Object Prototypes

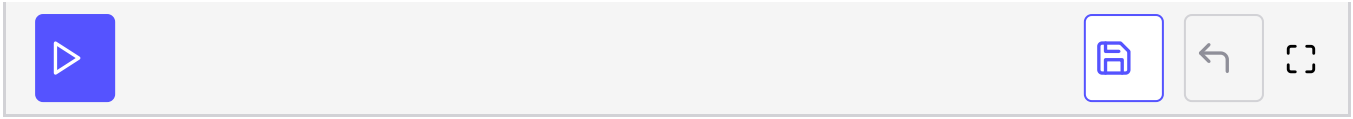
We'll discuss the '__proto__' property present on objects. It's the tool that lets us implement inheritance and is a vital part of JavaScript.

In this lesson, we'll cover a lot of seemingly unrelated topics. They're all necessary for understanding inheritance in JavaScript. We'll tie it all together in the next lesson.

`Object`, `Array`, and `Function`

JavaScript gives us access to three global functions: `Object`, `Array`, and `Function`. Yes, these are all functions.

```
console.log(Object); // -> [Function: Object]
console.log(Array); // -> [Function: Array]
console.log(Function); // -> [Function: Function]
```



You don't know it, but every time you create an object literal, the JavaScript engine is effectively calling `new Object()`. An object literal is an object created by writing `{}`, as in `var obj = {};`. So an object literal is an implicit call to `Object`.

Same goes for arrays and functions. We can think of an array as coming from the `Array` constructor and a function as coming from the `Function` constructor.

Object Prototypes

All JavaScript objects have a prototype. Browsers implement prototypes through the `__proto__` property and this is how we'll refer to it. This is often called the *dunder proto*, short for double underscore prototype. Don't EVER

reassign this property or use it directly. The MDN page for `__proto__` warns us in big red blocks to never do this.

Functions also have a `prototype` property. This is distinct from their `__proto__` property. This makes discussion rather confusing, so I'll spell out the syntax I'll be using. When I refer to a prototype and the word "prototype" isn't highlighted red, I'm referring to the `__proto__` property. When I use `prototype` in red, I'm talking about a function's `prototype` property.

The code editor on educative.io works differently than our native browser's console so many of these code blocks are purposely not runnable. The output depicted is what comes out if we were to enter the code snippets in the Chrome console.

If we were to log the prototype of an object in Chrome, this is what we'd see.

```
const obj = {};
console.log(obj.__proto__);
// -> {constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ, hasOwnProperty: ƒ, __lookupG
```

The `__proto__` property is a pointer to another object that has several properties on it. Every object literal (an object created by writing the characters `{}`) we create has this `__proto__` property pointing to this same object.

There are a few important points:

- Every object has a `__proto__` property
- The `__proto__` of an object literal is equal to `Object.prototype`, which we'll discuss soon
- The `__proto__` of `Object.prototype` is `null`

We'll explain why soon.

Property Lookup

To understand object prototypes, we need to discuss object lookup behavior. It's similar to scopes. When we look for a property of an object, the JavaScript engine will first check the object itself for the existence of the property. If not found, it'll go to the object's prototype and check that object. If found, it'll use that property.

If not found, it'll go to the prototype's prototype, and on and on until it finds an object with a `__proto__` property equal to `null`. So if we were to attempt to lookup the property `someProperty` on our `obj` object from above, the engine would first check the object itself. It wouldn't find it and would then jump to its `__proto__` object which is equal to `Object.prototype`. It wouldn't find it there either and upon seeing that the next `__proto__` is `null`, it would return `undefined`.

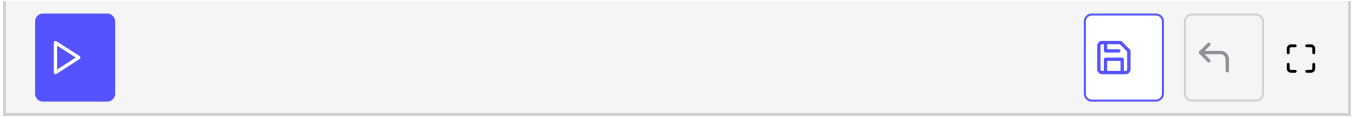
The Prototype Chain

This is called the prototype chain. It's normally described as a chain going downwards, with `null` at the very top and the object we're using at the bottom. When performing a lookup, the engine will traverse up the chain looking for the property and return the first one it finds, or `undefined` if it's not present in the prototype chain.

```
__proto__ === null
|
|
__proto__ === Object.prototype
|
|
{ object literal }
```

This can be demonstrated. Here we're going to work with `__proto__` directly for the purpose of demonstration. Again, don't ever do it.

```
const obj = {};
obj.__proto__.testValue = 'Hello!';

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



