Javascript is originally a functional programming language. Not until ES5 was the concept OOP introduced to Javascript.

**Object Oriented Programming (OOP)** is the concept of liking programming to real life. This is the process of giving objects in programming the characteristics of objects in reality. E.g In reality, objects like ‘car’ have attributes like color, number of doors, number of wheel drives etc and perform some actions like start, speed, stop etc.

The replication of such characteristics of a **car** can be done in programming via OOP. In object oriented programming those attributes are called properties while the actions are called methods.

**Ways to Create Objects in Javascript**

Objects can be created using the constructor functions with prototype in ES5 and using classes in ES6.

In ES5, Object Oriented Programming (OOP) is done by creating objects using the **Constructor function**. Let’s say you want a *Person* object, you create a constructor function and pass in the properties you want to set as arguments/parameters. After passing them in, we set them to properties of the object using the ‘*this*’ keyword. See code snippet below.

function Person(firstName, lastName, dateOfBirth){

this.firstName = firstName;

this.lastName = lastName;

this.dateOfBirth = new Date(dateOfBirth);

We then instantiate the object with the constructor function as shown below.

const person1 = new Person(‘Busari’, ‘Ridwan’, ’14-09-1998’);

The code above created the **person1** object. When you *console.log(person1);* you should get an object.

console.log(person1);

// Outputs result to the console

Notice the **dateOfBirth** was displayed as an object and not as a string, this is because, we passed it to the *Date* object so we can use the *Date* methods to get the Birth Year, Birth Month and Day which you will see below.

const person2 = new Person(‘Sari’, Wan’, ’24-07-2004’);

console.log(person2.dateOfBirth.getFullYear());

// We get 2004

Now, let’s add methods which are basically functions to our **Person** object.

There are two ways to do this in ES5. We can either add the methods directly inside our *Person* object or outside using the **.prototype**. You will see the implementation below as you read on.

Adding the methods directly into the object in ES5

function Person(firstname, lastname, dob){

this.firstName = firstname;

this.lastName = lastname;

this.dateOfBirth = new Date(dob);

this.getBirthYear = function() {

return this.dateOfBirth.getFullYear();

// the getFullYear() is a method of the Date object.

}

this.getFullName = () => `${this.firstName} ${this.lastName}`;

// I used template literal and arrow function above.

}

Now let’s call our method - getBirthYear and log it to the console.

console.log(person1.getBirthYear()); // logs the birth year

console.log(person1.getFullName()); // logs the full name

Now let’s talk about the prototypes. This is actually the better way to create object methods in ES5.

function Person(firstname, lastname, dob){

this.firstName = firstname;

this.lastName = lastname;

this.dateOfBirth = new Date(dob);

}

Person.prototype.getBirthYear = function() {

return this.dateOfBirth.getFullYear();

}

Let’s do the same with the getFullName method.

Person.prototype.getBirthYear = () => `${this.firstName} ${this.lastName}`;

Let’s log the newly created methods

console.log(person2.getFullName());

That’s it for object oriented programming in Javascript ES5

Let’s now dive into Object Oriented Programming in ES6 also called ES2015.

**How to Create Object in Javascript ES6 Using class keyword**

We use classes to create objects in ES6. This does the same thing as the constructor function of ES5 and its prototype, but it’s just a prettier way to write it.

Let’s create our Person class with a constructor/method which takes the arguments and assign it to it’s properties.

class Person {

constructor(firstName, lastName, dob ){

this.lastName = lastName;

this.firstName = firstName;

this.dob = dob;

// I have deliberately changed dateOfBirth to dob for ease of writing

}

// Now let’s create our prototypes or methods.

getBirthYear(){

return this.dob.getFullYear();

}

getFullName() {

return `${this.firstName} ${this.lastName}`;

}

}

Now, let’s instantiate our newly created object.

const person1 = new Person(‘You’, ‘Me’, ’05-11-2022’);

console.log(person1.getBirthYear()); // 2022

console.log(person1.getFullName()); // Busari Ridwan

The code is doing same thing as the constructor function of ES5 and as you can see, it’s a shorter, easier and cleaner way to create an object Person.

