Inheritance and the prototype chain

In programming, *inheritance* refers to passing down characteristics from a parent to a child so that a new piece of code can reuse and build upon the features of an existing one. JavaScript implements inheritance by using objects. Each object has an internal link to another object called its *prototype*. That prototype object has a prototype of its own, and so on until an object is reached with `null` as its prototype. By definition, `null` has no prototype and acts as the final link in this **prototype chain**. It is possible to mutate any member of the prototype chain or even swap out the prototype at runtime, so concepts like static dispatching do not exist in JavaScript.

JavaScript is a bit confusing for developers experienced in class-based languages (like Java or C++), as it is dynamic and does not have static types. While this confusion is often considered to be one of JavaScript's weaknesses, the prototypal inheritance model itself is, in fact, more powerful than the classic model. It is, for example, fairly trivial to build a classic model on top of a prototypal model — which is how classes are implemented.

Although classes are now widely adopted and have become a new paradigm in JavaScript, classes do not bring a new inheritance pattern. While classes abstract most of the prototypal mechanism away, understanding how prototypes work under the hood is still useful.

Inheritance with the prototype chain

Inheriting properties

JavaScript objects are dynamic "bags" of properties (referred to as **own properties**). JavaScript objects have a link to a prototype object. When trying to access a property of an object, the property will not only be sought on the object but on the prototype of the object, the prototype of the prototype, and so on until

either a property with a matching name is found or the end of the prototype chain is reached.

> **Note:** Following the ECMAScript standard, the notation `someObject.[[Prototype]]` is used to designate the prototype of `someObject`. The `[[Prototype]]` internal slot can be accessed and modified with the `Object.getPrototypeOf()` and `Object.setPrototypeOf()` functions respectively. This is equivalent to the JavaScript accessor `__proto__` which is non-standard but de-facto implemented by many JavaScript engines. To prevent confusion while keeping it succinct, in our notation we will avoid using `obj.__proto__` but use `obj.[[Prototype]]` instead. This corresponds to `Object.getPrototypeOf(obj)`.
>
> It should not be confused with the `func.prototype` property of functions, which instead specifies the `[[Prototype]]` to be assigned to all *instances* of objects created by the given function when used as a constructor. We will discuss the `prototype` property of constructor functions in a later section.

There are several ways to specify the `[[Prototype]]` of an object, which are listed in a later section. For now, we will use the `__proto__` syntax for illustration. It's worth noting that the `{ __proto__: ... }` syntax is different from the `obj.__proto__` accessor: the former is standard and not deprecated.

In an object literal like `{ a: 1, b: 2, __proto__: c }`, the value `c` (which has to be either `null` or another object) will become the `[[Prototype]]` of the object represented by the literal, while the other keys like `a` and `b` will become the *own properties* of the object. This syntax reads very naturally, since `[[Prototype]]` is just an "internal property" of the object.

Here is what happens when trying to access a property:

JS

```
const o = {
  a: 1,
  b: 2,
  // __proto__ sets the [[Prototype]]. It's specified here
```

```
  // as another object literal.
  __proto__: {
    b: 3,
    c: 4,
  },
};

// o.[[Prototype]] has properties b and c.
// o.[[Prototype]].[[Prototype]] is Object.prototype (we will explain
// what that means later).
// Finally, o.[[Prototype]].[[Prototype]].[[Prototype]] is null.
// This is the end of the prototype chain, as null,
// by definition, has no [[Prototype]].
// Thus, the full prototype chain looks like:
// { a: 1, b: 2 } ---> { b: 3, c: 4 } ---> Object.prototype ---> null

console.log(o.a); // 1
// Is there an 'a' own property on o? Yes, and its value is 1.

console.log(o.b); // 2
// Is there a 'b' own property on o? Yes, and its value is 2.
// The prototype also has a 'b' property, but it's not visited.
// This is called Property Shadowing

console.log(o.c); // 4
// Is there a 'c' own property on o? No, check its prototype.
// Is there a 'c' own property on o.[[Prototype]]? Yes, its value is 4.

console.log(o.d); // undefined
// Is there a 'd' own property on o? No, check its prototype.
// Is there a 'd' own property on o.[[Prototype]]? No, check its prototype.
// o.[[Prototype]].[[Prototype]] is Object.prototype and
// there is no 'd' property by default, check its prototype.
// o.[[Prototype]].[[Prototype]].[[Prototype]] is null, stop searching,
// no property found, return undefined.
```

Setting a property to an object creates an own property. The only exception to the getting and setting behavior rules is when it's intercepted by a getter or setter.

