Understand Prototypes and Prototypal Inheritance in JavaScript

by Ifeoma Imoh | April 11, 2022 | Web | 0 Comments

UPCOMING WEBINARS

Unlike object-oriented languages that use a class-based inheritance mechanism, JavaScript's inheritance system is prototypal. This post will show you how to achieve inheritance in JavaScript through the concept of objects being able to inherit properties from other objects.

Prototypes have a significant impact on how objects behave in JavaScript. This article will assist you in effectively grasping how prototypal inheritance works in JavaScript. Let's get started!

Most object-oriented languages like Java or PHP use a class-based inheritance mechanism. **There is no concept of classes acting as a blueprint for creating objects in JavaScript.**

What Does Inheritance Mean in JavaScript?

JavaScript inheritance is more widely known as "prototypal inheritance." Prototypal inheritance uses the concept of **prototype chaining,** and it refers to the linking of objects via the prototype property. When a constructor function creates an object, it does not create it based on the constructor's prototype; instead, it creates an object linked to the constructor's prototype object.

This form of inheritance via the prototype chain is commonly referred to as delegation in JavaScript.

Constructor Functions

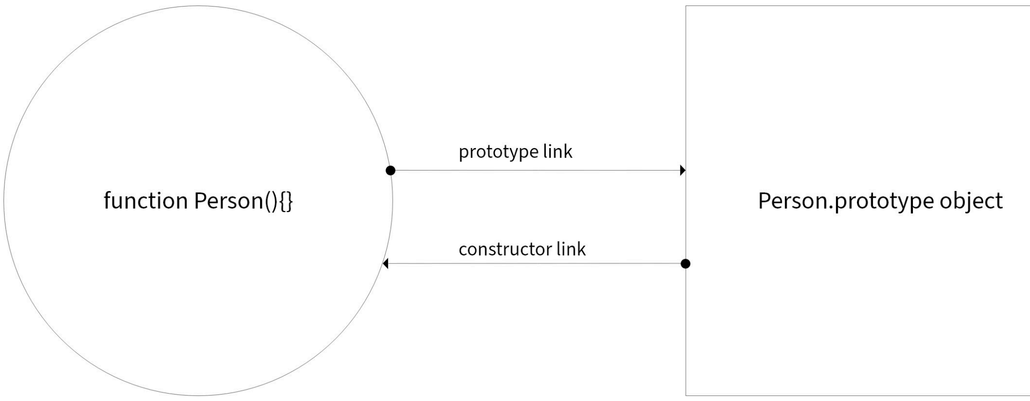
All functions in JavaScript are ordinary objects, which means they can have properties. They implement the `[[Call]]` method, but when invoked with the new operator, they become constructors (factories for making objects).

```
function Person() {} // Is an ordinary object

let ifeoma = new Person() // becomes a constructor here
```

JavaScript has several built-in objects (object, function, array, string, etc.), and every function in JavaScript is an instance of the built-in `function` object.

```
//When called, this function creates and initializes a new function object.
function Person() {}

//and is equivalent to this object creation expression
let Person = new Function()
```

Functions in JavaScript that can be used as constructors, whether built-in or user-defined, by default have a “prototype” property. This prototype property is only available on functions, not on objects.



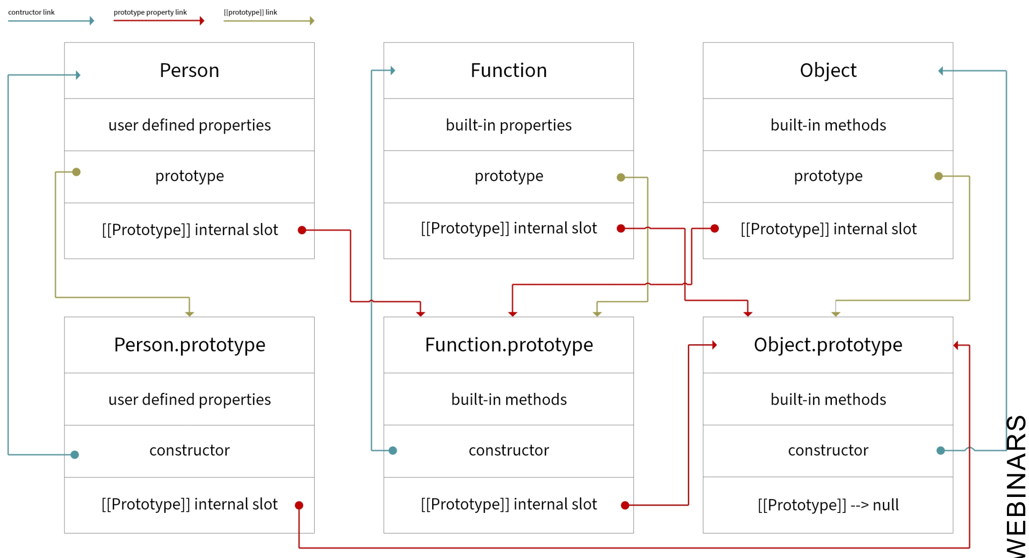
When a function is created or declared, another ordinary object (the function’s prototype object) is created internally and set as the initial value of the function’s prototype property.

```
//This function has a prototype property that points to it’s prototype object
//Arrow functions don't have the default prototype property.
function Person() {} // A function is declared

//Hypothetically speaking, this is what JavaScript does behind the scenes.
Person.prototype = {} // This is Person's prototype object.
```

This prototype object is used to enable prototype-based inheritance and shared properties. Every object created by a constructor has an implicit link or reference pointing to the value of its constructor’s prototype object. I’m going to explain this prototype object in more detail a little later on.