**INTRODUCTION**

OOP - Object-oriented Programming, in general, is very useful. It helps developers to model real-world things that we want to represent inside our code, and/or provide a simple way to access functionality that would otherwise be hard or impossible to make use of.  
Getting a full understanding of how OOP works in javascript is a bit difficult especially in ES5 syntax, ES6 class made it a lot easier to use object constructor but as developers, we will run into ES5 object prototype codes along our journey and in case you do not know, ES6 class, work as object prototypes under the hood.

This article will explain javascript object in ES5 and ES6 syntax. Stay tuned!

**WHY USE OBJECT CONSTRUCTOR NOTATION?**

You must have questioned the need for using object constructor and not stick with object literals. Well, object literals are easy and straight forward to understand but let's think of a scenario whereby we want to create an object from data gotten from an input field, for example; we have a website and in our website a form field that requests for the name, email, phone number, and address of our users. We want to use the data gotten from the form field to create an object as a way to keep user data together and then create a profile for each user and make sure each user should have the same properties(in this case name, email, number, and address). Using object literal notation requires us to create an object every time for each user, for instance from our form we got these data from three (3) users:

// 1st user

const user1 = {

name: 'Precious',

email: 'precious@gmail.com',

number: '+234-810-5025-740',

address: 'Earth'

}

// 2nd User

const user2 = {

name: 'Frank',

email: 'frank@gmail.com',

number: '+234-800-5525-540',

address: 'Jupiter'

}

// 3rd User

const user3 = {

name: 'Charles',

email: 'charles@yahoo.com',

number: '+234-810-4985-376',

address: 'Mars'

}

The code is repetitive and that is against the DRY(Don't Repeat Yourself) principle of programming and we don't want that. The perfect solution to that is using object constructor notation, and then making **instances** of the object. Now let's write the above code using object constructor notation and how to make instances of an object:-

// Object Constructor(ES5)

function User(name, email, number, address){

this.name = name;

this.email = email;

this.number= number;

this.address = address;

}

// Instances

// 1st user

const user1 = new User('Precious', 'precious@gmail.com', '+234-810-5025-740', 'Earth');

// 2nd user

const user2 = new User('Frank', 'frank@gmail.com', '+234-800-5525-540', 'Jupiter');

// 3rd User

const user3 = new User('Charles', 'charles@yahoo.com', '+234-810-4985-376', 'Mars');

From the above code, we just created, we used a **constructor** function which as the name implies is a function that constructs object **instances** to create objects from the data each user submitted in the form. It is DRY and clean with the constructor notation, and values from the object can be accessed with the same syntax

// OBJECT LITERAL NOTATION

// To get the name of the first user.

console.log(user1.name) // Precious

// OBJECT CONSTRUCTOR NOTATION(ES5)

// To get the name of the first user.

console.log(user1.name) // Precious

Let us explain some keywords that are used in the constructor notation.

1. The **this** keyword: In case you do not know before now, The **this** keyword in the constructor function above refers to the object itself i.e the user, meaning by saying **this.name = name** we mean the name property of that user should be set to the value of the parameter **name**. the **this** actually means different things in different context but inside the object constructor it is as stated above
2. The **new** keyword is simply used to instantiate(create) a new object from the constructor.

**OBJECT CONSTRUCTOR IN ES5 AND ES6**

* **ES5 Syntax**
  1. Prototype and prototypal Inheritance: We have looked at how object constructor is written in ES5 syntax now let's look at what an object prototype is. Syntax:

function Dog(name, age){

// Properties

this.name = name;

this.age = age;

// Method

this.canBark = function(){

if(this.age => '180 days'){

return 'YES';

} else{

return 'NO';

}

}

}

The method in the constructor function can better be written in javascript by writing it as a prototype like this:

function Dog(name, age){

// Properties

this.name = name;

this.age = age;

}

// Object Prototype

Dog.prototype.canBark = function(){

if(this.age => '180 days'){

return 'YES';

} else{

return 'NO';

}

}

Now, What is an Object Prototype? An object prototype is an object that is associated with every instance of an object by default in JavaScript. Prototypes allow you to easily define methods to all instances of a particular object. This is very useful in that the method is applied to the prototype, so it is only stored in the memory once, but every instance of the object has access to it.  
We can also add a property to the object using a prototype which is not possible normally after declaring a constructor function, but it should only be used for properties we want all instances to share:

Dog.prototype.breed = 'German shepherd'

