**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:

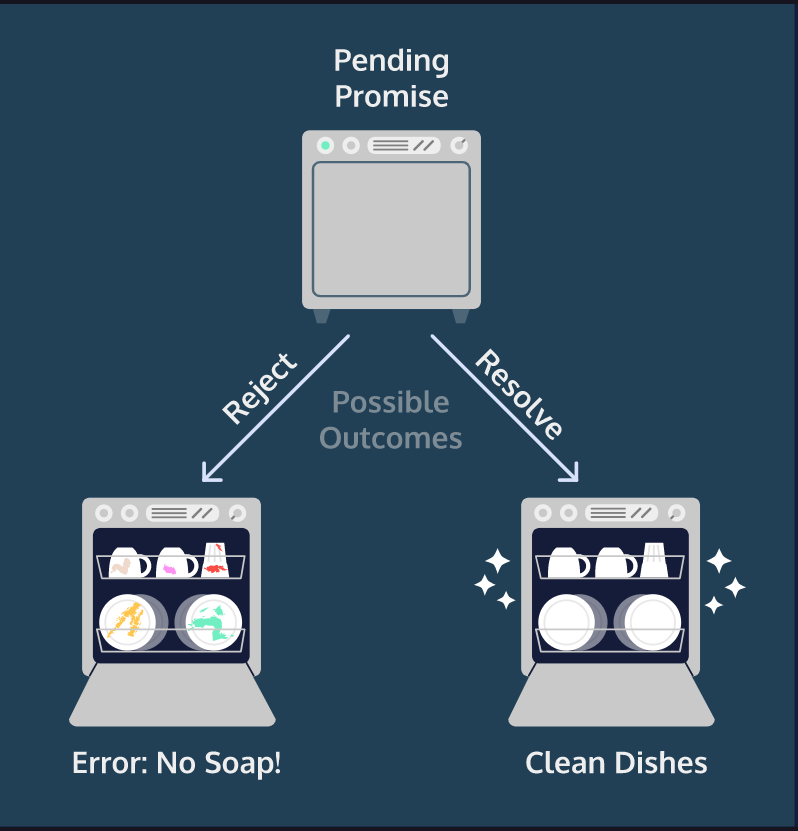
* **Pending**: The dishwasher is running but has not completed the washing cycle.
* **Fulfilled**: The dishwasher has completed the washing cycle and is full of clean dishes.
* **Rejected**: The dishwasher encountered a problem (it didn’t receive soap!) and returns unclean dishes.

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…”

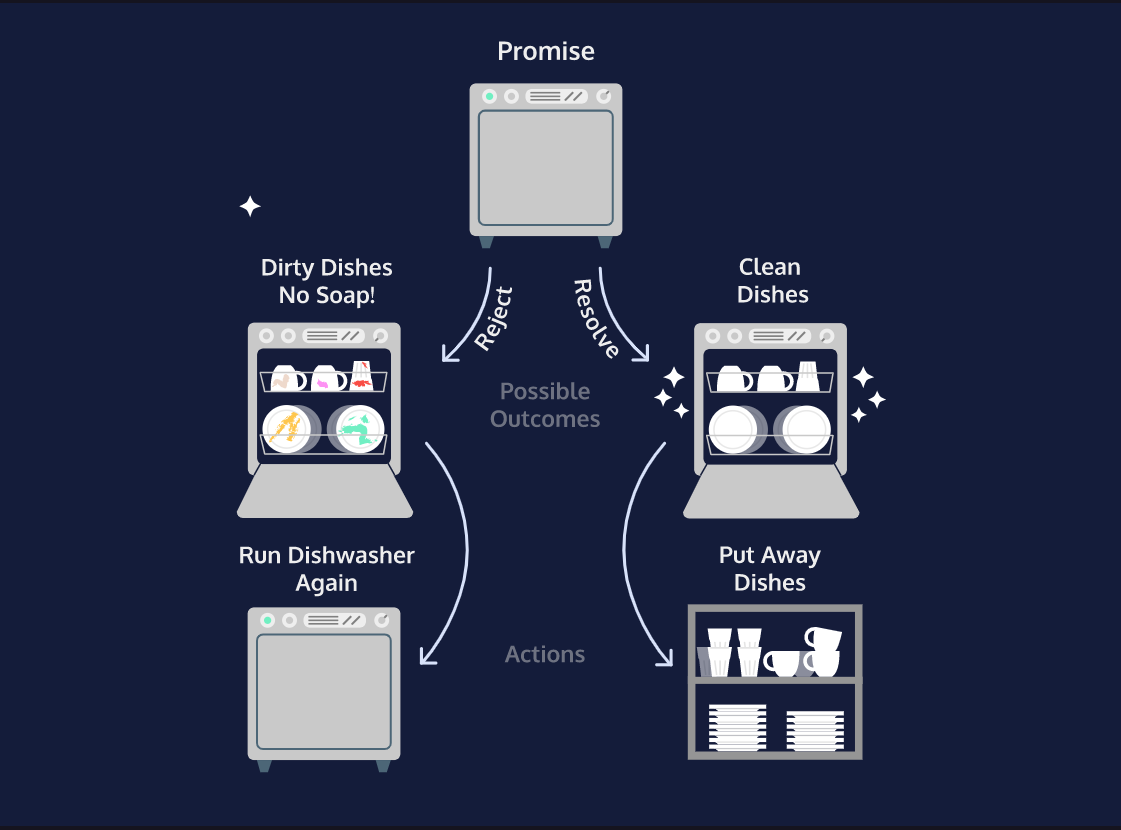
In the case of our dishwasher promise, the dishwasher will run **then**:

