Arrow Functions

WE'LL COVER THE FOLLOWING

In ES6 there is a new function syntax. There are some nuances to the syntax that we will be going over.

Before we get into the syntax for an Arrow Function, let's look at where the inspiration came from. CoffeeScript has had Arrow Functions for a while, there is a talk from Brendan Eich and Jeremy Ashkenas [(1)] and a post from Brendan [(2)] where they talk about how developers helped invent the future of JS. CoffeeScript was one of those languages that helped push JavaScript forward.

The syntax in CoffeeScript looks like this:

```
add = (a,b) ->
    a + b
```

For ES6 they settled on this syntax:

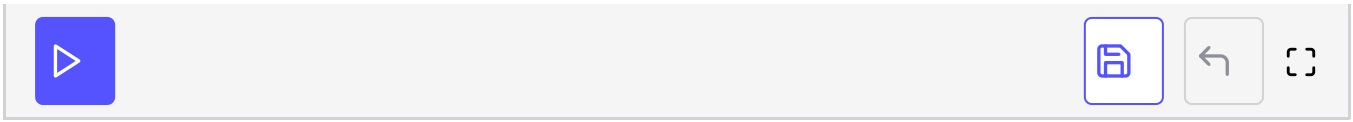
```
const add = (a,b) => {
  return a + b;
};

console.log(add(5, 10));
```

From an initial look at the syntax, you might be thinking “So, no function keyword?”. That is not quite accurate. The Arrow function can take many different forms. You can write an Arrow Function like an anonymous function.

```
const add = function(a,b) {
  return a + b;
}

const add = (a,b) => {
  return a + b;
}
```

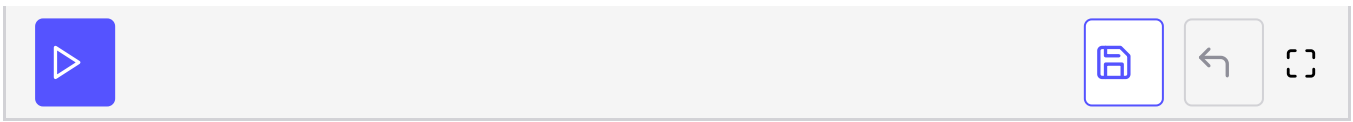
You call the functions the same way `add(2,3)`. If you have a single-line statement for your function, you can remove the parentheses and the `return` keyword to reduce everything to one line.

```
const add = (a,b) => a + b;
console.log(add(5, 10));
```



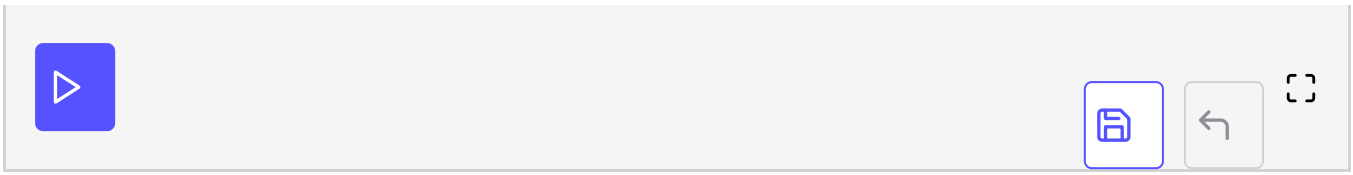
Whoa! In a single-statement Arrow Function like the one above, it is an implicit return as opposed to an explicit return. This makes the use of Arrow Functions in functional programming patterns really nice!

```
const numbers = [2,3,4,5,6];
const doubled = numbers.map((n) => n * 2);
console.log(doubled);
```



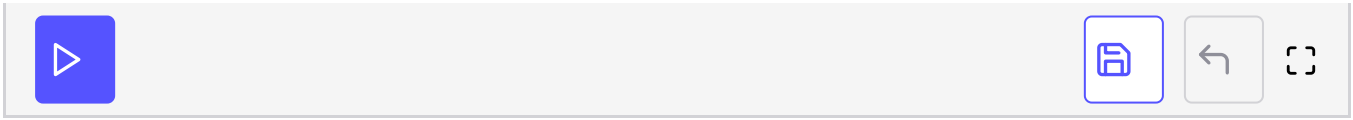
Since the `.map` method takes a callback function to perform on each element, and it is usually a very small operation, this works great. In fact if you have one parameter for your function you can even leave the parentheses off!

```
const numbers = [2,3,4,5,6];
const doubled = numbers.map(n => n * 2);
console.log(doubled);
```