What if we have another object that we want to have all the properties and methods of the first object and their own special properties and/or methods, what do we do keeping DRY in mind?  
The answer to that is provided by **prototypes** **inheritance** which simply means one object inheriting properties and methods of another. for instance, we want another group of dog to inherit some properties of the first group plus their own special properties(dog weight):

// Group 1

function Group1(dogName, dogAge){

// Properties

this.dogName = dogName;

this.dogAge = dogAge;

}

// Object Prototype

Group1.prototype.canBark = function(){

if(this.dogAge => '180 days'){

return 'YES';

} else{

return 'NO';

}

}

// Group 2

function Group2(dogName, dogAge, dogWeight){

Group1.call(this, dogName, dogAge);

this.dogWeight = dogWeight;

}

To Inherit the properties from the first group we used the call() method which is used to call the contractor we want to inherit its properties, and it takes in **this** as the first parameter and then the parameters to be inherited from that constructor(in this case:- dogName and dogAge). After which we then set the special property of the object(in this case: dogWeight);  
This only inherits the properties and not the prototypes. To inherit the prototypes, we will say:

Group2.prototype = object.create(Group1.prototype);

With this, we have made the 2nd group of dogs possess all the properties and objects of the 1st group.

* **ES6 Syntax**

Classes in ES6 is same as Object constructor function in ES5 under the hood that means both work in the same way just that ES6 has a much better syntax which one of my favorite tutor(Brad Traversy) calls "syntactic sugar" and also methods are directly made prototypes(made accessible to all instances of the Class). Now let's dive in ES6 classes.

1. Declaring an ES6 class & Constructor:

class Person{

constructor(firstName, lastName, age){

this.firstName = firstName;

this.lastName = lastName; // PROPERTIES

this.age = age;

}

getAge(){

return `${this.firstName} ${this.lastName}; // METHOD

}

}

KEYWORDS EXPLANATION:  
A. **class** - is simply used to declare a class(ES6 object) it is followed by the name of the object.  
B. **constructor** - Just as we used function in ES5 syntax. constructor is used construct the object.

NOTE: Value of objects in ES^ can be accessed in same way asin ES5 and also instantiation has the same syntax.

As simple as that looks we have just written our first object using ES6 class.

1. Inheritance: Inheritance in ES6 class has a different syntax and in involves using 2 new keyword **extends** and **Super**, let's take a look at it. if we want a customer object to inherit from the person object:

class Person{

constructor(firstName, lastName, age){

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

}

getAge(){

return `${this.firstName} ${this.lastName};

}

}

class Customer extends Person{

constructor(firstName, lastName, age, memberShip){

Super(firstName, lastName, age)

this.memberShip = memberShip;

}

}

KEYWORDS EXPLANATION:  
A. **extends** : specifies that the Customer object inherit the properties and methods the Person object.  
B. **super** : Just asin call() in ES5 objects, **super** states the properties inherited but here we don't have to use the **this** keyword.

NOTE: In ES6, we do not have to write a special line of code to inherit prototypes. as we already know, prototypes are accessible by all instances of the class object and so inherited by the extending class.

1. Lastly, let's talk about a special method available in ES6( **static** Methods): Static methods come in handy when we have methods that do not make use of argument passed into instantiate(create an instance) a copy of an object and we want all instances of the object to have it. for example, if we want all instances of the Person object to have a tag of 'human' we will write:

class Person{

constructor(firstName, lastName, age){

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

}

getAge(){

return `${this.firstName} ${this.lastName};

}

static isHuman(){

return 'Is a Human'

}

}

Just as simple as that. But mind you, Static methods are defined on the class itself, and not on the prototype.

That means you cannot call a static method with the instance but with the class itself e.g calling the static method in our class above will be

Person.isHuman();

**Prototype in JavaScript**

JavaScript is a dynamic language. You can attach new properties to an object at any time as shown below.

Example: Attach property to object

function Student() {

this.name = 'John';

this.gender = 'Male';

}

var studObj1 = new Student();

studObj1.age = 15;

alert(studObj1.age); // 15

var studObj2 = new Student();

alert(studObj2.age); // undefined

As you can see in the above example, age property is attached to studObj1 instance. However, studObj2 instance will not have age property because it is defined only on studObj1 instance.

So what to do if we want to add new properties at later stage to a function which will be shared across all the instances?

The answer is **Prototype**.