Similarly, you can create longer prototype chains, and a property will be sought on all of them.

JS

```
const o = {
  a: 1,
  b: 2,
  // __proto__ sets the [[Prototype]]. It's specified here
  // as another object literal.
  __proto__: {
    b: 3,
    c: 4,
    __proto__: {
      d: 5,
    },
  },
};

// { a: 1, b: 2 } ---> { b: 3, c: 4 } ---> { d: 5 } ---> Object.prototype ---> null

console.log(o.d); // 5
```

Inheriting "methods"

JavaScript does not have "methods" in the form that class-based languages define them. In JavaScript, any function can be added to an object in the form of a property. An inherited function acts just as any other property, including property shadowing as shown above (in this case, a form of *method overriding*).

When an inherited function is executed, the value of `this` points to the inheriting object, not to the prototype object where the function is an own property.

JS

```
const parent = {
  value: 2,
  method() {
    return this.value + 1;
  },
};

console.log(parent.method()); // 3
// When calling parent.method in this case, 'this' refers to parent

// child is an object that inherits from parent
const child = {
```

```
  __proto__: parent,
};
console.log(child.method()); // 3
// When child.method is called, 'this' refers to child.
// So when child inherits the method of parent,
// The property 'value' is sought on child. However, since child
// doesn't have an own property called 'value', the property is
// found on the [[Prototype]], which is parent.value.

child.value = 4; // assign the value 4 to the property 'value' on child.
// This shadows the 'value' property on parent.
// The child object now looks like:
// { value: 4, __proto__: { value: 2, method: [Function] } }
console.log(child.method()); // 5
// Since child now has the 'value' property, 'this.value' means
// child.value instead
```

Constructors

The power of prototypes is that we can reuse a set of properties if they should be present on every instance — especially for methods. Suppose we are to create a series of boxes, where each box is an object that contains a value which can be accessed through a `getValue` function. A naive implementation would be:

JS

```
const boxes = [
  { value: 1, getValue() { return this.value; } },
  { value: 2, getValue() { return this.value; } },
  { value: 3, getValue() { return this.value; } },
];
```

This is subpar, because each instance has its own function property that does the same thing, which is redundant and unnecessary. Instead, we can move `getValue` to the `[[Prototype]]` of all boxes:

JS

```
const boxPrototype = {
  getValue() {
    return this.value;
  },
};
```

```
const boxes = [
  { value: 1, __proto__: boxPrototype },
  { value: 2, __proto__: boxPrototype },
  { value: 3, __proto__: boxPrototype },
];
```

This way, all boxes' `getValue` method will refer to the same function, lowering memory usage. However, manually binding the `__proto__` for every object creation is still very inconvenient. This is when we would use a *constructor* function, which automatically sets the `[[Prototype]]` for every object manufactured. Constructors are functions called with `new`.

JS

```
// A constructor function
function Box(value) {
  this.value = value;
}

// Properties all boxes created from the Box() constructor
// will have
Box.prototype.getValue = function () {
  return this.value;
};

const boxes = [new Box(1), new Box(2), new Box(3)];
```

We say that `new Box(1)` is an *instance* created from the `Box` constructor function. `Box.prototype` is not much different from the `boxPrototype` object we created previously — it's just a plain object. Every instance created from a constructor function will automatically have the constructor's `prototype` property as its `[[Prototype]]` — that is, `Object.getPrototypeOf(new Box()) === Box.prototype`. `Constructor.prototype` by default has one own property: `constructor`, which references the constructor function itself — that is, `Box.prototype.constructor === Box`. This allows one to access the original constructor from any instance.

> **Note:** If a non-primitive is returned from the constructor function, that value will become the result of the `new` expression. In this case the

> `[[Prototype]]` may not be correctly bound — but this should not happen much in practice.

