let & const

WE'LL COVER THE FOLLOWING

- Scope
- let
  - block scope vs function scope
  - Temporal Dead Zone
- const
  - Not immutable.

In ES6 there are two new keywords available that allow us to define variables: `let` and `const`.

Before ES6, if you wanted to create a variable you would use the `var` keyword, and along with this keyword you would also define a name.

```
var author = "Ryan Christiani";
```

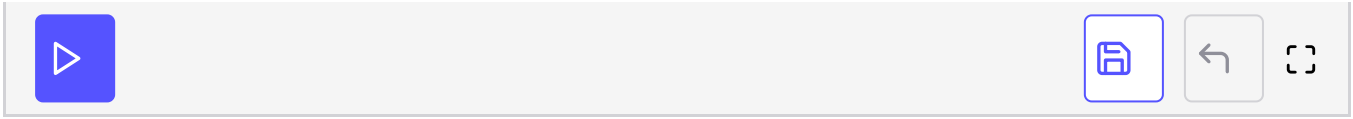
We use the equals sign to assign the value to the variable. To use the variable and the value stored inside, we simply need to write `author`. This would refer to the value `"Ryan Christiani"`.

You can create a variable without the use of the `var` keyword, but this is discouraged as it creates an undeclared global variable. But, what do we mean when we say a global variable?

Scope

In programming languages we have this idea of scope: when you define a variable it is scoped to a specific section of code. Let's look at some code and break it down:

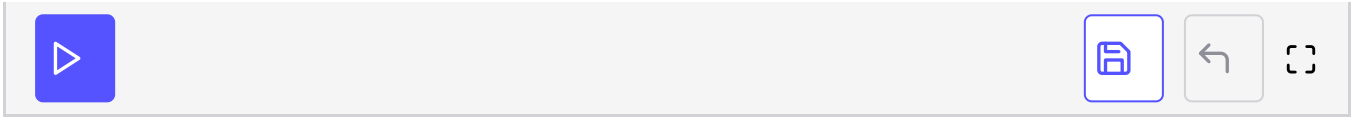
```
var rocket = "Falcon9";

function launch() {
    console.log('Launching ' + rocket);
}

launch(); // Launching Falcon9
```



In the example above, we have created a variable called `rocket` and it has been created outside of our `launch` function. Because it was created outside of the function it will be accessible inside of it via a closure. When a function is defined in JavaScript it will capture the variables that were available to it and make them available inside of the function.

So when we run `launch();` we see the text `Launching Falcon9`. If we changed the definition of the `rocket` variable so that is it is defined inside of a function called `rocketName()`, our `launch()` function will not work.

```
function rocketName() {
    var rocket = "Flacon9";
}

function launch() {
    console.log('Launching ' + rocket);
}

launch(); // Launching undefined
```



Variables created with the `var` keyword are **function scoped**, so unless they are created outside of a function, you will have no access to them. The code above will not work, because `rocket` exists only in the `rocketName` function.

let

In ES6 one of the new keywords is `let`, and its usage is just like using `var`.

```
let author = "Ryan Christiani";
```

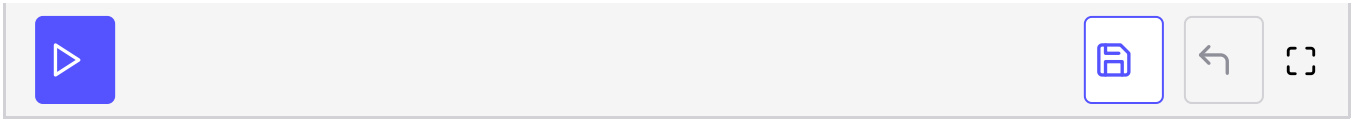
When you refer to a variable created with `let` you do it just the same as you would with `var`. Referencing `author` would get you the value of `Ryan Christiani`.