The built-in `Object` function is one of JavaScript’s fundamental objects. It has a prototype property on it that points to its prototype object—`Object.prototype`. `Object.prototype` is a prototypal JavaScript object and it is at the top of the prototype chain, so its `[[Prototype]]` internal slot is null.

`Object.prototype` contains a bunch of internal methods that are available to all objects (like `.toString()` and `valueOf()`). All built-in objects (string, function, etc.) in JavaScript descend from the `Object.prototype` object, and every object’s prototype chain ultimately links back to `Object.prototype`.



Some objects don't directly link to Object.prototype as their prototype; instead, they have another object with a different set of default properties.

Functions get their default properties from Function.prototype, and arrays get theirs from Array.prototype. Both Function.prototype and Array.prototype link directly to Object.prototype.

```
//Person is a function so it'll inherit or be linked to Function.prototype (which has methods apply, call, bind, etc.
function Person(name, job, hobby){
  this.name = name;
  this.job = job;
  this.hobby = hobby;
}
let ifeoma = new Person("Ifeoma", "Software developer", "Dancing");
// The chain in ascending order looks like this:
ifeoma -> Person.prototype → Object.prototype → null
//When you create an Array, it will inherit the properties of Array.prototype.
let arr = [1, "Ify", 3, true];
arr -> Array.prototype -> Object.prototype -> null
```

[[Prototype]] vs. .prototype vs. Prototype Object

All objects in JavaScript have an internal slot called [[Prototype]], which is a reference or link to another object. The value pointed at by the prototype internal slot of an object is generally referred to as "the prototype of that object."

There is a difference between an object's own [[Prototype]], a prototype property on a constructor function, and a prototype object. This is a very confusing naming choice, but I'll break it down with the example below.

```
//Basically when we declare the function Person, behind the scenes an object also gets created - Person.protc
function Person(name){
  this.name = name;
}
//Create (and return) a new empty object that has a [[Prototype]] internal slot. This [[Prototype] will be linked to
let ifeoma = new Person("Ifeoma")

// ifeoma's own [[Prototype]] will be linked to Person's prototype object because ifeoma is an instance of Perso
console.log(Object.getPrototypeOf(ifeoma) === Person.prototype); // -> true
console.log(ifeoma.__proto__ === Person.prototype);m// -> true
```

```
// Persons own [[Prototype]] will be linked to the built-in Function's prototype object because Person is an insta
console.log(Object.getPrototypeOf(Person) === Function.prototype); // -> true
console.log(Person.__proto__ === Function.prototype);m// -> true
```

- Person is a function, so by default, it is an instance of the built-in function object. The value of its own [[Prototype]] is the built-in Function.prototype object.

- Person also has a prototype object called Person.prototype—the object created when the function Person was declared. This prototype object will be the prototype of all instances created through the Person blueprint when using the new keyword—like ifeoma, as seen above.

- ifeoma is an instance of Person, so its own [[Prototype]] will be linked directly to Person.prototype.

- Functions in JavaScript that can be used as constructors have a prototype property by default that points to its prototype object. The Person prototype property points to its prototype object—Person.prototype.

> The **proto** property is an accessor property on Object.prototype with a getter (that exposes the value of an object's internal [[Prototype]]) and setter (allows an object's [[Prototype]] to be changed) function. Although it is implemented in most modern browsers, the **proto** property has been deprecated and is not standard for accessing the prototype chain. Current standards provide an equivalent Object.getPrototypeOf() method used to follow the prototype chain or retrieve the [[Prototype]] of an object since ECMAScript 2015.

Prototype Chaining

```
let ifeoma = {
  name: "Ifeoma",
  job: "Software developer",
  hobby: "Dancing"
}
```

In the snippet above, we created a simple object with the literal syntax. What if we want to create more objects with different names, jobs, etc.? There are many ways to achieve this in JavaScript, but we'll be using a constructor function.

```
// Constructor function names usually begin with a capital letter
//Define a functional object to hold persons in JavaScript
function Person(name, job, hobby){
  this.name = name;
  this.job = job;
  this.hobby = hobby;
}

//Create instances of Person
let ifeoma = new Person("Ifeoma", "Software developer", "Dancing");
```

We can create as many instances as we want from the Person constructor by prepending new to the call.

Let's see how the new operator works.