That being said, if you want to have the parenthesis and curly brackets, you are totally able to do so (*but then you have to use the return statement*).

```
const numbers = [2,3,4,5,6];
const doubled = numbers.map((n) => { return n * 2; });
console.log(doubled);
```

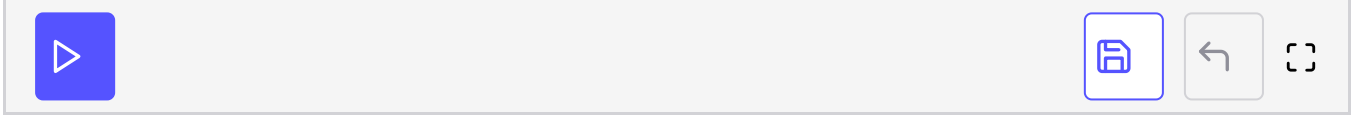


this

There is one place that the Arrow Function may get a bit confusing, and that is when we talk about the `this` keyword. The subject of `this` has been widely discussed. Sometimes it is intuitive as to what value it should be, and sometimes it is not.

Let's look at an example. On an object where we have a method, `this` is bound to the object that the method is on.

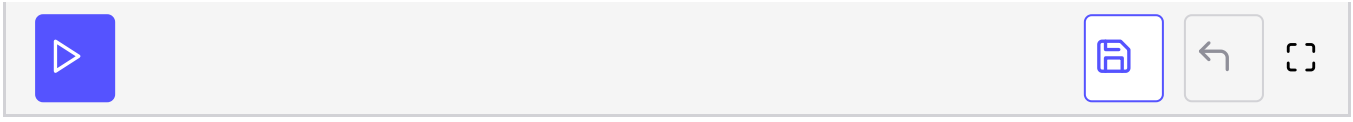
```
const person = {
  name: 'Ryan',
  sayName() {
    console.log(`Hi I am ${this.name}`);
  }
};

person.sayName();
```



In the above example `this` references the `person` object. Changing the function from an anonymous function to an Arrow Function, however, will alter the value.

```
const person = {
  name: 'Ryan',
  sayName: () => {
```

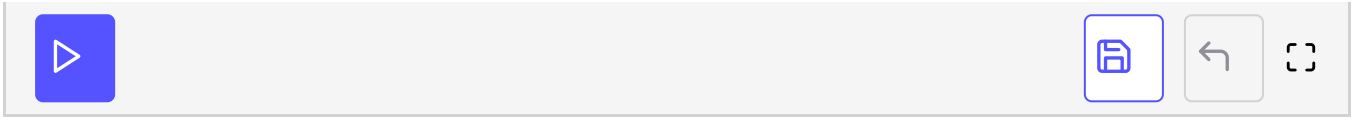
```
    console.log( `Hi I am ${this.name}` );
  }
};


person.sayName();
```



If you call the method you will get `Hi I am undefined`, and in strict mode it would throw `"TypeError: Cannot read property 'name' of undefined`. With the Arrow Function the `this` keyword is bound to what is called the lexical scope, or how I like to think of it, the parent scope. In the case of the browser, this might be the `window` object. We can, however, use this to our advantage!

We can use this to our advantage. Consider the code below, we have an object with a method. The `.showHobbies` method simply logs the hobbies to the console, or so we would hope!

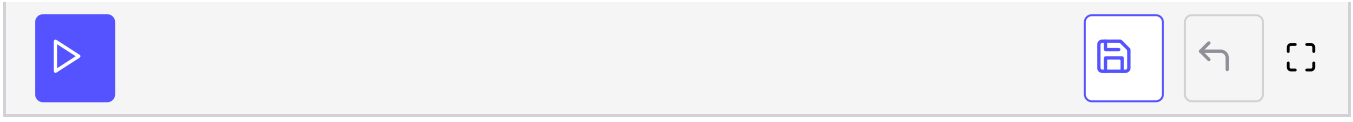
```
const person = {
    name: 'Ryan',
    hobbies: ['Robots','Games','Internet'],
    showHobbies: function() {
        this.hobbies.forEach(function(hobby){
            console.log(`${this.name} likes ${hobby}`);
        });
    }
};
person.showHobbies();
// undefined likes Robots
// undefined likes Games
// undefined likes Internet
```