* 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.

onst handleSuccess = (resolvedValue) => {

    console.log(resolvedValue);

};

const handleFailure = (rejectReason) => {

    console.log(rejectReason);

};

checkInventory(order)

    .then(handleSuccess, handleFailure);

**Using catch() with Promises**

One way to write cleaner code is to follow a principle called *separation of concerns*. Separation of concerns means organizing code into distinct sections each handling a specific task. It enables us to quickly navigate our code and know where to look if something isn’t working.

Remember, .then() will return a promise with the same settled value as the promise it was called on if no appropriate handler was provided. This implementation allows us to separate our resolved logic from our rejected logic. Instead of passing both handlers into one .then(), we can chain a second .then() with a failure handler to a first .then() with a success handler and both cases will be handled.

prom

.then((resolvedValue) => {

console.log(resolvedValue);

})

.then(null, (rejectionReason) => {

console.log(rejectionReason);

});

Since JavaScript doesn’t mind whitespace, we follow a common convention of putting each part of this chain on a new line to make it easier to read. To create even more readable code, we can use a different promise function: .catch().

The .catch() function takes only one argument, onRejected. In the case of a rejected promise, this failure handler will be invoked with the reason for rejection. Using .catch() accomplishes the same thing as using a .then() with only a failure handler.

Let’s look at an example using .catch():

prom

.then((resolvedValue) => {

console.log(resolvedValue);

})

.catch((rejectionReason) => {

console.log(rejectionReason);

});

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

* prom is a promise which randomly either resolves with 'Yay!' or rejects with 'Ohhh noooo!'.
* We pass a success handler to .then() and a failure handler to .catch().
* If the promise resolves, .then()‘s success handler will be invoked with 'Yay!'.
* If the promise rejects, .then() will return a promise with the same rejection reason as the original promise and .catch()‘s failure handler will be invoked with that rejection reason.

Let’s practice writing .catch() functions.

**Instructions**

**1.**

We’re going to refactor the functionality of the previous exercise but this time we’ll use .catch()! First invoke the checkInventory() function with the order. Remember, this function will return a promise.

**2.**

Add a .then() to the returned promise. Pass in the success handler handleSuccess().

Hint

Remember not to invoke the handleSuccess() when you pass it in as an argument to .then().

**3.**

Add a .catch() to the returned promise. Pass in the failure handler handleFailure().

Hint

In the previous exercise, we passed handleFailure() as a second argument to .then(). In this exercise we’ll chain a catch() function and pass handleFailure() in as an argument to catch(). Remember not to invoke the handleFailure() when you pass it in.

