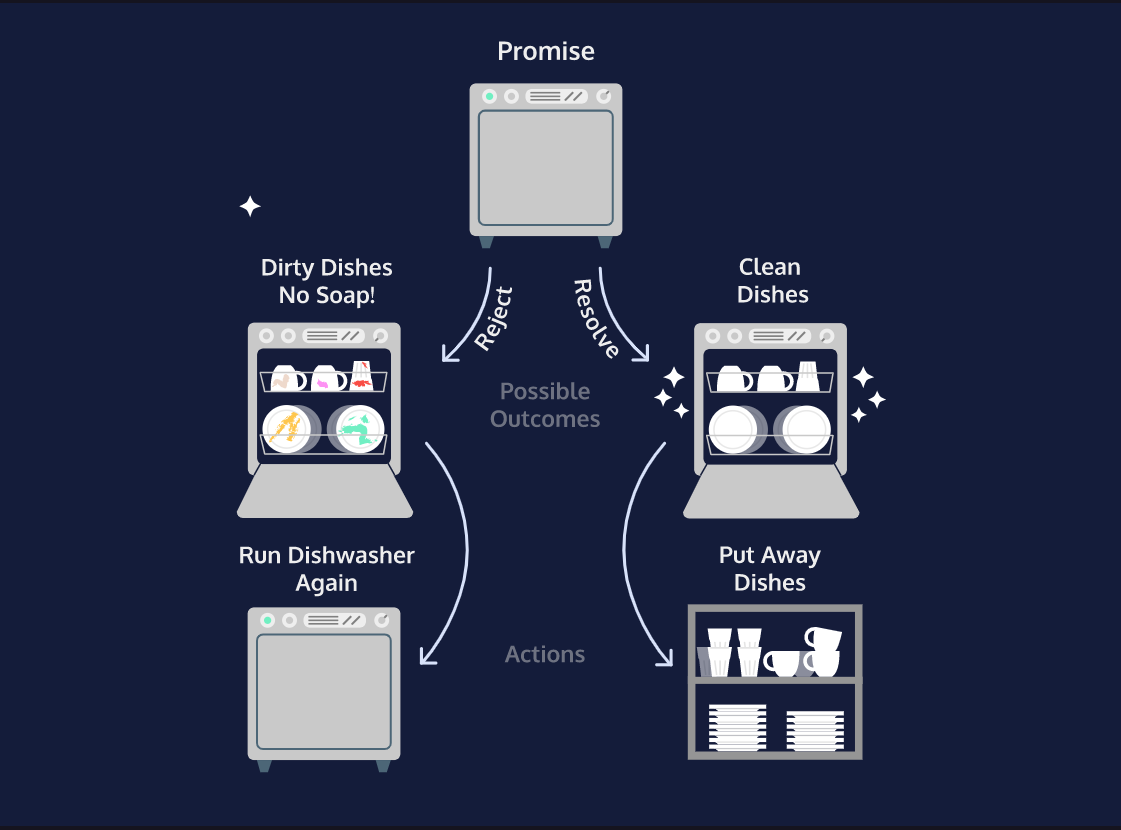
In the case of our dishwasher promise, the dishwasher will run **then**:

* If our promise rejects, this means we have dirty dishes, and we’ll add soap and run the dishwasher again.
* If our promise fulfills, this means we have clean dishes, and we’ll put the dishes away.

.then() is a higher-order function— it takes two callback functions as arguments. We refer to these callbacks as handlers. When the promise settles, the appropriate handler will be invoked with that settled value.

* The first handler, sometimes called onFulfilled, is a success handler, and it should contain the logic for the promise resolving.
* The second handler, sometimes called onRejected, is a failure handler, and it should contain the logic for the promise rejecting.

We can invoke .then() with one, both, or neither handler! This allows for flexibility, but it can also make for tricky debugging. If the appropriate handler is not provided, instead of throwing an error, .then() will just return a promise with the same settled value as the promise it was called on. One important feature of .then() is that it always returns a promise. We’ll return to this in more detail in a later exercise and explore why it’s so important.



# The onFulfilled and onRejected Functions

To handle a “successful” promise, or a promise that resolved, we invoke .then() on the promise, passing in a success handler callback function:

const prom = new Promise((resolve, reject) => {

resolve('Yay!');

});

const handleSuccess = (resolvedValue) => {

console.log(resolvedValue);

};

prom.then(handleSuccess); // Prints: 'Yay!'

Let’s break down what’s happening in the example code:

* prom is a promise which will resolve to 'Yay!'.
* We define a function, handleSuccess(), which prints the argument passed to it.
* We invoke prom‘s .then() function passing in our handleSuccess() function.
* Since prom resolves, handleSuccess() is invoked with prom‘s resolved value, 'Yay', so 'Yay' is logged to the console.

With typical promise consumption, we won’t know whether a promise will resolve or reject, so we’ll need to provide the logic for either case. We can pass both an onFulfilled and onRejected callback to .then().

let prom = new Promise((resolve, reject) => {

let num = Math.random();

if (num < .5 ){

resolve('Yay!');

} else {

reject('Ohhh noooo!');

}

});

const handleSuccess = (resolvedValue) => {

console.log(resolvedValue);

};

const handleFailure = (rejectionReason) => {

console.log(rejectionReason);

};

prom.then(handleSuccess, handleFailure);

Let’s break down what’s happening in the example code:

* prom is a promise which will randomly either resolve with 'Yay!'or reject with 'Ohhh noooo!'.
* We pass two handler functions to .then(). The first will be invoked with 'Yay!' if the promise resolves, and the second will be invoked with 'Ohhh noooo!' if the promise rejects.

Let’s write some onFulfilled and onRejected functions!

**Instructions**

**1.**

Take a look at the provided code. We require in a function, checkInventory(). It builds on the logic of the orderSunglasses() function you wrote in a previous exercise.

* checkInventory() takes in an array representing an order and returns a promise.
* If every item in the order is in stock, that promise resolves with the value "Thank you. Your order was successful."
* Otherwise, the promise rejects with the value "We're sorry. Your order could not be completed because some items are sold out".

We used setTimeout() to ensure that the checkInventory() promise settles asynchronously.

If you’d like, look at the **library.js** file to see how it works. Press “Check Work” when you’re ready to move on.

**2.**

Write a function, handleSuccess(). You’ll use this function later on as your success handler. handleSuccess() should have one parameter, representing a resolved value. Inside the body of handleSuccess(), log the parameter to the console.

Hint

You can use any type of function you like. As a function expression it might look something like this:

const ourExampleFunction = (value) => {

console.log(value);

};

**3.**

Write a function, handleFailure(). You’ll use this function later on as your failure handler. handleFailure() should have one parameter, representing a rejection reason. Inside the body of handleFailure(), log the parameter to the console.

Hint

You can use any type of function you like. As a function expression it might look something like this:

const ourExampleFunction = (value) => {

console.log(value);

};

**4.**

Invoke checkInventory() with order. This will return a promise. Attach a .then() function to this. Pass into .then() the two handlers you wrote as callback functions.

Hint

Remember not to invoke the handlers when you pass them into .then()

returnsPromise()

.then(myOnFulfilled, myOnRejected);

**5.**

Type node app.js in the terminal and hit enter.