The one key difference here is that a variable defined with `let` is a **block scoped** variable. Remember when we said that a `var` was a **function scoped** variable? Let's look at the difference between block and function scope.

block scope vs function scope

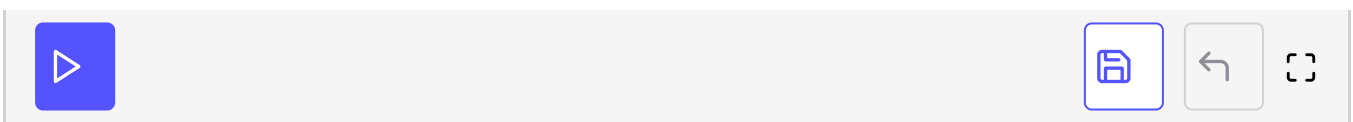
When we looked at `var` we saw that if we defined a variable inside of a function, it created the variable in there, and it is available inside of that function but not outside of it.

```
function setName() {
    var name = "Ryan";
}

console.log(name); //Undefined
```



If we declared a variable using `var` inside of a block statement, or anything with `{}` (like a conditional for example), it will make that variable available outside of that block!

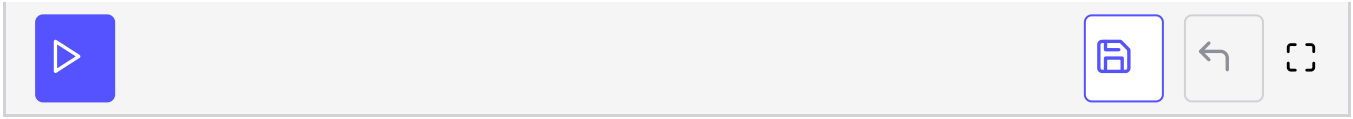
```
if(true) {
    var name = "Ryan";
}

console.log(name); //Ryan
```



And if you are familiar with other languages, say C for example, you might think this to be odd, since C has the ability to create block scope. Enter `let`, in ES6 `let` allows us to create a block scoped variable.

```
{
```

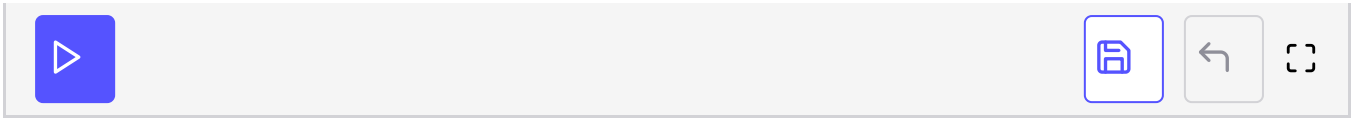
```
    let name = "Ryan";
}


console.log(name); //undefined

if(true) {
    let name = "Ryan";
}

console.log(name); //undefined
```



Variables defined with `let` will behave the same inside of a function as a variable defined with `var` would. However, now we have the benefit of block scope.

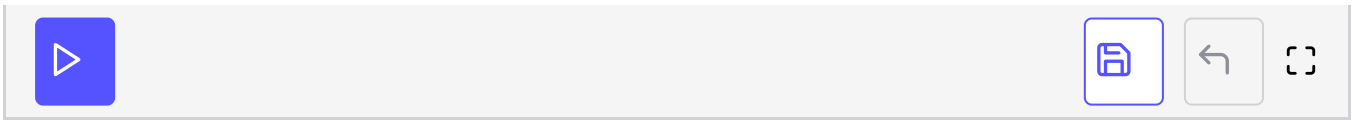
One place this works particularly well is when we are creating a variable to use in a `for` loop. Since `var` is function scoped it does not create a value that stays inside the loop.

```
for(var i = 0; i < 10; i++) {
    console.log(i);
}

console.log(i); //10
```



This can be a little confusing. If we change the variable declaration from `var` to `let`, however, it will change the behavior.

```
for(let i = 0; i < 10; i++) {
    console.log(i);
}

console.log(i); //ReferenceError: i is not defined
```

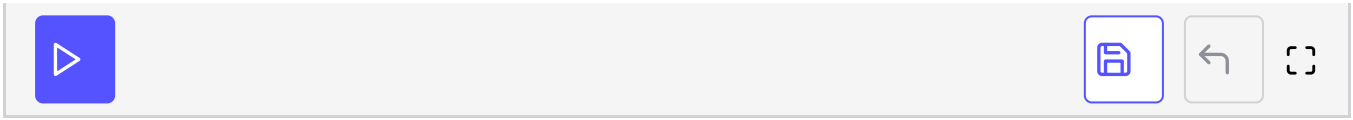


Temporal Dead Zone

There is one more gotcha that I want to point out, and this is something called the Temporal Dead Zone. In JavaScript, when the browser interprets your

