Checkpoint 3

Higher-order functions

Previously, you've learned to use loops to deal with data in arrays. For example, you may have looped through an array to find a particular item, or to find all the items within a particular constraint.

These kinds of problems are not unique. JavaScript has built-in array methods that can help you accomplish common data-manipulation tasks.

Outline

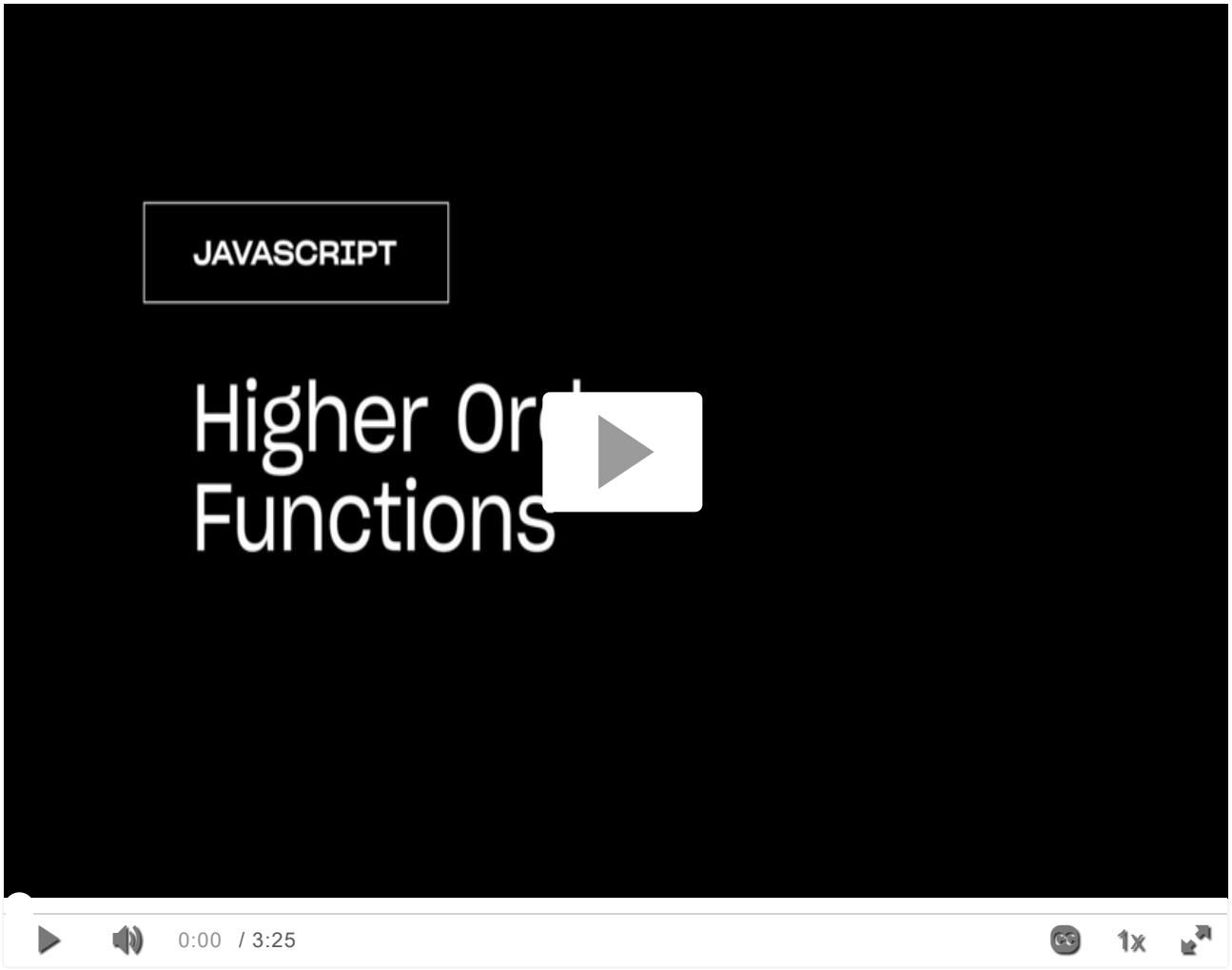
By the end of this checkpoint, you will be able to do the following:

- Define the term *higher-order function*
- Use `forEach()` to loop over items in an array

Higher-order functions

Start by watching the video below, which provides a brief introduction to this topic. Then, read through the rest of the checkpoint and

complete the practice work required. This will give you a full understanding of these concepts.



Outline

The built-in array methods that you will learn in this checkpoint and the next few are all called higher-order functions. A *higher-order function* is any function that either accepts a function as one of its incoming arguments or returns a function. Higher-order functions are incredibly powerful, and they are used a lot in JavaScript. Allowing for a function as an input or an output can lead to some very customizable tools.