The prototype is an object that is associated with every functions and objects by default in JavaScript, where function's prototype property is accessible and modifiable and object's prototype property (aka attribute) is not visible.

Every function includes prototype object by default.



The prototype object is special type of enumerable object to which additional properties can be attached to it which will be shared across all the instances of it's constructor function.

So, use prototype property of a function in the above example in order to have age properties across all the objects as shown below.

Example: prototype

function Student() {

this.name = 'John';

this.gender = 'M';

}

Student.prototype.age = 15;

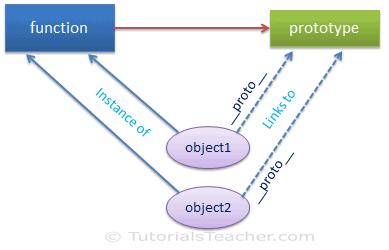
var studObj1 = new Student();

alert(studObj1.age); // 15

var studObj2 = new Student();

alert(studObj2.age); // 15

Every object which is created using literal syntax or constructor syntax with the new keyword, includes \_\_proto\_\_ property that points to prototype object of a function that created this object.



You can debug and see object's or function's prototype property in chrome or firefox's developer tool. Consider the following example.

Example: prototype

function Student() {

this.name = 'John';

this.gender = 'M';

}

var studObj = new Student();

console.log(Student.prototype); // object

console.log(studObj.prototype); // undefined

console.log(studObj.\_\_proto\_\_); // object

console.log(typeof Student.prototype); // object

console.log(typeof studObj.\_\_proto\_\_); // object

console.log(Student.prototype === studObj.\_\_proto\_\_ ); // true

As you can see in the above example, Function's prototype property can be accessed using <function-name>.prototype. However, an object (instance) does not expose prototype property, instead you can access it using \_\_proto\_\_.

The prototype property is special type of enumerable object which cannot be iterate using for..in or foreach loop.

**Object's Prototype**

As mentioned before, object's prototype property is invisible. Use Object.getPrototypeOf(obj) method instead of \_\_proto\_\_ to access prototype object.

Example: Object's prototype

function Student() {

this.name = 'John';

this.gender = 'M';

}

var studObj = new Student();

Student.prototype.sayHi= function(){

alert("Hi");

};

var studObj1 = new Student();

var proto = Object.getPrototypeOf(studObj1); // returns Student's prototype object

alert(proto.constructor); // returns Student function

The prototype object includes following properties and methods.

| **Property** | **Description** | |
| --- | --- | --- |
| constructor | Returns a function that created instance. | |
| \_\_proto\_\_ | This is invisible property of an object. It returns prototype object of a function to which it links to. | |
| **Method** | | **Description** |
| hasOwnProperty() | | Returns a boolean indicating whether an object contains the specified property as a direct property of that object and not inherited through the prototype chain. |
| isPrototypeOf() | | Returns a boolean indication whether the specified object is in the prototype chain of the object this method is called upon. |
| propertyIsEnumerable() | | Returns a boolean that indicates whether the specified property is enumerable or not. |
| toLocaleString() | | Returns string in local format. |
| toString() | | Returns string. |
| valueOf | | Returns the primitive value of the specified object. |

Chrome and Firefox denotes object's prototype as \_\_proto\_\_ which is public link whereas internally it reference as [[Prototype]]. Internet Explorer does not include \_\_proto\_\_. Only IE 11 includes it.

The getPrototypeOf() method is standardize since ECMAScript 5 and is available since IE 9.

**Changing Prototype**

As mentioned above, each object's prototype is linked to function's prototype object. If you change function's prototype then only new objects will be linked to changed prototype. All other existing objects will still link to old prototype of function. The following example demonstrates this scenario.

Example: Changing Prototype

function Student() {

this.name = 'John';

this.gender = 'M';

}

Student.prototype.age = 15;

var studObj1 = new Student();

alert('studObj1.age = ' + studObj1.age); // 15

var studObj2 = new Student();

alert('studObj2.age = ' + studObj2.age); // 15

Student.prototype = { age : 20 };

var studObj3 = new Student();

alert('studObj3.age = ' + studObj3.age); // 20

alert('studObj1.age = ' + studObj1.age); // 15

alert('studObj2.age = ' + studObj2.age); // 15

**Use of Prototype**

The prototype object is being used by JavaScript engine in two things, 1: to find properties and methods of an object 2: to implement inheritance in JavaScript.