The above constructor function can be rewritten in classes as:

```
JS
class Box {
  constructor(value) {
    this.value = value;
  }

  // Methods are created on Box.prototype
  getValue() {
    return this.value;
  }
}
```

Classes are syntax sugar over constructor functions, which means you can still manipulate `Box.prototype` to change the behavior of all instances. However, because classes are designed to be an abstraction over the underlying prototype mechanism, we will use the more-lightweight constructor function syntax for this tutorial to fully demonstrate how prototypes work.

Because `Box.prototype` references the same object as the `[[Prototype]]` of all instances, we can change the behavior of all instances by mutating `Box.prototype`.

```
JS
function Box(value) {
  this.value = value;
}
Box.prototype.getValue = function () {
  return this.value;
};
const box = new Box(1);

// Mutate Box.prototype after an instance has already been created
Box.prototype.getValue = function () {
  return this.value + 1;
};
box.getValue(); // 2
```

A corollary is, *re-assigning* `Constructor.prototype` (`Constructor.prototype = ...`) is a bad idea for two reasons:

- The `[[Prototype]]` of instances created before the reassignment is now referencing a different object from the `[[Prototype]]` of instances created after the reassignment — mutating one's `[[Prototype]]` no longer mutates the other.
- Unless you manually re-set the `constructor` property, the constructor function can no longer be traced from `instance.constructor`, which may break user expectation. Some built-in operations will read the `constructor` property as well, and if it is not set, they may not work as expected.

`Constructor.prototype` is only useful when constructing instances. It has nothing to do with `Constructor.[[Prototype]]`, which is the constructor function's *own* prototype, which is `Function.prototype` — that is, `Object.getPrototypeOf(Constructor) === Function.prototype`.

Implicit constructors of literals

Some literal syntaxes in JavaScript create instances that implicitly set the `[[Prototype]]`. For example:

JS

```
// Object literals (without the `__proto__` key) automatically
// have `Object.prototype` as their `[[Prototype]]`
const object = { a: 1 };
Object.getPrototypeOf(object) === Object.prototype; // true

// Array literals automatically have `Array.prototype` as their `[[Prototype]]`
const array = [1, 2, 3];
Object.getPrototypeOf(array) === Array.prototype; // true

// RegExp literals automatically have `RegExp.prototype` as their `[[Prototype]]`
const regexp = /abc/;
Object.getPrototypeOf(regexp) === RegExp.prototype; // true
```

We can "de-sugar" them into their constructor form.

JS

```
const array = new Array(1, 2, 3);
const regexp = new RegExp("abc");
```

For example, "array methods" like `map()` are simply methods defined on `Array.prototype`, which is why they are automatically available on all array instances.

> **Warning:** There is one misfeature that used to be prevalent — extending `Object.prototype` or one of the other built-in prototypes. An example of this misfeature is, defining `Array.prototype.myMethod = function () {...}` and then using `myMethod` on all array instances.
>
> This misfeature is called *monkey patching*. Doing monkey patching risks forward compatibility, because if the language adds this method in the future but with a different signature, your code will break. It has led to incidents like the SmooshGate, and can be a great nuisance for the language to advance since JavaScript tries to "not break the web".
>
> The **only** good reason for extending a built-in prototype is to backport the features of newer JavaScript engines, like `Array.prototype.forEach`.

It may be interesting to note that due to historical reasons, some built-in constructors' `prototype` property are instances themselves. For example, `Number.prototype` is a number 0, `Array.prototype` is an empty array, and `RegExp.prototype` is `/(?:)/`.

JS

```
Number.prototype + 1; // 1
Array.prototype.map((x) => x + 1); // []
String.prototype + "a"; // "a"
RegExp.prototype.source; // "(?:)"
Function.prototype(); // Function.prototype is a no-op function by itself
```

However, this is not the case for user-defined constructors, nor for modern constructors like `Map`.