This doesn't work they way we want. The `this` keyword inside of the `.forEach` here is bound to the `undefined`. In your browser (*without the strict mode*) `this` would be bound to window. We can actually pass another argument to `.forEach` for the `this` value, so we could rewrite it like:

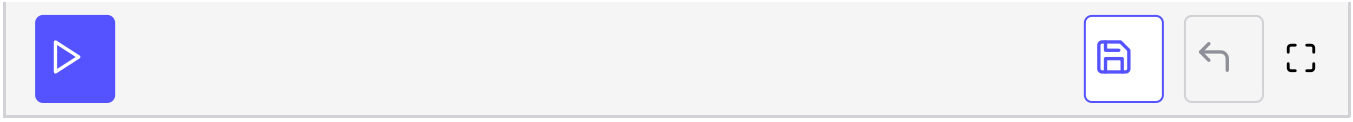
```
const person = {
  name: 'Ryan',
  hobbies: ['Robots','Games','Internet'],

  showHobbies: function() {
    this.hobbies.forEach(function(hobby){
      console.log(`${this.name} likes ${hobby}`);
```

```
    },this);
  }
};

person.showHobbies();
```



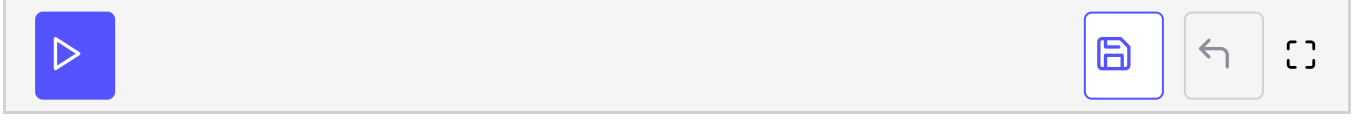
Here we pass the current `this` as the value we want to use as the context for `this` in `.forEach`. There is also a common trick that looks something like this.

```
const person = {
  name: 'Ryan',
  hobbies: ['Robots','Games','Internet'],

  showHobbies: function() {
    var self = this;
    this.hobbies.forEach(function(hobby){
      console.log(`${self.name} likes ${hobby}`);
    });
  }
};

person.showHobbies();
```



But with Arrow Functions we can use it to grab the parent scope(or the lexical scope) so that we don't have to change much of how we write our code.

```
const person = {
  name: 'Ryan',
  hobbies: ['Robots','Games','Internet'],

  showHobbies: function() {
    this.hobbies.forEach(hobby => {
        console.log(`${this.name} likes ${hobby}`);
    });
  }
};

person.showHobbies();
```

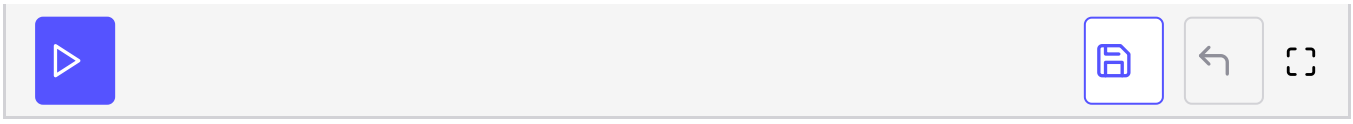


The scope for `this` in the arrow function is the same as the parent method.

Binding an Arrow Function

It should be noted that you can not bind the `this` context of an Arrow Function. Unlike a regular function, the arrow's have a lexical scope, so the scope is decided based on the call location. This means we can't use methods like `.call()`, `.apply()` or `.bind()` to change it.

```
const person = {
  city: 'Toronto'
}

const sayName = () => {
  console.log(this.city);
}

sayName.call(person); //undefined
```



arguments

There is another nuance of the Arrow Function that I want to point out: inside of an Arrow Function there is no access to the `arguments` keyword. If you are unfamiliar with `arguments` it is a keyword that supplies an array-like object that contains all of the arguments passed to the function. For more information, refer back to the Spread Operator and Rest Parameters chapter.

Additional Resources

- 1:https://youtu.be/QTj6Q_zV1yg?list=PL37ZVnwpeshGoyAz6XzjKtlgp9XrjVry_
- 2:https://brendaneich.com/tag/javascript-ecmascript-harmony-coffeescript/