Filtering Arrays

(https://github.com/learn-co-curriculum/phase-1-filtering-arrays) (https://github.com/learn-co-curriculum/phase-1-filtering-arrays/issues/new)

Learning Goals

- Explain the concept of filtering an array
- Build our own version of JavaScript's `Array.prototype.filter()` method
- Define what makes a function *pure* and explain why *pure functions* are often preferable to *impure functions*
- Use `Array.prototype.filter()`

Introduction

We've seen the `Array` methods available in JavaScript to find a *single* element, but sometimes we want to return *all* elements that match a certain condition. For example, we might want to search through an array and return values greater than one (`[1, 2, 3]` -> `[2, 3]`). In the JavaScript world, we refer to that search process as *filtering* an array. In this lesson we're going to build our own `filter()` function.

Filter

Let's revisit our array of Flatbook user objects:

```
const users = [
  {
    firstName: "Niky",
    lastName: "Morgan",
    favoriteColor: "Blue",
    favoriteAnimal: "Jaguar",
  },
  {
```

```
    firstName: "Tracy",
    lastName: "Lum",
    favoriteColor: "Yellow",
    favoriteAnimal: "Penguin",
  },
  {
    firstName: "Josh",
    lastName: "Rowley",
    favoriteColor: "Blue",
    favoriteAnimal: "Penguin",
  },
  {
    firstName: "Kate",
    lastName: "Travers",
    favoriteColor: "Red",
    favoriteAnimal: "Jaguar",
  },
  {
    firstName: "Avidor",
    lastName: "Turkewitz",
    favoriteColor: "Blue",
    favoriteAnimal: "Penguin",
  },
  {
    firstName: "Drew",
    lastName: "Price",
    favoriteColor: "Yellow",
    favoriteAnimal: "Elephant",
  },
];
```

To review, we know we can iterate over that collection and print out everyone's first name:

```
function firstNamePrinter(collection) {
  for (const user of collection) {
    console.log(user.firstName);
  }
}

firstNamePrinter(users);
// LOG: Niky
// LOG: Tracy
// LOG: Josh
// LOG: Kate
// LOG: Avidor
// LOG: Drew
```

We also know how to print out only users whose favorite color is blue:

```
function blueFilter(collection) {
  for (const user of collection) {
    if (user.favoriteColor === "Blue") {
      console.log(user.firstName);
    }
  }
}

blueFilter(users);
// LOG: Niky
// LOG: Josh
// LOG: Avidor
```

Now what if we want to filter our collection of users for those whose favorite color is red? We could define an entirely new function, `redFilter()` , but that seems wasteful. Instead, let's just pass in the color that we want to filter for as an argument:

```
function colorFilter(collection, color) {
  for (const user of collection) {
    if (user.favoriteColor === color) {
      console.log(user.firstName);
    }
  }
}

colorFilter(users, "Red");
// LOG: Kate
```

Nice! We've extracted some of the hard-coded logic out of the function, making it more generic and reusable. However, now we want to filter our users based on whose favorite animal is a jaguar, and our `colorFilter()` function won't work. Let's abstract the function a bit further:

```
function filter(collection, attribute, value) {
  for (const user of collection) {
    if (user[attribute] === value) {
      console.log(user.firstName);
    }
  }
}

filter(users, "favoriteAnimal", "Jaguar");
// LOG: Niky
// LOG: Kate
```

So our function is definitely getting more abstract, but what if we wanted to filter by two attributes? We'd have to do something like this:

```
function filter(collection, attribute1, value1, attribute2, value2) {
  for (const user of collection) {
    if (user[attribute1] === value1 && user[attribute2] === value2) {
      console.log(user.firstName);
```

```
      }
    }
  }

  filter(users, "favoriteAnimal", "Jaguar", "favoriteColor", "Blue");
  // LOG: Niky
```

This is getting slightly ridiculous by this point. That is **way** too much logic to be putting on the shoulders of our poor little filter function. Plus, now our filter will only work if we're filtering by two attributes. To fix this, we can extract the comparison logic into a separate function:

```
function filter(collection) {
  for (const user of collection) {
    if (likesElephants(user)) {
      console.log(user.firstName);
    }
  }
}

function likesElephants(user) {
  return user["favoriteAnimal"] === "Elephant";
}

filter(users);
// LOG: Drew
```

That separation of concerns feels nice. `filter()` doesn't remotely care what happens inside `likesElephants()`; it simply delegates the comparison and then trusts that `likesElephants()` correctly returns `true` or `false`. We're almost at the finish line, but there's one final abstraction we can make: right now, our `filter()` function can only make comparisons using `likesElephants()`. If we want to use a different comparison function, we'd have to rewrite `filter()`. However, there is another way: we can use a callback function!