function Student() {

this.name = 'John';

this.gender = 'M';

}

Student.prototype.sayHi = function(){

alert("Hi");

};

var studObj = new Student();

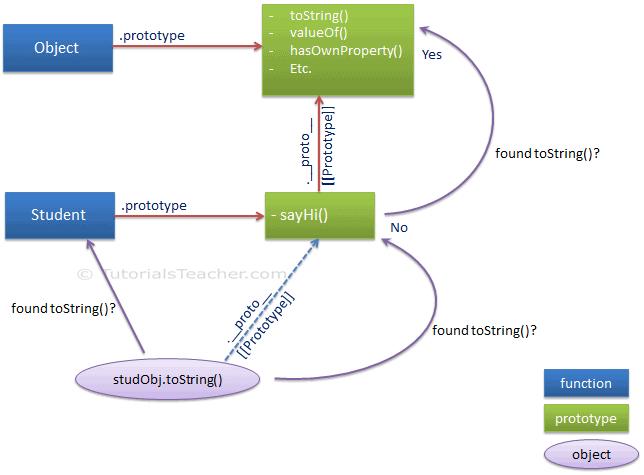
studObj.toString();

In the above example, toString() method is not defined in Student, so how and from where it finds toString()?

Here, prototype comes into picture. First of all, JavaScript engine checks whether toString() method is attached to studObj? (It is possible to attach a new function to a instance in JavaScript). If it does not find there then it uses studObj's \_\_proto\_\_ link which points to the prototype object of Student function. If it still cannot find it there then it goes up in the heirarchy and check prototype object of Object function because all the objects are derived from Object in JavaScript, and look for toString() method. Thus, it finds toString() method in the prototype object of Object function and so we can call studObj.toString().

This way, prototype is useful in keeping only one copy of functions for all the objects (instances).

The following figure illustrates the above scenario.



In JavaScript, an object can inherit properties of another object. The object from where the properties are inherited is called the prototype. In short, objects can inherit properties from other objects — the prototypes.

You’re probably wondering: why the need for inheritance in the first place? Well, inheritance solves the problem of data and logic duplication. By inheriting, objects can share properties and methods without the need of manually setting those properties and methods on each object.

**How to Access a Prototype’s Properties and Methods in JavaScript**

When we try to access a property of an object, the property is not only searched in the object itself. It's also searched in the prototype of the object, in the prototype of the prototype, and so on – until a property is found that matches the name or the end of the **prototype chain** is reached.

If the property or method isn’t found anywhere in the prototype chain, only then will JavaScript return undefined.

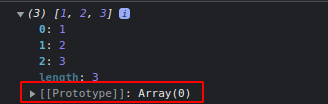
Every object in JavaScript has an internal property called [[Prototype]].

If we create an array and log it to the console like this:

const arr = [1,2,3]

console.log(arr)

We will see this:



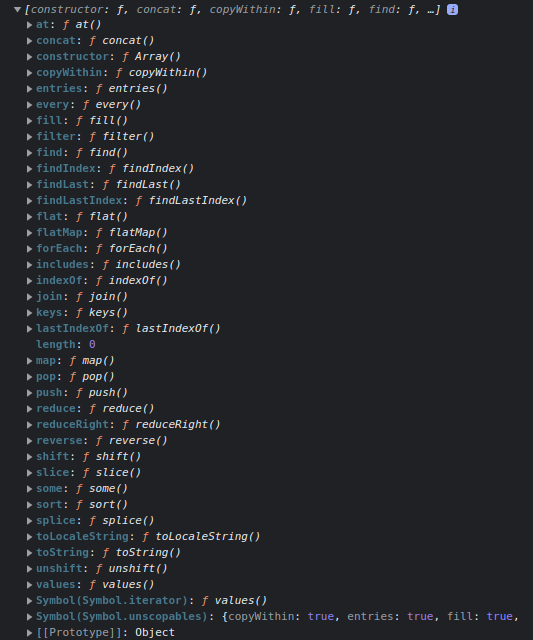
The double square brackets that enclose [[Prototype]] signify that it is an internal property, and cannot be accessed directly in code.

To find the [[Prototype]] of an object, we will use the Object.getPrototypeOf() method.

const arr = [1,2,3]

console.log(Object.getPrototypeOf(arr))

The output will consist of several built-in properties and methods:



Keep in mind that prototypes can also be changed and modified through different methods.