- When we put the new operator in front of a function call, the first thing that happens is a brand-new empty object is created.

- Then, it invokes the function it was called in front of with the specified arguments. Calling a function creates a new execution context with a this keyword defined to it. The value of this here will be pointing to the brand-new empty object created at first.

- Then, it adds the properties that we defined to the empty object (name, job, hobby).

- If the function doesn't return an object, it assumes that it is meant to return the this keyword, and, finally, it assigns the object to the variable (ifeoma).

For example, let's say we have some methods we want our instances to inherit or have access to. We have to add the methods to the Person prototype property. A constructor's prototype property is where we define methods and properties that we want to share across all instances.

```
Person.prototype.getName = function() {
  return this.name;
};
//Call the new method on ifeoma
ifeoma.getName(); // -> "Ifeoma"
```

Since we want the getName method to be the same for every instance, we are saving memory by keeping it in the prototype property of our constructor function, thereby creating a delegation relationship here for every instance created by Person to the Person.prototype object. This way, it only has to be in memory once, rather than for each instance. This would save you from having to duplicate code, be better for performance and require less memory.

Every instance of Person will have access to this getName method. Subsequently, properties or methods added to the Person prototype property can be accessed by all objects sharing the prototype.

When we try to access methods or properties on an object, JavaScript's default built-in [[GetPrototypeOf]] internal method performs the following:

- It checks if the property name is available inside the object itself via the [[prototype]] internal slot. If it finds it, the value will be returned.

- If it cannot find the requested property on the object directly, the operation will move up the prototype chain. The object's prototype will be searched—which is the prototype property of its parent.

- Each link in the prototype chain is walked through, one link at a time, until it finds a matching property name. The [[Prototype]] chain is terminated when it gets to the built-in Object.prototype object, which has null as its prototype and serves as the final link in this prototype chain.

- Undefined will be returned from the operation if no matching property is found.

> According to the JavaScript Specification, an object's prototype chain should have finite length (that is, starting from any object, recursively applying the [[GetPrototypeOf]] internal method to its result should eventually lead to the value **null**).

This is how the prototype chain works, and it is what makes inheritance possible. It should also be noted that if you have an object that defines a method or a property of the same name as its parent, the [[GetPrototypeOf]] method will return the object's property.

Class Syntax

The class was added in the ES6 specification that was released in 2015. It is considered a syntactic sugar layered on the current prototype-based inheritance. Apart from simple syntactic changes, a few things work differently in how they behave, but we won't cover those differences in this post.

Let's rewrite our Person function using the class syntax.

JavaScript

```
//You define a class by using the class keyword followed by the class name
class Person{

// add a constructor object with the constructor keyword
//properties that apply to each instance should be defined inside the constructor().
  constructor(name, job, hobby){
    this.name = name;
    this.job = job;
    this.hobby = hobby;
  }
  // you can add functions and properties as properties of the class object without the function or prototype key
  getName(){
    return this.name
  }
}

// Now create a new person!
let ifeoma = new Person("Ifeoma", "Software developer", "Dancing")
```

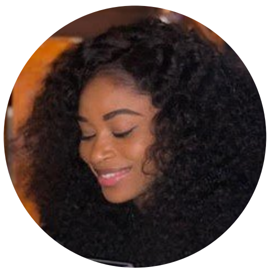
What happens when you use the class syntax isn't different from defining a constructor function that follows the prototype-based inheritance system, but using class makes your code much easier to read.

Summary

Let's recap:

- All objects in JavaScript have a hidden [[Prototype]] property that is either another object or null.
- JavaScript's inheritance system, as we have seen, is prototypal because objects can inherit properties from other objects.
- Adding methods or properties to the prototype of the constructor makes it available in memory only once and not for every instance.
- When a constructor is called with the new keyword, it creates a new object, and its prototype is set as the prototype of the resulting object.

JavaScript



ABOUT THE AUTHOR
Ifeoma Imoh

Ifeoma Imoh is a software developer and technical writer who is in love with all things JavaScript. Find her on Twitter or YouTube.

RELATED POSTS

WEB

Going Beyond ESLint: An Overview of Static Analysis in JavaScript

WEB

Operator Precedence in JavaScript

WEB

Demystifying Closures in JavaScript

WEB REACT

Immutability in JavaScript

UPCOMING WEBINARS

COMMENTS

0 Comments 1 **Login** ▾

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

♡ **Share** **Best** **Newest** **Oldest**

Be the first to comment.

All articles

TOPICS

- Web
- Mobile
- Desktop
- Design
- Productivity
- People
- Release
- AI

UPCOMING WEBINARS

search blogs...

Latest Stories
in Your Inbox

Subscribe to be the first to get our expert-written articles and tutorials for developers!

All fields are required

Email *

Country/Territory *

Select country/territory

Subscribe

Progress

Telerik and Kendo UI are part of Progress product portfolio. Progress is the leading provider of application development and digital experience technologies.

Company Technology Awards Press Releases Media Coverage Careers Offices