Implementing Inheritance

We'll dive into the nuances of prototypal inheritance and the tools that the language gives us to bend it to our needs. We'll discuss how prototypes and object '__proto__' properties help us implement inheritance.

Inheritance

We now get to another computer science topic: inheritance. Inheritance refers to an object's ability to access methods and other properties from another object. Objects can *inherit* things from other objects. Inheritance in JavaScript works through prototypes and this form of inheritance is often called *prototypal inheritance*.

Function `prototype`s

As mentioned, functions all have a `prototype` property distinct from their `__proto__` property. It's an object. **A function's `prototype`'s `__proto__` property is equal to `Object.prototype`**. In other words:

```
function fn() {}
const protoOfPrototype = Object.getPrototypeOf(fn.prototype);

// protoOfPrototype === fn.prototype.__proto__
console.log(protoOfPrototype === Object.prototype);
// -> true
```

Function Prototypes and `new`

A function's `prototype` property shows its usefulness in object oriented programming. You might remember from the lesson on `new` that using `new` does something to the object bound to `this` in the constructor function.

> Sets the object's internal prototypal-inheritance property, `__proto__`, to

> be the `prototype` property of the constructing function.

We can now understand this.

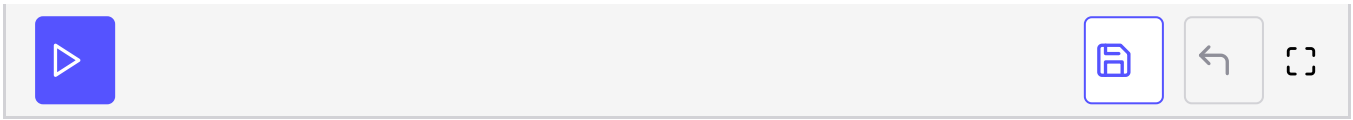
When we call a function with `new`, it sets the returned object's `__proto__` property equal to the function's `prototype` property. This is the key to inheritance.

We've assembled a few points so far:

- The `__proto__` of an object created by calling a function with `new` is equal to the `prototype` of that function
- The `__proto__` of a function's `prototype` is equal to `Object.prototype`
- The `__proto__` of `Object.prototype` is `null`

This lets us assemble the following prototype chain.

```
function Fn() {}
const obj = new Fn();

const firstProto = Object.getPrototypeOf(obj);
// firstProto === obj.__proto__
console.log(firstProto === Fn.prototype); // -> true

const secondProto = Object.getPrototypeOf(firstProto);
// secondProto === obj.__proto__.__proto__
console.log(secondProto === Object.prototype); // -> true

const thirdProto = Object.getPrototypeOf(secondProto);
// thirdProto === obj.__proto__.__proto__.__proto__
console.log(thirdProto === null); // -> true
```

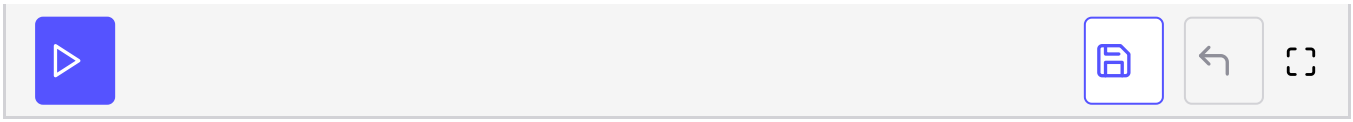


```
__proto__ === null
|
|
__proto__ === Object.prototype
|
|
__proto__ === Fn.prototype
|
|
obj
```

Implementing Inheritance

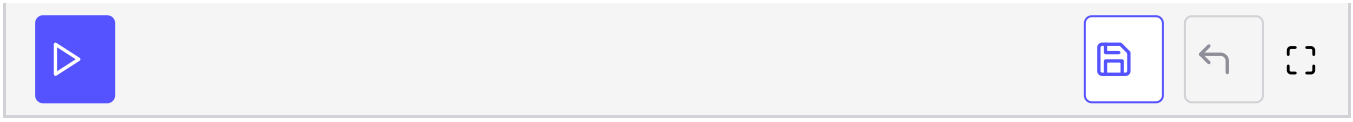
We can work with the `prototype` property directly and safely. By placing methods and other properties on a function's `prototype`, we enable all objects created by that function (using `new`) to access those properties through inheritance.

```
function Fn() {}

Fn.prototype.print = function() {
    console.log("Calling Fn.prototype's print method");
};


const obj = new Fn();
obj.print(); // -> Calling Fn.prototype's print method
```



You might be wondering what the point of this is. We can just attach this method inside the constructing function itself, like this.

```
function Fn() {
    this.print = function() {
        console.log("Calling Fn.prototype's print method");
    };
}

const obj = new Fn();
obj.print(); // -> Calling Fn.prototype's print method
```



You're right, this works. The difference is that this way, each object created by calling `new Fn()` will have its *own version* of `print` placed directly on the object. They'll be distinct functions in memory. The problem with this is performance and memory usage.