**The Prototype Chain**

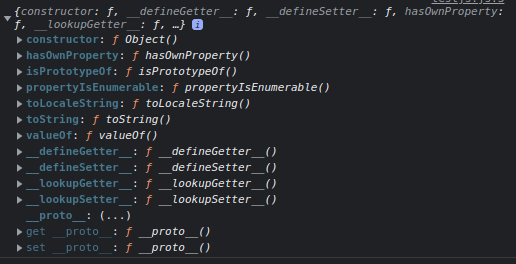
At the end of the prototype chain is Object.prototype. All objects inherit the properties and methods of Object. Any attempt to search beyond the end of the chain results in null.

If you look for the prototype of the prototype of an array, a function, or a string, you’ll see it’s an object. And that’s because in JavaScript all objects are descendants or instances of Object.prototype, which is an object that sets properties and methods to all other JavaScript data types.

const arr = [1,2,3]

const arrProto = Object.getPrototypeOf(arr)

console.log(Object.getPrototypeOf(arrProto))



Each type of prototype (for example array prototype) defines its own methods and properties, and in some cases overrides the Object.prototype methods and properties (that’s why arrays have methods that objects don’t).

But under the hood and going up the ladder of the prototype chain, **everything in JavaScript is built upon the** Object.prototype.

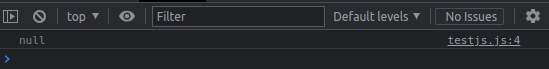
If we try to look into the prototype of Object.prototype we get null.

const arr = [1,2,3]

const arrProto = Object.getPrototypeOf(arr)

const objectProto = Object.getPrototypeOf(arrProto)

console.log(Object.getPrototypeOf(objectProto))



**A Prototype-Based Language**

JavaScript is a **prototype-based language**, meaning object properties and methods can be shared through generalized objects that have the ability to be cloned and extended.

When it comes to inheritance, JavaScript has only one structure: objects.

Each object has a private property (referred to as its [[Prototype]]) that maintains a link to another object called its prototype. That prototype object has its own prototype, and so on until an object whose prototype is null is reached.

By definition, null has no prototype, and acts as the final link in this chain of prototypes.

This is known as prototypical inheritance and differs from class inheritance. Among popular object-oriented programming languages, JavaScript is relatively unique, as other prominent languages such as PHP, Python, and Java are class-based languages, which instead define classes as blueprints for objects.

At this point you may be thinking "But we CAN implement classes on JavaScript!". And yes, we can, but as syntactic sugar. 🤫🤔

**Javascript Classes**

Classes are a way to set a blueprint to create objects with predefined properties and methods. By creating a class with specific properties and methods, you can later on instantiate objects from that class, that will inherit all the properties and methods that that class has.

In JavaScript, we can create classes in the following way:

class Alien {

constructor (name, phrase) {

this.name = name

this.phrase = phrase

this.species = "alien"

}

fly = () => console.log("Zzzzzziiiiiinnnnnggggg!!")

sayPhrase = () => console.log(this.phrase)

}

And then we can instantiate an object from that class like this:

const alien1 = new Alien("Ali", "I'm Ali the alien!")

console.log(alien1.name) // output: "Ali"

Classes are used as a way to make code more modular, organized, and understandable and are heavily used in OOP programming.

But keep in mind that JavaScript doesn’t really support classes like other languages. The class keyword was introduced with ES6 as syntactic sugar that facilitates this way of organizing code.

To visualize this, see that the same thing we did by previously defining a class, we can do it by defining a function and editing the prototype in the following way:

function Alien(name, phrase) {

this.name = name

this.phrase = phrase

this.species = "alien"

}

Alien.prototype.fly = () => console.log("Zzzzzziiiiiinnnnnggggg!!")

Alien.prototype.sayPhrase = () => console.log(this.phrase)

const alien1 = new Alien("Ali", "I'm Ali the alien!")

console.log(alien1.name) // output "Ali"

console.log(alien1.phrase) // output "I'm Ali the alien!"

alien1.fly() // output "Zzzzzziiiiiinnnnnggggg"

Any function can be invoked as a constructor with the keyword new and the prototype property of that function is used for the object to inherit methods from. In JavaScript, “class” is only used conceptually to describe the above practice – technically they’re just functions.

