

## Tip 23: Pull Out Subsets of Data with `filter()` and `find()`

In this tip, you'll learn how to change the size of an array while retaining the shape of the items.

### We'll cover the following

- `filter()`
  - Example: Filtering names in an array
  - Example: Filtering passing scores in an array
- `find()`
  - Example: Library scheduling app
    - Using a for loop
    - Using `find()`
    - Using currying

### `filter()` #

In the [previous](#) tip, you created a new array by pulling out only the relevant information from the original array. You'll likely encounter situations where you want to keep the shape of the data, but you only want a subset of the total items. Maybe you only want users that live in a certain city, but you still need all their information. The array method `filter()` will perform this exact action. Unlike the `map()` method, you aren't changing any information in the array—you're just reducing what you get back.

### Example: Filtering names in an array #

As an example, let's filter a simple array of strings. You have a team of people, and you want only people named some form of **Dave** (*David, Davis, Davina, and so on*). In my hometown, there's a sandwich shop that gives out a free sandwich once a year to anyone named Joe/Joseph/Joanna, so being able to filter people by name variant is a crucial task. You wouldn't want to deprive your Daves or Joes of a delicious lunch.

Start with a list of coworkers that you want to reduce down.



```
const team = [  
  'Michelle B',  
  'Dave L',  
  'Dave C',  
  'Courtney B',  
  'Davina M',  
];
```

You'll need to check to see if the string contains a form of "Dav" using the `match()` method on a string. This method will return *an array of information* if the string matches a regular expression matches and `null` if there's no match. In other words, `match()` will return a *truthy value*, an *array*, if there's a regex match and a *falsy value*, `null`, if there is none.

```
console.log('Dave'.match(/Dav/));  
console.log('Michelle'.match(/Dav/));
```



Traditionally, you'd solve the problem with a `for` loop. And as you've probably guessed by now, the solution isn't pretty.

```
const team = [  
  'Michelle B',  
  'Dave L',  
  'Dave C',  
  'Courtney B',  
  'Davina M',  
];  
  
const daves = [];  
  
for (let i = 0; i < team.length; i++) {  
  if (team[i].match(/Dav/)) {  
    daves.push(team[i]);  
  }  
}  
console.log(daves);
```



A `filter` function can do the exact same thing in a single line. Like the `map()` method, you call the method on an array and you get an array back.

There's one trick. Unlike the `map()` method, the function you pass into the

`filter()` method must return a truthy value. When you iterate over each item, if it

returns something truthy, it's retained. If it doesn't return a truthy value, it isn't retained. See why it's important to have a solid grasp of truthiness (the programmer kind, not the Colbert kind).

## Example: Filtering passing scores in an array #

Say you want to get the passing scores from an array. The filter function would take each score and say whether it was above the threshold (`60`) and keeps it if it is.

```
const scores = [30, 82, 70, 45];
function getNumberOfPassingScores(scores) {
  const passing = scores.filter(score => score > 59);
  console.log(passing);
  return passing.length;
}

console.log(getNumberOfPassingScores(scores));
```



The function returns either `true` or `false`, but the final array contains the actual values of `82` and `70`. The function checked each score one at a time, retaining the `score` (not the return value) if the return value was true. Note also, the return array preserves the order of the original.

Most important, `filter()` will always return an array, even if nothing matches the values. If you wanted to see how many perfect scores you'd get, you may be a little disappointed. But you can still confidently call the length property knowing you'll have an array of some sort. Simple and predictable.

```
const scores = [30, 82, 70, 45];

function getPerfectScores(scores) {
  const perfect = scores.filter(score => score === 100);
  console.log(perfect);
  return perfect.length;
}

console.log(getPerfectScores(scores));
```



To return to your hungry Dancer: In the previous anonymous functions you pass to

to return to your hungry Daves: In the previous anonymous functions you pass to `filter()`, you're returning a Boolean—`true` or `false`—while in this one, you want to check a string. Because `match()` returns truthy and falsy values, you can use it directly in the `filter` function.

Here's your simplified loop:

```
const team = [
  'Michelle B',
  'Dave L',
  'Dave C',
  'Courtney B',
  'Davina M',
];

const daves = team.filter(member => member.match(/Dav/));
console.log(daves);
```

## `find()` #

Filter is so easy to use that there's not much left to say. Still, there's one variation that can be very useful.