**4.**

We set our inventory of sunglasses to 0, so the order shouldn’t go through. Let’s make sure our code has the expected results. Type node app.js in the terminal and hit enter.

const {checkInventory} = require('./library.js');

const order = [['sunglasses', 0], ['bags', 2]];

const handleSuccess = (resolvedValue) => {

  console.log(resolvedValue);

};

const handleFailure = (rejectReason) => {

  console.log(rejectReason);

};

// Write your code below:

checkInventory(order).then(handleSuccess).catch(handleFailure);

**Chaining Multiple Promises**

One common pattern we’ll see with asynchronous programming is multiple operations which depend on each other to execute or that must be executed in a certain order. We might make one request to a database and use the data returned to us to make another request and so on! Let’s illustrate this with another cleaning example, washing clothes:

We take our dirty clothes and put them in the washing machine. If the clothes are cleaned, **then** we’ll want to put them in the dryer. After the dryer runs, if the clothes are dry, **then** we can fold them and put them away.

This process of chaining promises together is called *composition*. Promises are designed with composition in mind! Here’s a simple promise chain in code:

firstPromiseFunction()

.then((firstResolveVal) => {

return secondPromiseFunction(firstResolveVal);

})

.then((secondResolveVal) => {

console.log(secondResolveVal);

});

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

* We invoke a function firstPromiseFunction() which returns a promise.
* We invoke .then() with an anonymous function as the success handler.
* Inside the success handler we **return** a new promise— the result of invoking a second function, secondPromiseFunction() with the first promise’s resolved value.
* We invoke a second .then() to handle the logic for the second promise settling.
* Inside that .then(), we have a success handler which will log the second promise’s resolved value to the console.

In order for our chain to work properly, we had to return the promise secondPromiseFunction(firstResolveVal). This ensured that the return value of the first .then() was our second promise rather than the default return of a new promise with the same settled value as the initial.

Let’s write some promise chains!

**Instructions**

**1.**

Take a look at the provided code. We require in three functions: checkInventory(), processPayment(), shipOrder(). These functions each return a promise.

checkInventory() expects an order argument and returns a promise. If there are enough items in stock to fill the order, the promise will resolve to an array. The first element in the resolved value array will be the same order and the second element will be the total cost of the order as a number.

processPayment() expects an array argument with the order as the first element and the purchase total as the second. This function returns a promise. If there is a large enough balance on the giftcard associated with the order, it will resolve to an array. The first element in the resolved value array will be the same order and the second element will be a tracking number.

shipOrder() expects an array argument with the order as the first element and a tracking number as the second. It returns a promise which resolves to a string confirming the order has shipped.

If you’d like, look at the **library.js** file to see how these functions work. Press “Check Work” when you’re ready to move on to the next checkpoint.

**2.**

We set up a promise chain but it’s missing a couple important lines of code to make it function properly.

We invoked checkInventory() with order and chained a .then() function to it. This .then() has an anonymous function as its success handler, but it’s missing a return statement.

The success handler should return a processPayment() promise.

Hint

The processPayment() function should be invoked with promise with resolvedValue, the anonymous function’s parameter. This will return the expected promise to the next step in the chain.

If we had used a named function instead, here’s how it might look:

const myFirstSuccessHandler = (resolvedValue) => {

return processPayment(resolvedValue);

};

**3.**

We have a second .then() function on the chain. This .then() also has an anonymous function as its success handler and is missing a return statement.

The success handler should return a shipOrder() promise.

Hint

The shipOrder() function should be invoked with resolvedValue, the anonymous function’s parameter. This will return the expected promise to the next step in the chain.

If we had used a named function instead, here’s how it might look:

const mySecondSuccessHandler = (resolvedValue) => {

return shipOrder(resolvedValue);

};

**4.**

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

const {checkInventory, processPayment, shipOrder} = require('./library.js');

const order = {

  items: [['sunglasses', 1], ['bags', 2]],

  giftcardBalance: 79.82

};

checkInventory(order)

.then((resolvedValueArray) => {

  // Write the correct return statement here:

 return processPayment(resolvedValueArray);

})

.then((resolvedValueArray) => {

  // Write the correct return statement here:

  return shipOrder(resolvedValueArray);

})

.then((successMessage) => {

  console.log(successMessage);

})

.catch((errorMessage) => {

  console.log(errorMessage);

});

**Avoiding Common Mistakes**

Promise composition allows for much more readable code than the nested callback syntax that preceded it. However, it can still be easy to make mistakes. In this exercise, we’ll go over two common mistakes with promise composition.

