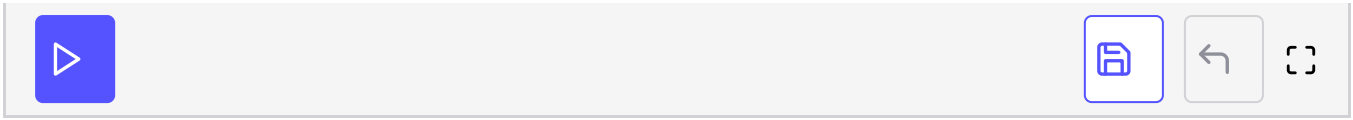
Arrow Functions () => {}

Learn an entirely new type of function. Learn arrow functions and how they allow us to write cleaner callbacks and functions in general. Learn how we can leverage their new rules of 'this' binding to make our code more intuitive and clean.

We now have a new way to write a function. The following two functions are equivalent.

```
const standardFnAdd = function(num1, num2) {
    console.log(num1 + num2);
    return num1 + num2;
}

const arrowFnAdd = (num1, num2) => {
    console.log(num1 + num2);
    return(num1 + num2);
};

standardFnAdd(2, 5); // -> 7
arrowFnAdd(2, 5); // -> 7
```



These new functions are called arrow functions. Let's break down what's happening so far. We can omit the `function` keyword. Between the parameter brackets and the function body brackets, we need to add `=>`.

This shorthand decreases the number of characters we need to write, but there's more to it than just that.

Note also that arrow functions can only be written as function *expressions*. They can't be function *declarations*.

No Return Needed

Consider the following. These two are functionally equivalent.

```
const standardFnAdd = function(num1, num2) {
```

```
    return num1 + num2;
}

const arrowFnAdd = (num1, num2) => num1 + num2;

console.log(standardFnAdd(2, 5)); // -> 7
console.log(arrowFnAdd(2, 5)); // -> 7
```

If all we have is a one-line function, we can omit the curly brackets and the return statement entirely.

If an arrow function's function body consists of a single return statement, the curly brackets around the body and the word `return` can both be omitted. The expression after the arrow will be returned automatically.

If we want to return an object from the arrow function, however, we need to wrap it in parentheses or else the arrow function reads it as a function body and doesn't do what we expect. These two are equivalent.

```
function createObject(prop) {
    return { prop: prop };
}

const arrowCreateObject = (prop) => ({ prop });
```

There's more.

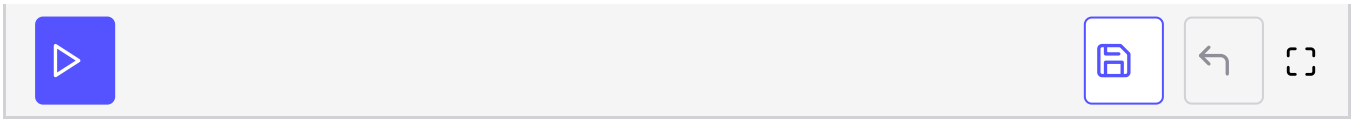
No Parentheses Needed

```
const add10 = function(num1) {
    return num1 + 10;
}

const arrowAdd10 = num1 => num1 + 10;

console.log(add10(2)); // -> 12
console.log(arrowAdd10(2)); // -> 12
```

In an arrow function, we can omit the parentheses around the function parameter if there is only one parameter.

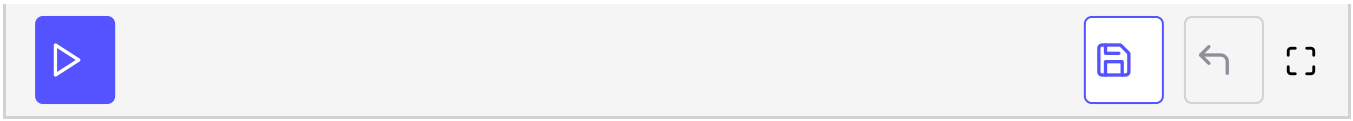
The power of arrow functions shines when they're written as callbacks. You'll write thousands of callback functions in your career. Many of them will be short, single-line functions. For example, if we have an array of objects and we wish to strip off one property of the objects, we could write this.

```
const arr = [{ prop: 'abc' }, { prop: 'def' }];
const props = arr.map(function(obj) {
    return obj.prop;
});
console.log(props); // -> [ 'abc', 'def' ]
```



Arrow functions make this easier to write but more importantly, much easier to read.

```
const arr = [{ prop: 'abc' }, { prop: 'def' }];
const props = arr.map(obj => obj.prop);
console.log(props); // -> [ 'abc', 'def' ]
```



When to Use Arrows

It may be tempting to use arrow functions exclusively now. Or, you may find them strange and unnerving and stick to standard functions. They are both tools that shine in different areas.

When writing callbacks, I practically always use arrow functions. It's clear that the function only exists to be used in one place, and therefore it doesn't need a name. It's also easier to read and write.

When writing stand-alone functions, I almost always use a standard function declaration. It's easier to find in the file and it's clear that it's a named function that's used elsewhere in the code.

As with most things in JavaScript, you'll find every viewpoint online and in the end it's up to you.

Phew. That’s it.