Performance

There may be times when you need thousands of new objects created from a constructor function. Using this second way of attaching `print`, we now have thousands of copies of `print`, each one attached to one of the objects.

Using the prototype chain, no matter how many objects we create out of `Fn`, we have one `print` sitting on `Fn.prototype`.

One method is not a big deal. Large programs, however, often have tens of methods that objects need. If an object needs access to 20 methods and we create 100,000 objects, the JavaScript engine has created 2,000,000 new functions.

If this needs to happen multiple times, this will cause noticable speed and memory issues. Compare this to having a total of 20 functions and giving each object the ability to use the same functions through the prototype chain. Much more scalable.

Using Chrome's `console.time` and `console.timeEnd` functions, we can directly show the difference in how long it takes. Here's the time difference of creating 2 million objects with functions directly on them vs. on the prototype. We're storing all the objects in an array.

```
> function ExampleFn() {
      this.print = function() {
          console.log("Calling print!");
      }
  }

  var objects = [];

  console.time('x');

  for (let i = 0; i < 2000000; i++) {
      objects.push(new ExampleFn());
  }

  console.timeEnd('x');
  x: 1151.960693359375ms
```

```
> function ExampleFn() {
  }

  ExampleFn.prototype.print = function() {
      console.log("Calling print!");
  }

  var objects = [];

  console.time('x');

  for (let i = 0; i < 2000000; i++) {
      objects.push(new ExampleFn());
  }

  console.timeEnd('x');
  x: 617.866943359375ms
```

As we can see, putting the `print` method on the prototype takes about half the time.

`__proto__` of Literals

As mentioned, an object's `__proto__` is equal to the `prototype` of the function that created the object. This rule applies to literals also. Remember that object literals come from `Object`, arrays come from `Array`, and functions come from `Function`.

```
console.log(
    Object.getPrototypeOf({}) === Object.prototype
); // -> true

console.log(
    Object.getPrototypeOf([]) === Array.prototype
); // -> true

console.log(
    Object.getPrototypeOf(function fn() {})
    === Function.prototype
); // -> true
```

We can now explain why we're able to call methods on arrays and objects. If we have an array `arr`, we can call `arr.map()` because the method `map` is

present on `Array.prototype`. We can call `obj.hasOwnProperty()` because

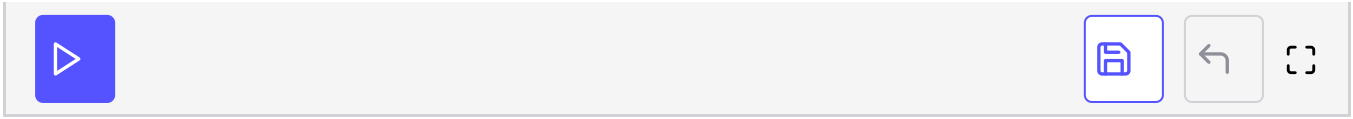
`hasOwnProperty` is present on `Object.prototype`. We've been using inheritance the whole time and didn't even know it.

The end of the `__proto__` chain of both `Array` and `Function` is equal to `Object.prototype`. They all derive from the same thing. This is why arrays, functions, and objects are all considered first-class objects in JavaScript.

`constructor`

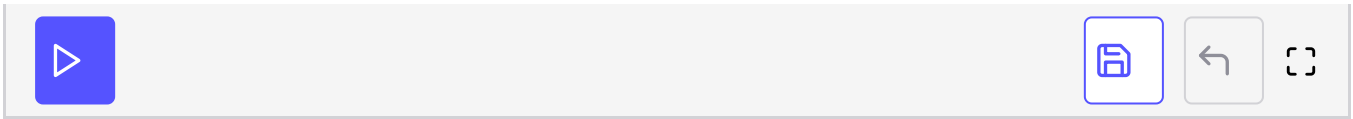
We've thrown the word constructor around a few times. Let's explain what it is. Every function's `prototype` has a `constructor` property on it that points back to the function itself. This is something the engine does for every function.

```
function Fn() {}
console.log(Fn.prototype.constructor === Fn);
// -> true
```



An object created by running `new Fn()` will have its `__proto__` equal to `Fn.prototype`. So if we were to attempt to log the `constructor` property of that object, the engine would give us `Fn` through its lookup process.