```
__proto__ === null
|
|
  proto   === Object.prototype -> testValue: 'Hello!'
```

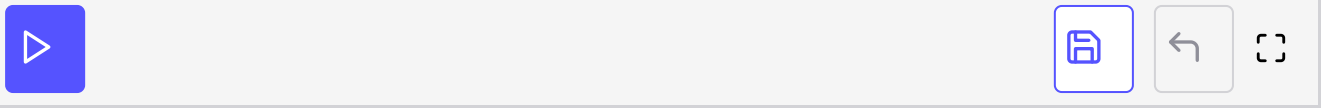
```
|


|
obj
```

When we log `obj`, we get an empty object because the property `testValue` isn't present directly on the object. However, logging `obj.testValue` triggers a lookup. The engine goes up the prototype chain and finds `testValue` present on the object's prototype and we see that value printing out.

`hasOwnProperty`

There's a method available on objects called `hasOwnProperty`. It'll return true or false based on whether the object itself contains the property being tested. Testing for `__proto__`, however, will ALWAYS return false.

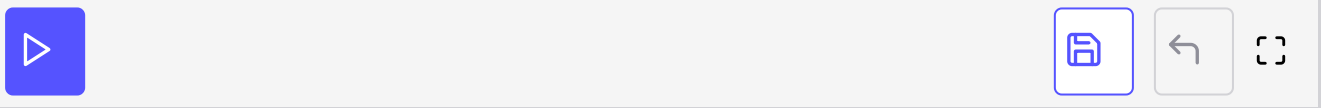
```
const obj = {};
obj.__proto__.testValue = 'Hello!';

console.log(obj.hasOwnProperty('testValue')); // -> false
console.log(obj.hasOwnProperty('__proto__')); // -> false
console.log(obj.__proto__.hasOwnProperty('testValue')); // -> true
```



`Object.getOwnPropertyNames`

We can get every property that is present directly on an object through this function.

```
const obj = { prop: 'Hi there!' };
obj.__proto__.testProp = 'Hello!';
console.log(Object.getOwnPropertyNames(obj)); // -> [ 'prop' ]
```



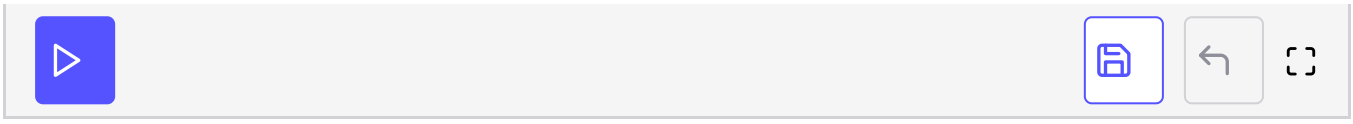
`getPrototypeOf`

Working with `__proto__` directly is always encouraged. There are other safer, more indirect ways of dealing with object prototypes.

`Object.getPrototypeOf` is a function that returns an object's `__proto__`

property.

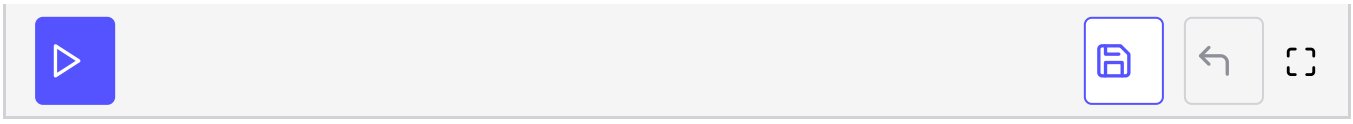
```
const obj = {};
console.log(Object.getPrototypeOf(obj) === obj.__proto__);
// -> true
```



`setPrototypeOf`

Its sister function is `Object.setPrototypeOf`. It allows us to change an object's `__proto__` property.

```
const obj = {};
const protoObj = {};
Object.setPrototypeOf(obj, protoObj);
console.log(Object.getPrototypeOf(obj) === protoObj);
// -> true
```



We'll be using these functions where applicable from here on.

Objects Summary

- We have access to `Function`, `Object`, and `Array`, three native JavaScript constructors
- All normally created objects have a `__proto__` property which should not be tampered with
- The `__proto__` property is used by the engine to perform property lookup
- `hasOwnProperty` can help reveal whether an object owns the property being used, or whether the property belongs further up the `__proto__` chain
- `Object.getOwnPropertyNames` returns an array of an object's own property keys
- `getPrototypeOf` and `setPrototypeOf` provide safer ways to interact with an object's `__proto__` property

Phew.

We'll see how prototypes help us implement inheritance in the next lesson.