Lesson **Thursday**

Introduction to Programming (/introduction-to-programming) / Arrays and Looping (/introduction-to-programming/arrays-and-looping) / Optional Review: Which Loop Should I Use?

Text

Note: You can use any kind of loop for this section's independent project provided that you correctly solve the problem. We recommend using `Array.prototype.forEach()` or `for` loops for the independent project.

Over the course of this section, we've covered many different looping techniques. It may feel overwhelming to decide which loop is best for a specific use case. This lesson will hopefully narrow things down and make life a little easier.

Also, it's important to note that at this point in your learning, you shouldn't worry too much about which loop is "best" for the job at hand. You should feel free to practice them all — and also to experiment with code and solutions that interest you. For the upcoming independent project, you will be required to loop — but how you implement a loop is up to you.

Which Loop Should I Use?

All of these examples assume that you are starting with the following array:

```
> const array = [0,1,2,3,4,5];
```

Yep, we are returning to the doubled array example — but this time all of our looping examples are in one place!

`for`

Example

```
> let doubledArray = [];
> for (let index = 0; index < array.length; index +=1) {
  doubledArray.push(array[index] * 2);
}
```

When to Use

This is a great "starter" loop. You should practice it frequently — at least until you get the hang of it. At that point, you should favor other kinds of loops instead! What's nice about this loop is that you can break out of it, unlike with `Array.prototype.forEach()` or `Array.prototype.map()`.

`Array.prototype.forEach()`

Example

```
> let doubledArray = [];
> array.forEach(function(element) {
  doubledArray.push(element * 2);
});
```

When to Use

Use this when both conditions are met:

1. You want to loop through *every* element in an array without breaking out of the loop;
2. You don't want a *transformed* array.

`for...of`

Example

```
> let doubledArray = [];
> for (const element of array) {
  doubledArray.push(element * 2);
}
```

When to Use

Use this whenever you want to loop through an array, string, or object, but you *don't* want to transform the elements into a new string or array. It comes down to preference, but generally you can favor this one over `Array.prototype.forEach()`. We can break out of this loop, too, which makes it more versatile than `Array.prototype.forEach()`.

`Array.prototype.map()`

Example

```
> const doubledArray = array.map(function(element) {
  return element * 2;
});
```

When to Use

Use this whenever you want to iterate through every element in an array and create a new array with all of its elements *transformed*.

`while` and `do...while`

Example — `while`

```
> let index = 0;
> let doubledArray = [];
> while
 (index < array.length) {
  doubledArray.push(array[index] * 2);
  index ++;
}
```

Example — `do...while`

```
> let index = 0;
> let doubledArray = [];
> do {
  doubledArray.push(array[index] * 2);
  index ++;
} while (index < array.length)
```

When to Use

You won't use this one often — practice it until you have the hang of it and then use it sparingly. It can be useful when you want to loop only until a specific condition is met — or when you are writing code that interacts with users.

What About a Use Case That Doesn't Fit Perfectly?

Let's look at a use case that doesn't fit perfectly — and then solve it with two different kinds of loops. Let's say we want to use a loop to create a *transformed* string. Should we solve the problem with `Array.prototype.map()` if we need to transform the string into an array first? Or should we solve it with `for...of` even though we should favor `Array.prototype.map()` for transformations?

Well, neither approach is wrong. Ultimately, we can try both solutions and see which feels more elegant and concise. Let's return to our `for...of` vowelized loop example:

```
> const consonantString = "bdfmxtgl";
> let vowelizedString = "";
> for (const letter of consonantString) {
  vowelizedString = vowelizedString.concat(letter + "a");
}
> vowelizedString;
"badafamaxatagala"
```

Here's how we'd solve the same problem with `Array.prototype.map()`:

```
> const consonantString = "bdfmxtgl";
> const consonantArray = consonantString.split("");
> const vowelizedArray = consonantArray.map(function(element) {
  return element + "a";
});
> vowelizedArray.join("");
"badafamaxatagala"
```

Both of these solutions come up with the same answer in the same number of lines. Ultimately, the approach we take here comes down to preference. There are valid reasons for both approaches.

You can use just about any kind of loop to solve many problems — and while there are best practices and ways to improve our code, there's no one right approach.

Previous (/introduction-to-programming/arrays-and-looping/further-exploration-while-loops)

Next (/introduction-to-programming/arrays-and-looping/linkedin)

Lesson 48 of 50

Last updated February 28, 2023

disable dark mode

Epicodus (http://www.epicodus.com)