The Complete Rules to `new`

The 'new' keyword is a central part of object-oriented programming in JavaScript. This lesson will discuss how it turns a normal function into a constructor. We'll cover all of the logic that the engine inserts into our function calls when we use 'new' and how it helps us write powerful code.

Javascript's "new" Keyword Explained as Simply as Possible

Normal Function Call

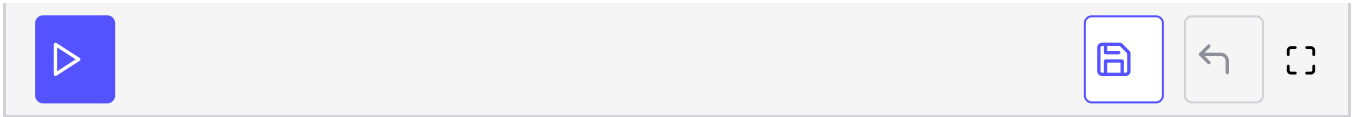
To explain what `new` does, let's start with just a normal function, called without `new`. We want to write a function that will create "`person`" objects. It'll give these objects `name` and `age` properties based on parameters that it takes in.

```
function personFn(name, age) {
    const personObj = {};

    personObj.name = name;
    personObj.age = age;

    return personObj;
}

const alex = personFn('Alex', 30);
console.log(alex); // -> { name: 'Alex', age: 30 }
```



Simple enough. We create an object, add the properties to it, and return it at the end.

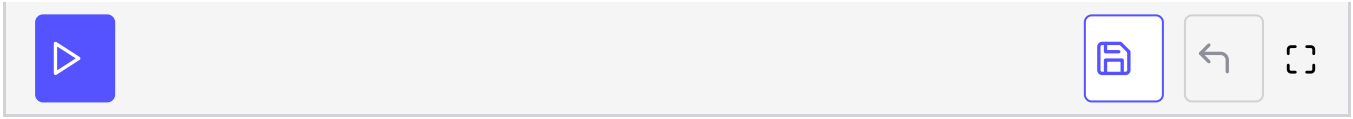
`new`

Let's create a function that does the same thing, but we want it to be invoked using `new`. This function will create the same object as the one above. Common practice is to make functions that are meant to be invoked with `new` start with a capital letter. These functions are also referred to as

start with a capital letter. These functions are also referred to as **constructors**.

```
function PersonConstructor(name, age) {
    this.name = name;
    this.age = age;
}

const christa = new PersonConstructor('Christa', 30);
console.log(christa); // -> { name: 'Christa', age: 30 }
```



Invoking `personFn` normally and invoking `PersonConstructor` with `new` both result in the same object being created. What's going on?

The `new` keyword invokes a function in a special way. It adds some implicit code that we don't see. Let's expand the above function to show everything that's happening. Below, the commented lines are pseudo-code representing functionality that is implicitly added by the JS engine when using new.

NOTE: The following code and explanation make references to prototypes and the `__proto__` property. At this point, you probably aren't familiar with them. Don't worry about it. We'll cover them in a future lesson.

```
function PersonConstructor(name, age) {
    // this = {};
    // this.__proto__ = PersonConstructor.prototype;

    // if (there is a return statement
    // in the function body that
    // returns anything EXCEPT an
    // object, array, or function) {
    //     return 'this' (the newly
    //     constructed object)
    //     instead of that item at
    //     the return statement;
    // }

    this.name = name;
    this.age = age;

    // return this;
}
```

Let's break it down. `new`:

1. Creates a new object and binds it to the `this` keyword.

2. Sets the object's internal prototype-inheritance property, `__proto__`, to be the `prototype` of the constructing function. This also makes it so the constructor of the new object is prototypically inherited.

3. Sets up logic such that if a variable of any type other than object, array, or function is returned in the function body, return `this`, the newly constructed object, instead of what the function says to return.

4. At the end of the function, returns `this` if there is no return statement in the function body.

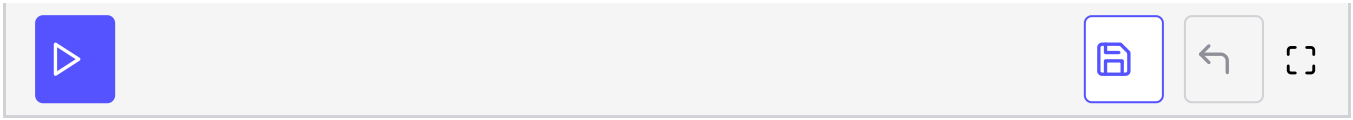
Let's show that these statements are valid, one by one.

```
function Demo() {
    console.log(this);
    this.value = 5;
    return 10;
}

/*1*/ const demo = new Demo(); // -> Demo {}
/*2*/ console.log(demo.__proto__ === Demo.prototype); // -> true
/*3*/ console.log(demo); // -> Demo { value: 5 }

function SecondDemo() {
    this.val = '2nd demo';
}

/*4*/ console.log(new SecondDemo()); // -> SecondDemo { val: '2nd demo' }
```



`new` and OOP

We can see how `new` ties in with OOP. It makes it so that our function automatically constructs a new object for us (hence the term constructor) and also returns it at the end. It frees us from creating and returning the object ourselves and allows us to focus on adding the properties we want on the object.

When we see `new` being used, we should automatically see that the purpose of the function is to create an object and we should expect that object being returned to us.

Calling a non-constructor with ‘new’

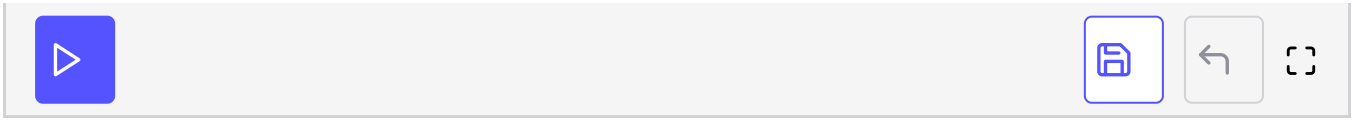
What happens if we invoke a normal function like `personFn` using `new`? Nothing special. The same rules apply. in the case of `personFn`, we see nothing explicitly happening.

```
function personFn(name, age) {
    const personObj = {};

    personObj.name = name;
    personObj.age = age;

    return personObj;
}

const alex = new personFn('Alex', 30);
console.log(alex); // -> { name: 'Alex', age: 30 }
```



Why? Let’s add our implicit code in to `personFn`.

```
function personFn(name, age) {
    // this = {};
    // this.constructor = PersonConstructor;
    // this.__proto__ = PersonConstructor.prototype;

    // if (there is a return statement
    // in the function body that
    // returns anything EXCEPT an
    // object, array, or function) {
    //     return 'this' (the newly
    //     constructed object)
    //     instead of that item at
    //     the return statement;
    // }

    const personObj = {};

    personObj.name = name;
    personObj.age = age;

    return personObj;

    // return this;
}
```

The implicit code is still added in:

- It binds `this` to a new object and sets its constructor and prototype.

- It adds logic that will return `this` instead of a non-object.
- It adds an implicit `return this;` statement at the end.

This doesn't affect our code, since we don't use the `this` keyword in our code. We also explicitly return an object, `personObj`, so the returning logic and the `return this;` line have no use.

Effectively, using `new` to invoke our function here has no effect on the output. If we were using `this` or if we weren't returning an object, the function would have different effects when invoked with and without `new`.

We shouldn't try to write code like this block we see directly above. If we're using `new`, we should use `this` in our function and we shouldn't create a new object ourselves. Not following this pattern will make our code more confusing to read and others reading our code may even think it's a mistake.

That's it.