**JavaScript Prototype in Plain Language**

Prototype is a fundamental concept that every JavaScript developer must understand, and this article aims to explain JavaScript’s prototype in plain, detailed language. If you don’t understand JavaScript’s prototype after reading this blog post, please ask questions in the comments below. I will personally answer all questions.

To understand prototype in JavaScript you must understand objects in JavaScript. If you aren’t already familiar with objects, you should read my post JavaScript Objects in Detail. Also, know that a property is simply a variable defined on a function.

There are two interrelated concepts with prototype in JavaScript:

1. First, every JavaScript function has a prototype property (this property is empty by default), and you attach properties and methods on this prototype property when you want to implement inheritance. This prototype property is not enumerable; that is, it isn’t accessible in a for/in loop. But Firefox and most versions of Safari and Chrome have a \_\_proto\_\_ “pseudo” property (an alternative syntax) that allows you to access an object’s prototype property. You will likely never use this \_\_proto\_\_ pseudo property, but you should know that it exists and it is simply a way to access an object’s prototype property in some browsers.

The prototype property is used primarily for inheritance; you add methods and properties on a function’s prototype property to make those methods and properties available to instances of that function.

Consider this simple example of inheritance with the prototype property (more on inheritance later):

|  |  |
| --- | --- |
|  | function PrintStuff (myDocuments) { |
|  | ​this.documents = myDocuments; |
|  | } |
|  | ​ |
|  | ​// We add the print () method to PrintStuff prototype property so that other instances (objects) can inherit it:​ |
|  | PrintStuff.prototype.print = function () { |
|  | console.log(this.documents); |
|  | } |
|  | ​ |
|  | ​// Create a new object with the PrintStuff () constructor, thus allowing this new object to inherit PrintStuff's properties and methods.​ |
|  | ​var newObj = new PrintStuff ("I am a new Object and I can print."); |
|  | ​ |
|  | ​// newObj inherited all the properties and methods, including the print method, from the PrintStuff function. Now newObj can call print directly, even though we never created a print () method on it.​ |
|  | newObj.print (); //I am a new Object and I can print. |

1. The second concept with prototype in JavaScript is the prototype attribute. Think of the prototype attribute as a characteristic of the object; this characteristic tells us the object’s “parent”. In simple terms: An object’s prototype attribute points to the object’s “parent”—the object it inherited its properties from. The prototype attribute is normally referred to as the*prototype object*, and it is set automatically when you create a new object.To expound on this: Every object inherits properties from some other object, and it is this other object that is the object’s prototype attribute or “parent.” (You can think of the *prototype attribute* as the lineage or the parent). In the example code above, *newObj*‘s prototype is PrintStuff.prototype.

Note: All objects have attributes just like object properties have attributes. And the object attributes are *prototype, class,* and *extensible* attributes. It is this prototype attribute that we are discussing in this second example.

Also note that the \_\_proto\_\_ “pseudo” property contains an object’s prototype object (the parent object it inherited its methods and properties from).

Constructor  
Before we continue, let’s briefly examine the constructor. A constructor is a function used for initializing new objects, and you use the new keyword to call the constructor.  
For example:

|  |  |
| --- | --- |
|  | function Account () { |
|  | } |
|  | ​// This is the use of the Account constructor to create the userAccount object.​ |
|  | ​var userAccount = new Account (); |

Moreover, all objects that inherit from another object also inherit a *constructor* property. And this constructor property is simply a property (like any variable) that holds or points to the constructor of the object.

|  |  |
| --- | --- |
|  | //The constructor in this example is Object () ​ |
|  | ​var myObj = new Object (); |
|  | ​// And if you later want to find the myObj constructor:​ |
|  | console.log(myObj.constructor); // Object()​ |
|  | ​ |
|  | ​// Another example: Account () is the constructor​ |
|  | ​var userAccount = new Account (); |
|  | ​// Find the userAccount object's constructor​ |
|  | console.log(userAccount.constructor); // Account() |

Prototype Attribute of Objects Created with new Object () or Object Literal  
All objects created with object literals and with the Object constructor inherits fromObject.prototype. Therefore, Object.prototype is the prototype attribute (or the prototype object) of all objects created with new Object () or with {}. Object.prototype itself does not inherit any methods or properties from any other object.

|  |  |
| --- | --- |
|  | // The userAccount object inherits from Object and as such its prototype attribute is Object.prototype.​ |
|  | ​var userAccount = new Object (); |
|  | ​ |
|  | ​// This demonstrates the use of an object literal to create the userAccount object; the userAccount object inherits from Object; therefore, its prototype attribute is Object.prototype just as the userAccount object does above.​ |
|  | ​var userAccount = {name: “Mike”} |

Prototype Attribute of Objects Created With a Constructor Function  
Objects created with the new keyword and any constructor other than the Object () constructor, get their prototype from the constructor function.

