Introduction to `this`, `new` & OOP

Learn what object-oriented programming is and how it's implemented in JavaScript. We'll go over the theory of OOP and discuss what the keywords 'this' and 'new' mean in regards to code.

A Gentle Introduction to OOP

At the highest level, we can describe OOP as an attempt to package a set of data and functionality into one item that we can work with. The idea is that information and functions that work on that information (by manipulating it, logging it, deleting it, etc.) should be a self-contained package. It's a data-centric model.

This eliminates several problems, such as the need for global variables that might conflict with one another, and the spread of code across a file that makes it difficult to read and follow.

In JavaScript, this package that we're trying to put all of our data and functions in will be a plain object.

The keywords `this` and `new` are signals to a developer that we are using object oriented programming. They indicate that we are creating and working with packaged sets of information.

The idea of 'this'

`this` is closely tied to OOP. It is a source of great pain for many developers, even those with years of experience. It behaves differently in JavaScript than it does in other languages.

We'll use this general idea as our guideline: *In JavaScript, `this` inside a function is meant to reference the object that the function is a property of.*

The rules in JavaScript are a bit odd, but understanding the point of `this` and why it exists in programming languages should ease the explanation. It's meant to draw the developer's attention to *one single object*. When used, the

`this` keyword is basically shouting out that the main object this function is supposed to work on is `this`.

Examining 'this'

In JavaScript, *every function is called on some object* and therefore `this` has a value in every function. Let's start with a normal function call.

Useless `this` in Free Function Invocation

```
function fn() {
    // do something...
}

fn();
```

Although it seems as if `fn` is called freely, not bound to any object, it's actually stored by the JavaScript engine as a property on the global object. In a browser, the global object is `window`. If we open up our browser's console and type in the code in the block above with the following line, we can verify that `fn` is indeed a property of `window`.

```
console.log(window.fn === fn); // -> true
```

We know that the equality operator, when working on functions or other objects, checks the reference, so we can be certain that `fn` is exactly the same as `window.fn`. When we omit `window`, JavaScript implicitly enters it into our code.

What does this mean for `this`? Remember that according to our guideline, `this` in a function is supposed to represent the object that the function is being called on. In the case of simply calling `fn()`, that's exactly what we see.

```
function fn() {
    console.log(this);
}

fn(); // -> Window {frames: Window, postMessage: ƒ, …}
```

We see that `this` is equal to `window`. So it follows our guideline of representing the object the function is being called upon.
The act of calling a function freely as we're doing here is referred to as a *free function invocation*. The value of `this` in an FFI is the global object.

Why it's Useless Here

As we can see, having `this` equal the global object isn't very useful. If we want to set a variable on the global object from inside a function (something we should really never be doing), we can just omit the `var`, `let`, and `const` keywords entirely.

If we want to access something in the global scope, we still don't need `this`. If we ever did need access to the global scope we could always use `window` or whatever the global object is in the current environment.

Ideally, `this` should be something like `undefined` inside a function invoked freely. Unfortunately, it's not, and we have to deal with the issues it causes in terms of code readability and understanding.

We've covered the useless case of `this`. We'll never be using it this way, but we should be aware of the fact that this functionality exists. Let's move on to how `this` is actually used in OOP.

Method Call

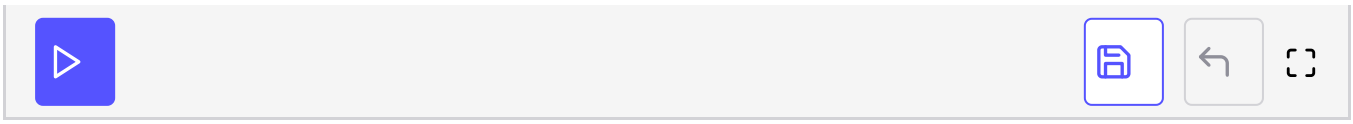
A function that is a property of an object is called a *method*. Technically, all functions are methods, as free functions are implicitly turned into methods of the global object. Let's look at a function that is explicitly a method of some object that we define.

```
const obj = {
    str: 'Hello!',
    fn: function() {
        console.log(this);
    }
};

obj.fn(); // -> { str: 'Hello!', fn: [Function: fn] }
```

The function is a property of `obj`. We see that `this` inside the function when we invoke `obj.fn()` is equal to the object `obj`. This again meets our guideline that `this` in a function should be the object that the function is on.

Let's say we want to have a counter on `obj`. We want a method on `obj` that will increment the counter.

```
const obj = {
    counter: 0,
    incrementCounter: function() {
        this.counter++;
        this.logCounter();
    },
    logCounter: function() {
        console.log(this.counter);
    }
};

obj.incrementCounter(); // -> 1
obj.incrementCounter(); // -> 2
obj.incrementCounter(); // -> 3
```



We're starting to see the crux of object oriented programming. The object is made into a self-contained item which should be manipulated using its own internal methods.

Introduction to `new`

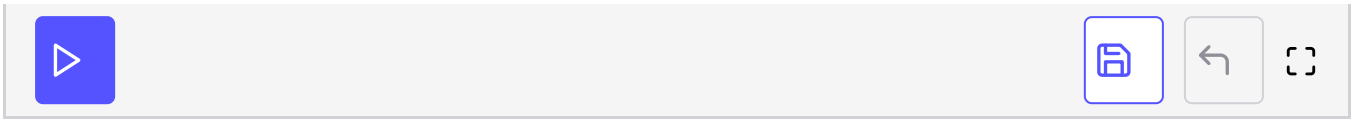
`new` is another keyword used in OOP. `new` is used to invoke a function in a different way than we're used to. The point of `new` is to indicate that we want an object returned by the function.

It indicates that we intend to use OOP and that the object the function returns is going to contain a packaged set of data and maybe methods to go along with that data.

The keyword also retains its original meaning - when we use `new`, we're always being returned a fresh, newly constructed object with a brand new reference. This object comes into existence inside the function.

```
function fn() {
}
```

```
console.log(fn()); // -> undefined
console.log(new fn()); // -> fn {}
```



Even though we have no code inside `fn`, when invoked with `new`, it still automatically returns a new object. We'll go into these rules and nuances in the next lesson. For now, we see that `new` is closely tied to the generation of objects in OOP.

Note that the item we see logged to the console due to line 5 above logs out `fn {}`. **If we log an object that was created using `new`, the console logs out the name of the function that created the object and then the object**.

This short bit of object oriented theory should have you primed for the next few lessons.