JS

```
Map.prototype.get(1);
// Uncaught TypeError: get method called on incompatible Map.prototype
```

Building longer inheritance chains

The `Constructor.prototype` property will become the `[[Prototype]]` of the constructor's instances, as-is — including `Constructor.prototype`'s own `[[Prototype]]`. By default, `Constructor.prototype` is a *plain object* — that is, `Object.getPrototypeOf(Constructor.prototype) === Object.prototype`. The only exception is `Object.prototype` itself, whose `[[Prototype]]` is `null` — that is, `Object.getPrototypeOf(Object.prototype) === null`. Therefore, a typical constructor will build the following prototype chain:

JS

```
function Constructor() {}

const obj = new Constructor();
// obj ---> Constructor.prototype ---> Object.prototype ---> null
```

To build longer prototype chains, we can set the `[[Prototype]]` of `Constructor.prototype` via the `Object.setPrototypeOf()` function.

JS

```
function Base() {}
function Derived() {}
// Set the `[[Prototype]]` of `Derived.prototype`
// to `Base.prototype`
Object.setPrototypeOf(Derived.prototype, Base.prototype);

const obj = new Derived();
// obj ---> Derived.prototype ---> Base.prototype ---> Object.prototype ---> null
```

In class terms, this is equivalent to using the `extends` syntax.

JS

```
class Base {}
class Derived extends Base {}
```

```
const obj = new Derived();
// obj ---> Derived.prototype ---> Base.prototype ---> Object.prototype ---> null
```

You may also see some legacy code using `Object.create()` to build the inheritance chain. However, because this reassigns the `prototype` property and removes the `constructor` property, it can be more error-prone, while performance gains may not be apparent if the constructors haven't created any instances yet.

JS

```
function Base() {}
function Derived() {}
// Re-assigns `Derived.prototype` to a new object
// with `Base.prototype` as its `[[Prototype]]`
// DON'T DO THIS — use Object.setPrototypeOf to mutate it instead
Derived.prototype = Object.create(Base.prototype);
```

Inspecting prototypes: a deeper dive

Let's look at what happens behind the scenes in a bit more detail.

In JavaScript, as mentioned above, functions are able to have properties. All functions have a special property named `prototype`. Please note that the code below is free-standing (it is safe to assume there is no other JavaScript on the webpage other than the below code). For the best learning experience, it is highly recommended that you open a console, navigate to the "console" tab, copy-and-paste in the below JavaScript code, and run it by pressing the Enter/Return key. (The console is included in most web browser's Developer Tools. More information is available for Firefox Developer Tools, Chrome DevTools, and Edge DevTools.)

JS

```
function doSomething() {}
console.log(doSomething.prototype);
// It does not matter how you declare the function; a
// function in JavaScript will always have a default
// prototype property — with one exception: an arrow
// function doesn't have a default prototype property:
```

```
const doSomethingFromArrowFunction = () => {};
console.log(doSomethingFromArrowFunction.prototype);
```

As seen above, `doSomething()` has a default `prototype` property, as demonstrated by the console. After running this code, the console should have displayed an object that looks similar to this.

```
{
  constructor: ƒ doSomething(),
  [[Prototype]]: {
    constructor: ƒ Object(),
    hasOwnProperty: ƒ hasOwnProperty(),
    isPrototypeOf: ƒ isPrototypeOf(),
    propertyIsEnumerable: ƒ propertyIsEnumerable(),
    toLocaleString: ƒ toLocaleString(),
    toString: ƒ toString(),
    valueOf: ƒ valueOf()
  }
}
```

> **Note:** The Chrome console uses `[[Prototype]]` to denote the object's prototype, following the spec's terms; Firefox uses `<prototype>`. For consistency we will use `[[Prototype]]`.

We can add properties to the prototype of `doSomething()`, as shown below.

JS

```
function doSomething() {}
doSomething.prototype.foo = "bar";
console.log(doSomething.prototype);
```

This results in:

```
{
  foo: "bar",
  constructor: ƒ doSomething(),
  [[Prototype]]: {
    constructor: ƒ Object(),
    hasOwnProperty: ƒ hasOwnProperty(),
```

```
    isPrototypeOf: ƒ isPrototypeOf(),
    propertyIsEnumerable: ƒ propertyIsEnumerable(),
    toLocaleString: ƒ toLocaleString(),
    toString: ƒ toString(),
    valueOf: ƒ valueOf()
  }
}
```

We can now use the `new` operator to create an instance of `doSomething()` based on this prototype. To use the new operator, call the function normally except prefix it with `new`. Calling a function with the `new` operator returns an object that is an instance of the function. Properties can then be added onto this object.