**Mistake 1:** Nesting promises instead of chaining them.

returnsFirstPromise()

.then((firstResolveVal) => {

return returnsSecondValue(firstResolveVal)

.then((secondResolveVal) => {

console.log(secondResolveVal);

})

})

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

* We invoke returnsFirstPromise() which returns a promise.
* We invoke .then() with a success handler.
* Inside the success handler, we invoke returnsSecondValue() with firstResolveVal which will return a new promise.
* We invoke a second .then() to handle the logic for the second promise settling all **inside** the first then()!
* Inside that second .then(), we have a success handler which will log the second promise’s resolved value to the console.

Instead of having a clean chain of promises, we’ve nested the logic for one inside the logic of the other. Imagine if we were handling five or ten promises!

**Mistake 2:** Forgetting to return a promise.

returnsFirstPromise()

.then((firstResolveVal) => {

returnsSecondValue(firstResolveVal)

})

.then((someVal) => {

console.log(someVal);

})

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

* We invoke returnsFirstPromise() which returns a promise.
* We invoke .then() with a success handler.
* Inside the success handler, we create our second promise, but we forget to return it!
* We invoke a second .then(). It’s supposed to handle the logic for the second promise, but since we didn’t return, this .then() is invoked on a promise with the same settled value as the original promise!

Since forgetting to return our promise won’t throw an error, this can be a really tricky thing to debug!

**Instructions**

**1.**

The code in **app.js** works correctly, but it doesn’t follow best practices.

Type node app.js in the terminal and hit enter, so you can see what the program outputs.

**2.**

Refactor, or rewrite, the code to avoid the two common mistakes: nesting and forgetting to return properly.

Hint

To use best practices, we can refactor this code:

returnsFirstPromise()

.then((firstResolveVal) => {

returnsSecondValue(firstResolveVal);

.then((secondResolveVal) => {

console.log(secondResolveVal);

});

});

Into this code:

returnsFirstPromise()

.then((firstResolveVal) => {

return returnsSecondValue(firstResolveVal);

})

.then((someVal) => {

console.log(someVal);

});

**3.**

Type node app.js in the terminal and hit enter to make sure your program is still working as expected.

const {checkInventory, processPayment, shipOrder} = require('./library.js');

const order = {

  items: [['sunglasses', 1], ['bags', 2]],

  giftcardBalance: 79.82

};

// Refactor the code below:

checkInventory(order)

    .then((resolvedValueArray) => {

        return processPayment(resolvedValueArray);

    })

    .then((resolvedValueArray) => {

        return shipOrder(resolvedValueArray);

    })

    .then((successMessage) => {

        console.log(successMessage);

    });

**Using Promise.all()**

When done correctly, promise composition is a great way to handle situations where asynchronous operations depend on each other or execution order matters. What if we’re dealing with multiple promises, but we don’t care about the order? Let’s think in terms of cleaning again.

For us to consider our house clean, we need our clothes to dry, our trash bins emptied, and the dishwasher to run. We need **all** of these tasks to complete but not in any particular order. Furthermore, since they’re all getting done asynchronously, they should really all be happening at the same time!

To maximize efficiency we should use *concurrency*, multiple asynchronous operations happening together. With promises, we can do this with the function Promise.all().

Promise.all() accepts an array of promises as its argument and returns a single promise. That single promise will settle in one of two ways:

* If every promise in the argument array resolves, the single promise returned from Promise.all() will resolve with an array containing the resolve value from each promise in the argument array.
* If any promise from the argument array rejects, the single promise returned from Promise.all() will immediately reject with the reason that promise rejected. This behavior is sometimes referred to as *failing fast*.