For Example:

|  |  |
| --- | --- |
|  | function Account () { |
|  | ​ |
|  | } |
|  | ​var userAccount = new Account () // userAccount initialized with the Account () constructor and as such its prototype attribute (or prototype object) is Account.prototype. |

Similarly, any array such as var myArray = new Array (), gets its prototype from Array.prototype and it inherits Array.prototype’s properties.

So, there are two general ways an object’s prototype attribute is set when an object is created:

1. If an object is created with an object literal (var newObj = {}), it inherits properties from Object.prototype and we say its prototype object (or prototype attribute) is Object.prototype.
2. If an object is created from a constructor function such as new Object (), new Fruit () or new Array () or new Anything (), it inherits from that constructor (Object (), Fruit (), Array (), or Anything ()). For example, with a function such as Fruit (), each time we create a new instance of Fruit (var aFruit = new Fruit ()), the new instance’s prototype is assigned the prototype from the Fruit constructor, which is Fruit.prototype.Any object that was created with new Array () will have Array.prototype as its prototype. An object created with new Fruit () will have Fruit.prototype as its prototype. And any object created with the Object constructor (Obj (), such as var anObj = new Object() ) inherits from Object.prototype.

It is important to know that in ECMAScript 5, you can create objects with an Object.create() method that allows you to set the new object’s prototype object. We will cover ECMAScript 5 in a later post.

Why is Prototype Important and When is it Used?  
These are two important ways the prototype is used in JavaScript, as we noted above:

1. Prototype Property: Prototype-based Inheritance  
   Prototype is important in JavaScript because JavaScript does not have classical inheritance based on Classes (as most object oriented languages do), and therefore all inheritance in JavaScript is made possible through the prototype property. JavaScript has a prototype-based inheritance mechanism.Inheritance is a programming paradigm where objects (or Classes in some languages) can inherit properties and methods from other objects (or Classes). In JavaScript, you implement inheritance with the prototype property. For example, you can create a Fruit function (an object, since all functions in JavaScript are objects) and add properties and methods on the Fruit prototype property, and all instances of the Fruit function will inherit all the Fruit’s properties and methods.

Demonstration of Inheritance in JavaScript:

|  |  |
| --- | --- |
|  | function Plant () { |
|  | ​this.country = "Mexico"; |
|  | ​this.isOrganic = true; |
|  | } |
|  | ​ |
|  | ​// Add the showNameAndColor method to the Plant prototype property​ |
|  | Plant.prototype.showNameAndColor = function () { |
|  | console.log("I am a " + this.name + " and my color is " + this.color); |
|  | } |
|  | ​ |
|  | ​// Add the amIOrganic method to the Plant prototype property​ |
|  | Plant.prototype.amIOrganic = function () { |
|  | ​if (this.isOrganic) |
|  | console.log("I am organic, Baby!"); |
|  | } |
|  | ​ |
|  | ​function Fruit (fruitName, fruitColor) { |
|  | ​this.name = fruitName; |
|  | ​this.color = fruitColor; |
|  | } |
|  | ​ |
|  | ​// Set the Fruit's prototype to Plant's constructor, thus inheriting all of Plant.prototype methods and properties.​ |
|  | Fruit.prototype = new Plant (); |
|  | ​ |
|  | ​// Creates a new object, aBanana, with the Fruit constructor​ |
|  | ​var aBanana = new Fruit ("Banana", "Yellow"); |
|  | ​ |
|  | ​// Here, aBanana uses the name property from the aBanana object prototype, which is Fruit.prototype:​ |
|  | console.log(aBanana.name); // Banana​ |
|  | ​ |
|  | ​// Uses the showNameAndColor method from the Fruit object prototype, which is Plant.prototype. The aBanana object inherits all the properties and methods from both the Plant and Fruit functions.​ |
|  | console.log(aBanana.showNameAndColor()); // I am a Banana and my color is yellow. |

Note that the *showNameAndColor* method was inherited by the *aBanana* object even though it was defined all the way up the prototype chain on the Plant.prototype object.

Indeed, any object that uses the Fruit () constructor will inherit all the Fruit.prototype properties and methods and all the properties and methods from the Fruit’s prototype, which is Plant.prototype. This is the principal manner in which inheritance is implemented in JavaScript and the integral role the prototype chain has in the process.