Try the following code:

JS

```
function doSomething() {}
doSomething.prototype.foo = "bar"; // add a property onto the prototype
const doSomeInstancing = new doSomething();
doSomeInstancing.prop = "some value"; // add a property onto the object
console.log(doSomeInstancing);
```

This results in an output similar to the following:

```
{
  prop: "some value",
  [[Prototype]]: {
    foo: "bar",
    constructor: ƒ doSomething(),
    [[Prototype]]: {
      constructor: ƒ Object(),
      hasOwnProperty: ƒ hasOwnProperty(),
      isPrototypeOf: ƒ isPrototypeOf(),
      propertyIsEnumerable: ƒ propertyIsEnumerable(),
      toLocaleString: ƒ toLocaleString(),
      toString: ƒ toString(),
      valueOf: ƒ valueOf()
    }
  }
}
```

As seen above, the `[[Prototype]]` of `doSomeInstancing` is `doSomething.prototype`. But, what does this do? When you access a property of `doSomeInstancing`, the runtime first looks to see if `doSomeInstancing` has that property.

If `doSomeInstancing` does not have the property, then the runtime looks for the property in `doSomeInstancing.[[Prototype]]` (a.k.a. `doSomething.prototype`). If `doSomeInstancing.[[Prototype]]` has the property being looked for, then that property on `doSomeInstancing.[[Prototype]]` is used.

Otherwise, if `doSomeInstancing.[[Prototype]]` does not have the property, then `doSomeInstancing.[[Prototype]].[[Prototype]]` is checked for the property. By default, the `[[Prototype]]` of any function's `prototype` property is `Object.prototype`. So, `doSomeInstancing.[[Prototype]].[[Prototype]]` (a.k.a. `doSomething.prototype.[[Prototype]]` (a.k.a. `Object.prototype`)) is then looked through for the property being searched for.

If the property is not found in `doSomeInstancing.[[Prototype]].[[Prototype]]`, then `doSomeInstancing.[[Prototype]].[[Prototype]].[[Prototype]]` is looked through. However, there is a problem: `doSomeInstancing.[[Prototype]].[[Prototype]].[[Prototype]]` does not exist, because `Object.prototype.[[Prototype]]` is `null`. Then, and only then, after the entire prototype chain of `[[Prototype]]`'s is looked through, the runtime asserts that the property does not exist and conclude that the value at the property is `undefined`.

Let's try entering some more code into the console:

JS

```
function doSomething() {}
doSomething.prototype.foo = "bar";
const doSomeInstancing = new doSomething();
doSomeInstancing.prop = "some value";
console.log("doSomeInstancing.prop:     ", doSomeInstancing.prop);
console.log("doSomeInstancing.foo:      ", doSomeInstancing.foo);
console.log("doSomething.prop:          ", doSomething.prop);
console.log("doSomething.foo:           ", doSomething.foo);
console.log("doSomething.prototype.prop:", doSomething.prototype.prop);
console.log("doSomething.prototype.foo: ", doSomething.prototype.foo);
```

This results in the following:

```
doSomeInstancing.prop:      some value
doSomeInstancing.foo:       bar
doSomething.prop:           undefined
doSomething.foo:            undefined
doSomething.prototype.prop: undefined
doSomething.prototype.foo:  bar
```

Different ways of creating and mutating prototype chains

We have encountered many ways to create objects and change their prototype chains. We will systematically summarize the different ways, comparing each approach's pros and cons.

