Value vs. Reference with ES2015

We'll learn how 'const' works with the concept of value vs. reference in JS.

Value vs. Reference with `const`

There's a small piece of the discussion of `const` that was left out in the related previous lesson. We haven't discussed how it works with objects. To truly understand that, we first needed an understanding of value vs. reference in JS. With that understanding, we can now finish our discussion of `const`.

Objects are mutable

When using `const`, with primitives, once a value is assigned to a constant variable, that variable is immutable. It cannot be changed.

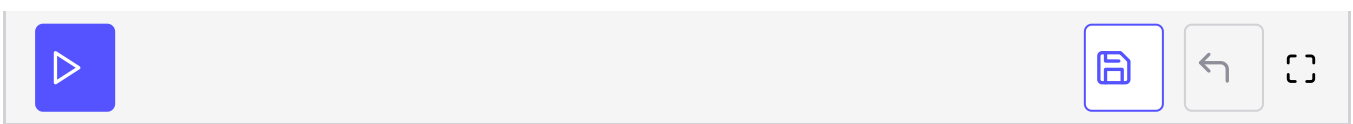
With reference types, namely arrays and objects, it is the reference that cannot be changed. Writing `const arr = [];` means that `arr` always has a reference to the array that was just created.

The array itself is not immutable and can be manipulated. Values can be inserted, removed, sorted, etc. Anything can be done to it. We just can't reassign `arr` to a different reference or value.

```
var varArr = [];
let letArr = [];
const constArr = [];

varArr.push(3);
letArr.push(4);
constArr.push(5);

varArr = ['New var array'];
letArr = ['New let array'];
constArr = ['New const array']; // -> Uncaught TypeError: Assignment to constant variable.
```

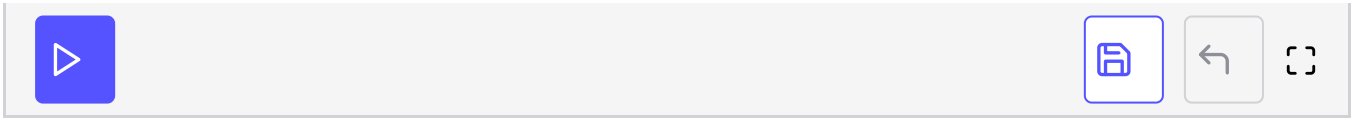


Effectively, what this means is that if we use a constant variable, we can't use `=` to change the variable. Anything else is fair game.

No empty initializations

An empty initialization is not allowed with `const`. Whereas using `var x;` or `let x;` would put the value `undefined` in `x`, `const x;` throws an error. It must be given a value on declaration.

```
var a;
let b;
const c; // -> Uncaught SyntaxError: Missing initializer in const declaration
```



That concludes this section and our understanding of the fundamentals of variables and functions. Fantastic work so far.

I'll see you in the next section.