Let's refactor our filter function to take a callback:

```
const users = [
  {
    firstName: "Niky",
    lastName: "Morgan",
    favoriteColor: "Blue",
    favoriteAnimal: "Jaguar",
  },
  {
    firstName: "Tracy",
    lastName: "Lum",
    favoriteColor: "Yellow",
    favoriteAnimal: "Penguin",
  },
  {
    firstName: "Josh",
    lastName: "Rowley",
    favoriteColor: "Blue",
    favoriteAnimal: "Penguin",
  },
  {
    firstName: "Kate",
    lastName: "Travers",
    favoriteColor: "Red",
    favoriteAnimal: "Jaguar",
  },
  {
    firstName: "Avidor",
    lastName: "Turkewitz",
    favoriteColor: "Blue",
    favoriteAnimal: "Penguin",
  },
  {
```

```
    firstName: "Drew",
    lastName: "Price",
    favoriteColor: "Yellow",
    favoriteAnimal: "Elephant",
  },
];

function filter(collection, cb) {
  for (const user of collection) {
    if (cb(user)) {
      console.log(user.firstName);
    }
  }
}

filter(users, function (user) {
  return user.favoriteColor === "Blue" && user.favoriteAnimal === "Penguin";
});
// LOG: Josh
// LOG: Avidor

filter(users, function (user) {
  return user.favoriteColor === "Yellow";
});
// LOG: Tracy
// LOG: Drew
```

Our `filter()` function doesn't know or care about any of the comparison logic encapsulated in the callback function. All it does is take in a collection and a callback and `console.log()` out the `firstName` of every `user` object that makes the callback return `true`. And because we've extracted the logic into a separate function, our `filter` now works regardless of how many conditions we want to filter on.

Pure functions

One final note about `filter()` and manipulating objects in JavaScript. We touched on this in the discussions of *destructive* and *nondestructive* operations, but there's some function-specific terminology that's important to know. A function in JavaScript can be *pure* or *impure*.

If a *pure function* is repeatedly invoked with the same set of arguments, the function will **always return the same result**. Its behavior is predictable. Additionally, invoking the function has no external side-effects such as making a network or database call or altering any object(s) passed to it as an argument.

Impure functions are the opposite: the return value is not predictable, and invoking the function might make network or database calls or alter any objects passed in as arguments.

This function is impure because the return value is not predictable:

```
function randomMultiplyAndFloor() {
  return Math.floor(Math.random() * 100);
}

randomMultiplyAndFloor();
// => 53
randomMultiplyAndFloor();
// => 66
```

This one's impure because it alters the passed-in object:

```
const ada = {
  name: "Ada Lovelace",
  age: 202,
};

function happyBirthday(person) {
  console.log(
    `Happy birthday, ${person.name}! You're ${++person.age} years old!`
  );
}
```

```
happyBirthday(ada);
// LOG: Happy birthday, Ada Lovelace! You're 203 years old!

happyBirthday(ada);
// LOG: Happy birthday, Ada Lovelace! You're 204 years old!

ada;
// => {name: "Ada Lovelace", age: 204}
```

When possible, it's generally good to avoid impure functions for the following two reasons:

1. Predictable code is good. If you can be sure that a function will always return the same value when provided the same inputs, it makes writing tests for that function a cinch.
2. Because pure functions don't have side effects, it makes debugging a lot easier. Imagine that our code errors out due to an array that doesn't contain the correct properties.
   - If that array was returned from a pure function, our debugging process would be linear and well-scoped. We would first check what inputs were provided to the pure function. If the inputs are correct, that means the bug is inside our pure function. If the inputs aren't correct, then we figure out why they aren't correct. Case closed!
   - If, however, the array is modified by impure functions, we'd have to follow the data around on a wild goose chase, combing through each impure function to see where and how the array is modified.

> **Top Tip**: The fewer places a particular object can be modified, the fewer places we have to look when debugging.