On occasion, you might be lucky enough to know that there will be at most one match (or you're only interested in one match) in your array. In that case, you can use a method that's similar to `filter()` called `find()`. The `find()` method takes a *function as argument*, a function that returns a *truthy or falsy* value, and returns only the *first* result that evaluates to *true*. If there's no true value, it returns `undefined`.

This is great when you know there will only be one value—looking for an entry with a specific ID, for example. Or if you want the first instance of a particular item—getting the last update to a page by a particular user on a sorted array.

Here's a good way to think about this: If you'd normally use a `break` statement in a loop, the action is a good candidate for `find()`.

## Example: Library scheduling app #

Let's say you're writing a scheduling app for library instructors. Each instructor works in several locations, but no location has more than one instructor.

Your array of instructors would look like this:

```
const instructors = [
  {
    name: 'Jim',
    libraries: ['MERIT'],
  },
  {
    name: 'Sarah',
    libraries: ['Memorial', 'SLIS'],
  },
  {
    name: 'Eliot',
    libraries: ['College Library'],
  },
];
```

## Using a **for** loop #

If you were to write a **for** loop to check it, you'd go through each one and **break** when you get to the correct result.

```
const instructors = [
  {
    name: 'Jim',
    libraries: ['MERIT'],
  },
  {
    name: 'Sarah',
    libraries: ['Memorial', 'SLIS'],
  },
  {
    name: 'Eliot',
    libraries: ['College Library'],
  },
];

let memorialInstructor;

for (let i = 0; i < instructors.length; i++) {
  if (instructors[i].libraries.includes('Memorial')) {
    memorialInstructor = instructors[i];
    break;
  }
}

console.log(memorialInstructor);
```



This loop will check the first instructor and see that he doesn't meet the criteria. The second instructor does meet the criteria, saving the incredible labor of looking at the third instructor. Of course, in real-world data, there may be hundreds or

even thousands of results. Stopping at the first instance is a nice little optimization to avoid iterating over the whole set.

## Using `find()` #

How does this translate into a `find()` function? It's simple: The `if` block contains everything you need to change this into a `find()` function. Using the ideas from `filter()`, try to write it out.

You probably came up with something like this:

```
const instructors = [
  {
    name: 'Jim',
    libraries: ['MERIT'],
  },
  {
    name: 'Sarah',
    libraries: ['Memorial', 'SLIS'],
  },
  {
    name: 'Eliot',
    libraries: ['College Library'],
  },
];

const librarian = instructors.find(instructor => {
  return instructor.libraries.includes('Memorial');
});

console.log(librarian);
```

Once again, you've reduced several lines down to a simple expression (*could be a one-liner, but it runs off the printed page!*) while simultaneously removing an unstable `let` with a predictable `const`. The only down-side to using `find()` is that you can't be absolutely sure of the return value. If there's no match, you get `undefined`, while with `filter()` you'd get an empty array if there were no matches. But using your knowledge of short circuiting, you can always add an `||` statement combined with a default.

```
const images = [
  {
    path: './me.jpg',
    profile: false
```

```

    }
  ];

  const profile = images.find(image => image.profile) || {
    path:
      './default.jpg'
  };

  console.log(profile);

```



## Using currying #

There may be one thing bothering you about that `find()` function: *You had to hard code the name of the library, Memorial*. The challenge with an array function is that it takes a single argument, the item being checked. This is a problem if you want to add a second parameter, a variable to check the item against.

What do you do if you want to check against another location? Fortunately, you don't need to write a function for every library. Rather, you'd use a technique called **currying** to reduce the number of arguments down to one. You'll see this a lot more in [Tip 34](#), Maintain Single Responsibility Parameters with Partially Applied Functions, but it's one of my favorite techniques, so I'll go ahead and give you a taste.

```

const instructors = [
  {
    name: 'Jim',
    libraries: ['MERIT'],
  },
  {
    name: 'Sarah',
    libraries: ['Memorial', 'SLIS'],
  },
  {
    name: 'Eliot',
    libraries: ['College Library'],
  },
];

const findByLibrary = library => instructor => {
  return instructor.libraries.includes(library);
};

const librarian = instructors.find(findByLibrary('MERIT'));
console.log(librarian);

```



But don't get too far ahead. There are more array methods to explore.

---

In the next tip, you'll break the pattern of returning a new array by using `forEach()` to perform an action on each array without getting any return values.