Objects created with syntax constructs

JS

```
const o = { a: 1 };
// The newly created object o has Object.prototype as its [[Prototype]]
// Object.prototype has null as its [[Prototype]].
// o ---> Object.prototype ---> null

const b = ["yo", "sup", "?"];
// Arrays inherit from Array.prototype
// (which has methods indexOf, forEach, etc.)
// The prototype chain looks like:
// b ---> Array.prototype ---> Object.prototype ---> null

function f() {
  return 2;
}
// Functions inherit from Function.prototype
// (which has methods call, bind, etc.)
// f ---> Function.prototype ---> Object.prototype ---> null

const p = { b: 2, __proto__: o };
// It is possible to point the newly created object's [[Prototype]] to
// another object via the __proto__ literal property. (Not to be confused
// with Object.prototype.__proto__ accessors)
// p ---> o ---> Object.prototype ---> null
```

When using the `__proto__` key in object initializers, pointing the `__proto__` key to something that is not an object only fails silently without throwing an exception. Contrary to the `Object.prototype.__proto__` setter, `__proto__` in object literal initializers is standardized and optimized, and can even be more performant than `Object.create`. Declaring extra own properties on the object at creation is more ergonomic than `Object.create`.

With constructor functions

JS

```
function Graph() {
  this.vertices = [];
  this.edges = [];
}

Graph.prototype.addVertex = function (v) {
  this.vertices.push(v);
};

const g = new Graph();
// g is an object with own properties 'vertices' and 'edges'.
// g.[[Prototype]] is the value of Graph.prototype when new Graph() is executed.
```

Constructor functions have been available since very early JavaScript. Therefore, it is very fast, very standard, and very JIT-optimizable. However, it's also hard to "do properly" because methods added this way are enumerable by default, which is inconsistent with the class syntax or how built-in methods behave. Doing longer inheritance chains is also error-prone, as previously demonstrated.

With Object.create()

Calling `Object.create()` creates a new object. The `[[Prototype]]` of this object is the first argument of the function:

JS

```
const a = { a: 1 };
// a ---> Object.prototype ---> null

const b = Object.create(a);
// b ---> a ---> Object.prototype ---> null
```

```
console.log(b.a); // 1 (inherited)

const c = Object.create(b);
// c ---> b ---> a ---> Object.prototype ---> null

const d = Object.create(null);
// d ---> null (d is an object that has null directly as its prototype)
console.log(d.hasOwnProperty);
// undefined, because d doesn't inherit from Object.prototype
```

Similar to the `__proto__` key in object initializers, `Object.create()` allows directly setting the prototype of an object at creation time, which permits the runtime to further optimize the object. It also allows the creation of objects with `null` prototype, using `Object.create(null)`. The second parameter of `Object.create()` allows you to precisely specify the attributes of each property in the new object, which can be a double-edged sword:

- It allows you to create non-enumerable properties, etc., during object creation, which is not possible with object literals.
- It is much more verbose and error-prone than object literals.
- It may be slower than object literals, especially when creating many properties.

With classes

JS

```
class Rectangle {
  constructor(height, width) {
    this.name = "Rectangle";
    this.height = height;
    this.width = width;
  }
}

class FilledRectangle extends Rectangle {
  constructor(height, width, color) {
    super(height, width);
    this.name = "Filled rectangle";
    this.color = color;
  }
}
```

```
const filledRectangle = new FilledRectangle(5, 10, "blue");
// filledRectangle ---> FilledRectangle.prototype ---> Rectangle.prototype --->
Object.prototype ---> null
```

Classes offer the highest readability and maintainability when defining complex inheritance structures. Private properties are a feature with no trivial replacement in prototypal inheritance. However, classes are less optimized than traditional constructor functions and are not supported in older environments.

With Object.setPrototypeOf()

While all methods above will set the prototype chain at object creation time, `Object.setPrototypeOf()` allows mutating the `[[Prototype]]` internal property of an existing object. It can even force a prototype on a prototype-less object created with `Object.create(null)` or remove the prototype of an object by setting it to `null`.

JS

```
const obj = { a: 1 };
const anotherObj = { b: 2 };
Object.setPrototypeOf(obj, anotherObj);
// obj ---> anotherObj ---> Object.prototype ---> null
```

However, you should set the prototype during creation if possible, because setting the prototype dynamically disrupts all optimizations that engines have made to the prototype chain. It might cause some engines to recompile your code for de-optimization, to make it work according to the specs.