Here's a pure take on our `randomMultiplyAndFloor()` function:

```
function multiplyAndFloor(num) {
  return Math.floor(num * 100);
}

const randNum = Math.random();
```

```
randNum;
// => 0.9123939589869237

multiplyAndFloor(randNum);
// => 91
multiplyAndFloor(randNum);
// => 91
```

And one that returns a new object instead of mutating the passed-in object:

```
const adaAge202 = {
  name: "Ada Lovelace",
  age: 202,
};

function happyBirthday(person) {
  const newPerson = Object.assign({}, person, { age: person.age + 1 });

  console.log(
    `Happy birthday, ${newPerson.name}! You're ${newPerson.age} years old!`
  );

  return newPerson;
}

const adaAge203 = happyBirthday(adaAge202);
// LOG: Happy birthday, Ada Lovelace! You're 203 years old!

adaAge202;
// => {name: "Ada Lovelace", age: 202}
```

```
adaAge203;
// => {name: "Ada Lovelace", age: 203}
```

Tying it all together

As a final challenge, let's rewrite our `filter()` function as a pure function that returns a new array containing the filtered elements:

```
const users = [
  {
    firstName: "Niky",
    lastName: "Morgan",
    favoriteColor: "Blue",
    favoriteAnimal: "Jaguar",
  },
  {
    firstName: "Tracy",
    lastName: "Lum",
    favoriteColor: "Yellow",
    favoriteAnimal: "Penguin",
  },
  {
    firstName: "Josh",
    lastName: "Rowley",
    favoriteColor: "Blue",
    favoriteAnimal: "Penguin",
  },
  {
    firstName: "Kate",
    lastName: "Travers",
    favoriteColor: "Red",
    favoriteAnimal: "Jaguar",
  },
```

```
    {
      firstName: "Avidor",
      lastName: "Turkewitz",
      favoriteColor: "Blue",
      favoriteAnimal: "Penguin",
    },
    {
      firstName: "Drew",
      lastName: "Price",
      favoriteColor: "Yellow",
      favoriteAnimal: "Elephant",
    },
  ];

  function filter(collection, cb) {
    const newCollection = [];

    for (const user of collection) {
      if (cb(user)) {
        newCollection.push(user);
      }
    }

    return newCollection;
  }

  const bluePenguinUsers = filter(users, function (user) {
    return user.favoriteColor === "Blue" && user.favoriteAnimal === "Penguin";
  });

  bluePenguinUsers;
  // => [{ firstName: "Josh", lastName: "Rowley", favoriteColor: "Blue", favoriteAnimal: "Penguin" }, { firstName: "Avic
```

```
const yellowUsers = filter(users, function (user) {
  return user.favoriteColor === "Yellow";
});

yellowUsers;
// => [{ firstName: "Tracy", lastName: "Lum", favoriteColor: "Yellow", favoriteAnimal: "Penguin" }, { firstName: "Drew

users.length;
```

Woohoo! We successfully built a clone of JavaScript's built-in `.filter()` array method!

Using `Array.prototype.filter()`

Now that we've built our own version of `filter()`, we have a better understanding of what JavaScript's built-in `filter()` method is doing for us and how it works under the hood. Here's an example of what a call to `filter()` might look like:

```
[1, 2, 3, 4, 5].filter(function (num) {
  return num > 3;
});
// => [4, 5]
```

The method accepts one argument, a callback function that it will invoke with each element in the array. For each element passed to the callback, if the callback's return value is `true`, that element is copied into a new array. If the callback's return value is `false`, the element is filtered out. After iterating over every element in the collection, `.filter()` returns the new array.

Conclusion

As we've learned in this lesson, using JavaScript's built-in `filter()` method enables us to write more efficient, less repetitive code. Specifically:

- We no longer need to create a `for` or `for ... of` loop.

- In each iteration through the array, the current element is stored in a variable for us. We no longer need to access elements using their index values.
- A new array is automatically created and returned after the iterations are complete, so we no longer need to create an empty array and push elements into it.

Finally, `Array` methods like `find()`, `filter()` and the other methods we will learn about in this section are *expressive*. As soon as we (or other developers) see that `filter()` is being called, we know that the code is looking for elements in an array that meet a certain condition and returning a new array containing those elements. Or if we see that `map()` (which we'll learn about next) is being called, we immediately know that the code is modifying the elements in an array and returning an array containing the modified values. This makes our code easier to read and understand than if we use a generic looping construct.

Resources

- **MDN — `Array.prototype.filter()` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter)**
- **Tutorial Horizon — Pure vs. Impure Functions (https://tutorialhorizon.com/javascript/pure-vs-impure-functions/)**