For example, suppose that you created a shop and wished to give discounts on some items. You could create a function to calcula the discounted price as follows:

```
const discountedPrice = (price, discount) => price *
```

Then you could call the function to calculate final prices for some items with a 10% discount, like this:

```
const finalPriceBed = discountedPrice(200, 0.9); // $
const finalPricePillow = discountedPrice(52, 0.9); //
const finalPriceCurtain = discountedPrice(32, 0.9); /
```

Did you notice that in each case, you are passing the same value of `0.9`? Instead, you could simply make a new function that calculates the 10% discount, and then you wouldn't need to pass that argument each time. But what if you sometimes wanted to give 25% discounts? You would then need another function that calculates the 25% discount.

Outline

You can use a higher-order function that generates the functions that you need. For example, consider the following function. What does it return?

```
const discountedPrice = (discount) => {
  return (price) => price * discount;
}
```

The function above returns a function that accepts a single parameter `price` and multiplies the price by the `discount` provided. You can use it to first create a function that calculates the 10% discoun d then use the generated function to calculate the discounts.

```
const tenPercentDiscount = discountedPrice(0.9);
const finalPriceBed = tenPercentDiscount(200); // 90
const finalPricePillow = tenPercentDiscount(52); // 9
const finalPriceCurtain = tenPercentDiscount(32); //
```

And if you wanted to, you could later modify this function to calculate a 25% discount, like this:

```
const twentyFivePercentDiscount = discountedPrice(0.7
const finalPriceBed = twentyFivePercentDiscount(200);
const finalPricePillow = twentyFivePercentDiscount(52
const finalPriceCurtain = twentyFivePercentDiscount(3
```

This example is quite trivial, but over time, you will encounter many examples of higher-order functions that behave in a similar manner.

Outline

One final note on this example: the `discountedPrice()` function above was deliberately written with an explicit return statement. Because this function returns a single value, you can use a more concise syntax and drop the return statement. In other words, you could rewrite this function as follows:

```
const discountedPrice = discount => price => price *
```

Despite the simple definition, higher-order functions can be difficult to understand and write. For now, you will learn about existing higher-order functions that are built into JavaScript. Later on, you will write your own.

The `forEach()` method

Take a look at the following array of park information:

```
let parks = [
  { name: "Biscayne", rating: 4.2 },
  { name: "Grand Canyon", rating: 5 },
  { name: "Gateway Arch", rating: 4.5 },
  { name: "Indiana Dunes", rating: 4.1 },
];
```

It's common to write a loop to do something for each element in an array, like this:

```
for (let i = 0; i < parks.length; i++) {
  console.log(parks[i].name);
}
// Biscayne
// Grand Canyon
// Gateway Arch
// Indiana Dunes
```

Outline

Because you are doing the same thing to each element of the array, you can write a function to perform that same task with each element. Then you can call that function repeatedly in the loop.

```
const logPark = (park) => console.log(park.name);


for (let i = 0; i < parks.length; i++) {
  logPark(parks[i]);
}
```

You can think of the `for` loop as applying this function to each element of the array, one at a time. The function gets each item in succession. In this example, the function parameter `park` is called, because that's helpful for remembering what the item is.

The `forEach()` method lets you do this without a `for` loop. This lets you think about the items themselves, instead of counting indexes.

To see the syntax, take a look at the following example, which is equivalent to the loop above.

```
parks.forEach(logPark);
// Biscayne
// Grand Canyon
// Gateway Arch
// Indiana Dunes
```


The `forEach()` method accepts a function as an argument. The function that you pass to the method is referred to as a callback function. In other words, a *callback function* is a function that is passed into another function as an argument.

How `forEach()` works

Where does `park` come from? That is, how does `forEach()` know what to put there? The `forEach()` method is a higher-order function, in that it takes a callback function as its argument.

So you provide `forEach()` with a callback function. Then, internally, `forEach()` executes that callback function once for each ele t of the array.

In some instances, you may use an anonymous function as the callback. Take a look at the following example. This is equivalent to the code above, but it has been rewritten to use an anonymous function.

```
parks.forEach((park) => console.log(park.name));
// Biscayne
// Grand Canyon
// Gateway Arch
// Indiana Dunes
```

In the example above, you aren't passing a named function to `forEach()`; rather, you are defining an anonymous function in the invocation of `forEach()`.

Customizing `forEach()`

Outline

Just like for any other function, you could call the argument anything that you want, and you would get the same results. You can see this in action below:

```
parks.forEach((element) => {
  console.log(element.name);
});
// Biscayne
// Grand Canyon
// Gateway Arch
// Indiana Dunes
```

But naming the argument after what the item means is more helpful to other developers, so the examples in this program will stick to t

The `forEach()` method also gives you access to other arguments that you can use in the function, including the `index` and the original `collection`.

```
parks.forEach((park, index, collection) => {
  console.log(`(${index + 1}/${collection.length}) ${
});
// (1/4) Biscayne
// (2/4) Grand Canyon
// (3/4) Gateway Arch
// (4/4) Indiana Dunes
```

In the above example, you can see that you have access to the individual `element` at each point in the array, the `index` at that point, and then the entire `collection`.

Outline

Checkpoint

This checkpoint will be autograded. Please click the link below to open your assignment in a new tab. Once you complete the assignment, you will see a button allowing you to submit your answers and move on to the next checkpoint.

Your work

03.31.21 **Approved**

Completed Next checkpoint

How would you rate this content?

Report a typo or other issue

Go to Overview

Outline