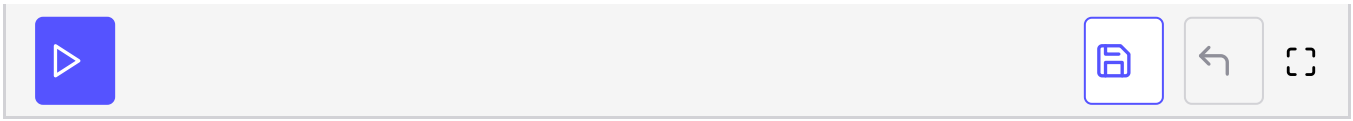
code it will do a pass where it looks for any declarations. However, it will not assign the value just yet. Because of this it is possible to use a variable before it has a value: it will simply print `undefined`.

```
console.log(person); //undefined
var person = "Ryan";
```



The browser knows there is a variable called `person`, it just doesn't know what the value is. With the new `let` keyword, this is no longer the case.

```
console.log(person); //ReferenceError
let person = 'Ryan';
```



const

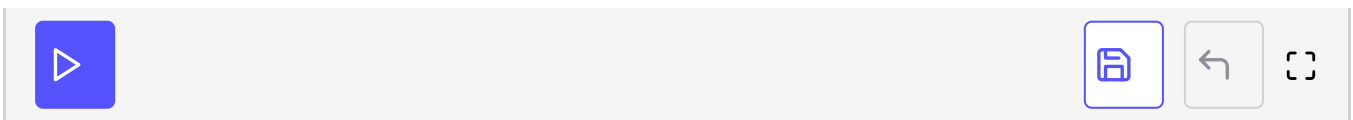
The `const` keyword behaves like `let`. It is block scope, but there is one difference that is pretty important. When you declare a variable with `const` it creates a read-only value, meaning you can use the value stored in it, but you can not reassign the value.

Before `const`, a common convention was to use uppercase names to hint that the value stored in this variable was a constant value.

```
var API_KEY = '195fhbgBqg268334mf124';
```

The obvious problem with this is that it COULD be reassigned whenever you wanted. With `const` this is not possible.

```
const api_key = '195fhbgBqg268334mf124';
api_key = 'some other key'; //TypeError: Assignment to constant variable
```



Trying to reassign the value throws a TypeError, which is great. If you defined a variable to be constant, you want to make sure it is just that.

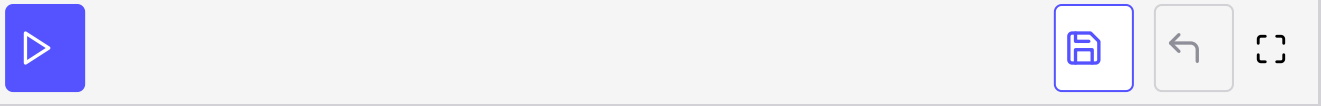
Not immutable.

There is another small gotcha to watch out for: assigning a value here does not make it immutable. We are able to assign objects to `const` variables.

```
const person = {
    name: 'Ryan Christiani'
}
```

And since a `const` variable is a read-only value, you might assume that altering the object would throw an error, however this is not the case.

```
const person = {
    name: 'Ryan Christiani'
}
person.age = 31;

console.log(person);
```



This is completely valid; since the object's keys are not protected here you are able to alter and add to them. The value stored on `person` is still an object. If you are looking for immutability, consider using `Object.freeze`. Introduced in ES5.1, this method will freeze and prevent new properties from being added.

```
const person = {
    name: 'Ryan Christiani'
}
Object.freeze(person);

// Try changing a field in frozen object
person.name = 'Instructor Ryan Christiani'
// Try adding something to a frozen object
person.age = 31;

// frozen object doesn't change
console.log(person);
```