With the __proto__ accessor

All objects inherit the `Object.prototype.__proto__` setter, which can be used to set the `[[Prototype]]` of an existing object (if the `__proto__` key is not overridden on the object).

> **Warning:** `Object.prototype.__proto__` accessors are **non-standard** and deprecated. You should almost always use `Object.setPrototypeOf` instead.

JS

```
const obj = {};
// DON'T USE THIS: for example only.
obj.__proto__ = { barProp: "bar val" };
obj.__proto__.__proto__ = { fooProp: "foo val" };
console.log(obj.fooProp);
console.log(obj.barProp);
```

Compared to `Object.setPrototypeOf`, setting `__proto__` to something that is not an object fails silently without throwing an exception. It also has slightly better browser support. However, it is non-standard and deprecated. You should almost always use `Object.setPrototypeOf` instead.

Performance

The lookup time for properties that are high up on the prototype chain can have a negative impact on the performance, and this may be significant in the code where performance is critical. Additionally, trying to access nonexistent properties will always traverse the full prototype chain.

Also, when iterating over the properties of an object, **every** enumerable property that is on the prototype chain will be enumerated. To check whether an object has a property defined on *itself* and not somewhere on its prototype chain, it is necessary to use the `hasOwnProperty` or `Object.hasOwn` methods. All objects, except those with `null` as `[[Prototype]]`, inherit `hasOwnProperty` from `Object.prototype` — unless it has been overridden further down the prototype chain. To give you a concrete example, let's take the above graph example code to illustrate it:

JS

```
function Graph() {
  this.vertices = [];
  this.edges = [];
}

Graph.prototype.addVertex = function (v) {
  this.vertices.push(v);
};

const g = new Graph();
```

```
// g ---> Graph.prototype ---> Object.prototype ---> null

g.hasOwnProperty("vertices"); // true
Object.hasOwn(g, "vertices"); // true

g.hasOwnProperty("nope"); // false
Object.hasOwn(g, "nope"); // false

g.hasOwnProperty("addVertex"); // false
Object.hasOwn(g, "addVertex"); // false

Object.getPrototypeOf(g).hasOwnProperty("addVertex"); // true
```

Note: It is **not** enough to check whether a property is `undefined`. The property might very well exist, but its value just happens to be set to `undefined`.

Conclusion

JavaScript may be a bit confusing for developers coming from Java or C++, as it's all dynamic, all runtime, and it has no static types at all. Everything is either an object (instance) or a function (constructor), and even functions themselves are instances of the `Function` constructor. Even the "classes" as syntax constructs are just constructor functions at runtime.

All constructor functions in JavaScript have a special property called `prototype`, which works with the `new` operator. The reference to the prototype object is copied to the internal `[[Prototype]]` property of the new instance. For example, when you do `const a1 = new A()`, JavaScript (after creating the object in memory and before running function `A()` with `this` defined to it) sets `a1.[[Prototype]] = A.prototype`. When you then access properties of the instance, JavaScript first checks whether they exist on that object directly, and if not, it looks in `[[Prototype]]`. `[[Prototype]]` is looked at *recursively*, i.e., `a1.doSomething`, `Object.getPrototypeOf(a1).doSomething`, `Object.getPrototypeOf(Object.getPrototypeOf(a1)).doSomething` etc., until it's found or `Object.getPrototypeOf` returns `null`. This means that all properties defined on `prototype` are effectively shared by all instances, and you can even later change parts of `prototype` and have the changes appear in all existing instances.

If, in the example above, you do `const a1 = new A(); const a2 = new A();`, then `a1.doSomething` would actually refer to `Object.getPrototypeOf(a1).doSomething` — which is the same as the `A.prototype.doSomething` you defined, i.e., `Object.getPrototypeOf(a1).doSomething === Object.getPrototypeOf(a2).doSomething === A.prototype.doSomething`.

It is essential to understand the prototypal inheritance model before writing complex code that makes use of it. Also, be aware of the length of the prototype chains in your code and break them up if necessary to avoid possible performance problems. Further, the native prototypes should **never** be extended unless it is for the sake of compatibility with newer JavaScript features.