Let’s look at a code example:

let myPromises = Promise.all([returnsPromOne(), returnsPromTwo(), returnsPromThree()]);

myPromises

.then((arrayOfValues) => {

console.log(arrayOfValues);

})

.catch((rejectionReason) => {

console.log(rejectionReason);

});

Let’s break down what’s happening:

* We declare myPromises assigned to invoking Promise.all().
* We invoke Promise.all() with an array of three promises— the returned values from functions.
* We invoke .then() with a success handler which will print the array of resolved values if each promise resolves successfully.
* We invoke .catch() with a failure handler which will print the first rejection message if any promise rejects.

**Instructions**

**1.**

Our business is doing so well that we’re running low on inventory. We want to reach out to some distributors to see if they have the items we need. We only want to make one restocking order, so we’ll only want to place the order if **all** of the items are available.

Take a look at the provided code. We require in one function: checkAvailability().

checkAvailability() expects two string arguments: an item and a distributor. It returns a promise. The function simulates checking that the given distributor has a given item. 80% of the time it will resolve the promise with the item, and 20% of the time it will reject the promise with an error message stating that the item isn’t available.

We also provided two functions which will serve as success and failure handlers.

If you’d like, look at the **library.js** file to see how these functions work. Press “Check Work” when you’re ready to move on to the next checkpoint.

**2.**

Create three variables each assigned to a separate promise:

* checkSunglasses should be assigned the value returned from invoking checkAvailability() with 'sunglasses' as its first argument and 'Favorite Supply Co.' as its second argument.
* checkPants should be assigned the value returned from invoking checkAvailability() with 'pants' as its first argument and 'Favorite Supply Co.' as its second argument.
* checkBags should be assigned the value returned from invoking checkAvailability() with 'bags' as its first argument and 'Favorite Supply Co.' as its second argument.

**3.**

Invoke Promise.all() with an array containing checkSunglasses, checkPants, and checkBags.

Hint

It can sometimes be helpful to break things into additional steps. If you wanted, you could create a variable to hold your array of promises, and then pass that into Promise.all():

const firstPromise = returnsPromOne();

const secondPromise = returnsPromTwo();

const thirdPromise = returnsPromThree();

const promiseArray = [firstPromise, secondPromise, thirdPromise];

Promise.all(promiseArray);

**4.**

Chain a .then() to the promise returned from Promise.all(). You should pass in onFulfill to serve as the success handler.

Hint

Since Promise.all() returns a single promise, we’ll chain this .then() directly onto it similarly to this:

Promise.all([firstPromise, secondPromise, thirdPromise])

.then((resolvedValue) => {console.log(resolvedValue)})

**5.**

Add a .catch() to the chain. You should pass in onReject to serve as the failure handler.

**6.**

Type node app.js in the terminal and hit enter to execute your program.

const {checkAvailability} = require('./library.js');

const onFulfill = (itemsArray) => {

  console.log(`Items checked: ${itemsArray}`);

  console.log(`Every item was available from the distributor. Placing order now.`);

};

const onReject = (rejectionReason) => {

  console.log(rejectionReason);

};

// Write your code below:

const checkSunglasses = checkAvailability('sunglasses', 'Favorite Supply Co.');

const checkPants = checkAvailability('pants', 'Favorite Supply Co.');

const  checkBags = checkAvailability('bags', 'Favorite Supply Co.');

Promise.all([checkSunglasses, checkPants, checkBags])

  .then(onFulfill)

  .catch(onReject);

# Review

Awesome job! Promises are a difficult concept even for experienced developers, so pat yourself on the back. You’ve learned a ton about asynchronous JavaScript and promises. Let’s review:

* Promises are JavaScript objects that represent the eventual result of an asynchronous operation.
* Promises can be in one of three states: pending, resolved, or rejected.
* A promise is settled if it is either resolved or rejected.
* We construct a promise by using the new keyword and passing an executor function to the Promise constructor method.
* setTimeout() is a Node function which delays the execution of a callback function using the event-loop.
* We use .then() with a success handler callback containing the logic for what should happen if a promise resolves.
* We use .catch() with a failure handler callback containing the logic for what should happen if a promise rejects.
* Promise composition enables us to write complex, asynchronous code that’s still readable. We do this by chaining multiple .then()‘s and .catch()‘s.
* To use promise composition correctly, we have to remember to return promises constructed within a .then().
* We should chain multiple promises rather than nesting them.
* To take advantage of concurrency, we can use Promise.all().