For more in-depth coverage on Objective Oriented Programming in JavaScript, get Nicholas Zakas’s [Principles of Object-Oriented Programming in JavaScript](https://leanpub.com/oopinjavascript) (it is only $14.99).

1. Prototype Attribute: Accessing Properties on Objects  
   Prototype is also important for accessing properties and methods of objects. The prototype attribute (or prototype object) of any object is the “parent” object where the inherited properties were originally defined.This is loosely analogous to the way you might inherit your surname from your father—he is your “prototype parent.” If we wanted to find out where your surname came from, we would first check to see if you created it yourself; if not, the search will move to your prototype parent to see if you inherited it from him. If it was not created by him, the search continues to his father (your father’s prototype parent).

Similarly, if you want to access a property of an object, the search for the property begins directly on the object. If the JS runtime can’t find the property there, it then looks for the property on the object’s prototype—the object it inherited its properties from.  
If the property is not found on the object’s prototype, the search for the property then moves to prototype of the object’s prototype (the father of the object’s father—the grandfather). And this continues until there is no more prototype (no more great-grand father; no more lineage to follow). This in essence is the prototype chain: the chain from an object’s prototype to its prototype’s prototype and onwards. And JavaScript uses this prototype chain to look for properties and methods of an object.  
If the property does not exist on any of the object’s prototype in its prototype chain, then the property does not exist and *undefined* is returned.

This prototype chain mechanism is essentially the same concept we have discussed above with the prototype-based inheritance, except we are now focusing specifically on how JavaScript accesses object properties and methods via the prototype object.

This example demonstrates the prototype chain of an object’s prototype object:

|  |  |
| --- | --- |
|  | var myFriends = {name: "Pete"}; |
|  | ​ |
|  | ​// To find the name property below, the search will begin directly on the myFriends object and will immediately find the name property because we defined the property name on the myFriend object. This could be thought of as a prototype chain with one link.​ |
|  | console.log(myFriends.name); |
|  | ​ |
|  | ​// In this example, the search for the toString () method will also begin on the myFriends’ object, but because we never created a toString method on the myFriends object, the compiler will then search for it on the myFriends prototype (the object which it inherited its properties from).​ |
|  | ​ |
|  | ​// And since all objects created with the object literal inherits from Object.prototype, the toString method will be found on Object.prototype—see important note below for all properties inherited from Object.prototype. ​ |
|  | myFriends.toString (); |
|  |  |

Object.prototype Properties Inherited by all Objects  
All objects in JavaScript inherit properties and methods from Object.prototype. These inherited properties and methods are *constructor, hasOwnProperty (), isPrototypeOf (), propertyIsEnumerable (), toLocaleString (), toString (), and valueOf ()*. ECMAScript 5 also adds 4 accessor methods to Object.prototype.

Here is another example of the prototype chain:

|  |  |
| --- | --- |
|  | function People () { |
|  | ​this.superstar = "Michael Jackson"; |
|  | } |
|  | ​// Define "athlete" property on the People prototype so that "athlete" is accessible by all objects that use the People () constructor.​ |
|  | People.prototype.athlete = "Tiger Woods"; |
|  | ​ |
|  | ​var famousPerson = new People (); |
|  | famousPerson.superstar = "Steve Jobs"; |
|  | ​ |
|  | ​// The search for superstar will first look for the superstar property on the famousPerson object, and since we defined it there, that is the property that will be used. Because we have overwritten the famousPerson’s superstar property with one directly on the famousPerson object, the search will NOT proceed up the prototype chain. ​ |
|  | console.log (famousPerson.superstar); // Steve Jobs​ |
|  | ​ |
|  | ​// Note that in ECMAScript 5 you can set a property to read only, and in that case you cannot overwrite it as we just did.​ |
|  | ​ |
|  | ​// This will show the property from the famousPerson prototype (People.prototype), since the athlete property was not defined on the famousPerson object itself.​ |
|  | console.log (famousPerson.athlete); // Tiger Woods​ |
|  | ​ |
|  | ​// In this example, the search proceeds up the prototype chain and find the toString method on Object.prototype, from which the Fruit object inherited—all objects ultimately inherits from Object.prototype as we have noted before.​ |
|  | console.log (famousPerson.toString()); // [object Object] |

All built-in constructors (Array (), Number (), String (), etc.) were created from the Object constructor, and as such their prototype is Object.prototype.