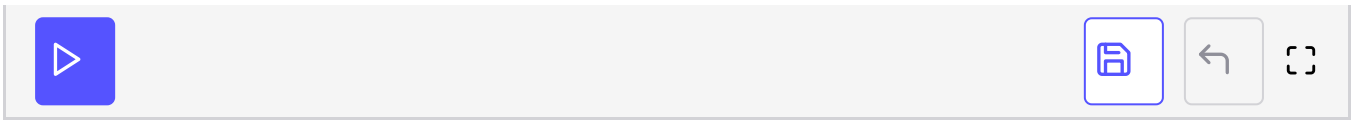
```
function Fn(){}
const obj = new Fn();

console.log(obj.constructor);
// -> [Function: Fn]
```



Why it's Useful

The `constructor` property on an object is useful because it can tell us how an object was created. Logging the `constructor` property directly on an object will tell us exactly which function created our object.

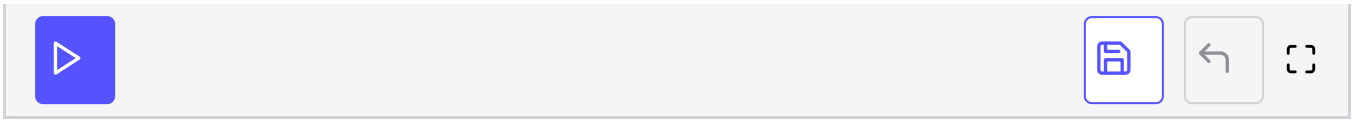
```
function Fn() {};

const normalObj = {};
const fnObj = new Fn();

console.log(normalObj.constructor); // -> [Function: Object]
console.log(fnObj.constructor); // -> [Function: Fn]
```



`Object.create`

There's a way to set the prototype of an object manually. `Object.create`. This function will take in an object as a parameter. It'll return a brand new object whose `__proto__` property is equal to the object that was passed in.

```
const prototypeObj = {
    testValue: 'Hello!'
};

const obj = Object.create(prototypeObj);
console.log(obj); // -> {}

console.log(
    Object.getPrototypeOf(obj) === prototypeObj
); // -> true

console.log(obj.testValue); // -> 'Hello!'
```



This gives us an easy way to extend the prototype chain. We can make objects inherit from any object we like, not just a function's `prototype`.

You likely won't have a need for this in the near future so I'll spare the details. If you'd like more information and examples, the MDN page for `Object.create` is a great resource.

Phew.

That was a *lot*. I know. However, this lesson gives you a deep understanding of inheritance in JavaScript.

Summary

Functions

- All functions have a `prototype` proeprty distinct from `__proto__`
- The `__proto__` of a function's `prototype` is equal to `Object.prototype`
- The `__proto__` of `Object.prototype` is `null`
- The `__proto__` of an object created by invoking a function with `new` is equal to the function's `prototype`

Prototypes

In short, inheritance in JavaScript is implemented through the prototype chain. Every normally created object, array, and function has a prototype chain of `__proto__` properties ending with `Object.prototype` at the top. This is why they're all considered first-class objects in JavaScript.

Functions have a `prototype` property in addition to the `__proto__` property. When using a constructor function with `new`, it's good practice to place methods on the function's `prototype` instead of on the object itself. The returned object's `__proto__` will be equal to the function's `prototype` so it will inherit all methods on the function's `prototype`. This prevents unnecessary memory usage and improves speed.

We can check if an object has its own property by using the `hasOwnProperty` method. We can manually set up inheritance by using `Object.create`.

Quiz

Feel free to test your understanding.

1 What will this print out?

```
const obj = {};
console.log(obj.__proto__ === Object.__proto__);
```

2

What will this print out?

```
const obj = {};
console.log(obj.__proto__ === Object.prototype);
```

3

What will this print out?

```
function Fn() {}
const obj = new Fn();
console.log(obj.__proto__ === Object.prototype);
```

4

What will this print out?

```
function Fn() {}
const obj = new Fn();
console.log(obj.__proto__ === Fn.prototype);
```

5

What will this print out?

```
function Fn() {}
const obj = new Fn();
console.log(obj.__proto__.__proto__ === Object.prototype);
```

6

What will this print out?

```
function Fn() {}
const obj = new Fn();
console.log(obj.__proto__.__proto__.__proto__);
```

7

What will this print out?

```
function Fn() {}
const obj = Object.create(Fn.prototype);
console.log(obj.__proto__ === Fn.prototype);
```

8

What will this print out?

```
function Fn() {}
const obj = Object.create(Fn);
console.log(obj.__proto__ === Fn.prototype);
```

9

What will this print out?

```
function Fn() {}
const obj1 = new Fn();
const obj2 = Object.create(obj1);
console.log(obj2.__proto__.__proto__ === Fn.prototype);
```

10

What will this print out?

```
function Fn() {}
const obj1 = new Fn();
const obj2 = Object.create(obj1);
console.log(obj2.__proto__.__proto__.__proto__ === Object.pro
totype);
```

Check Answers