**What is prototypical inheritance?** [**#**](https://www.educative.io/blog/understanding-and-using-prototypal-inheritance-in-javascript#What-is-prototypical-inheritance)

A diagram of a diagram

Description automatically generated

Simply put, prototypical inheritance refers to the ability to access object properties from another object. We use a **JavaScript prototype** to add new properties and methods to an existing object constructor. We can then essentially tell our JS code to inherit properties from a prototype. Prototypical inheritance allows us to reuse the properties or methods from one JavaScript object to another through a reference pointer function.

All JavaScript objects inherit properties and methods from a prototype:

* Date objects inherit from Date.prototype.
* Array objects inherit from Array.prototype.
* Player objects inherit from Player.prototype.

The Object.prototype is on top of the prototype inheritance chain. ​ Date objects, Array objects, and Player objects all inherit from Object.prototype.

**Revisiting an old example**

Let’s walk through an example of prototypical inheritance you’re likely familiar with from grade school: all squares are rectangles, but not all rectangles are squares. If we think of this as a JS program, we could say that the rectangle is a prototype to the square: the square inherits all properties of a rectangle (i.e. four-sides and closed), while also adding a new feature (i.e. all sides are the same length).

We could not, however, construct this same concept using the square as a prototype, because there are properties of a square that do not apply to rectangles (i.e. all sides are the same length).

We can see how prototypal inheritance works on the basis of specifying categories within a group from least specific to most – from rectangle to square. In code, this concept can sometimes be lost in the syntax. If you find this happens, **speak the relations** between objects and listen to where you draw distinctions. If you hear, “all \_\_\_ are ***, but…not all \_\_\_ are***”, that is where a new prototypical relationship should be added.

**Cons of Prototypal Inheritance**

Prototypical inheritance clearly has a lot of benefits for JavaScript programmings, but, like all tools, it does have limitations. Let’s take a look at the key downsides to look out for as you write a prototype-based program:

* **Inheritance cannot flow in circles as this will create an error.** For example, if user linked premiumFamily as a prototype in the above program, an error would occur as this would create a loop.
* **Objects cannot inherit from multiple prototypes.** As we saw above, they can inherit multiple object’s properties through a chain, however another object linked as a prototype explicitly will cause an error. This is the case even if the additional prototype is within the same chain. For example, familyPremium could not have explicit links to both premiumUser and user.
* **Prototypical relationships can only be made to objects.** This is because the \_\_proto\_\_ function works as a forwarder, directing the program where to find the value it is looking for. As the program either knows where to look or it doesn’t, the function can be only either null or an object. All other types will be discarded.

**Important terms**

**\_\_proto\_\_ property**

In Javascript, every object has its own hidden, internal property, [[Prototype]]. We can access that [[Prototype]] using the \_\_proto\_\_ property. This calls the program to mark the template object as a hidden type. JavaScript objects must be linked to this prototype object. Now, an object’s properties can be accessed by the inheritor object.

Let’s take a look at the syntax for accessing and setting the [[Prototype]] property of an object.

//using \_\_proto\_\_ to access and set the [[Prototype]] of "anObject"

anObject.\_\_proto\_\_ = someotherObject

**Object.create**

JavaScript ECMAScript 5 comes with the function Object.create( ). This method can be used to replacenew. We can use it to create an empty object based on a defined prototype and then assign it to a different prototype. Take a look at the syntax:

Object.create(proto, [propertiesObject])

Object.create methods can accept two arguments: propertiesObject and prototypeObject.

**Object.prototype.constructor**

All objects have a constructor property. If an object is created without the using a constructor function, it will have a constructor property. The constructor property will return a reference to the object’s Object constructor function. It will return 1, true1, and ”test”. Take a look at an example below.

let o = {} o.constructor === Object // true

let o = new Object o.constructor === Object // true

let a = [] a.constructor === Array // true

let a = new Array a.constructor === Array // true

let n = new Number(3) n.constructor === Number // true

**hasOwnProperty**

Using hasOwnProperty, we can test if an object contains a certain prototype property; the method will return true or false depending. This will help you clarify if an object has its own property or if it is inheriting instead. Take a look at the syntax below:

obj.hasOwnProperty(prop)

**The Prototype Chain**

Prototypal inheritance uses the concept of **prototype chaining**. Let’s explore that concept. Every object created contains [[Prototype]], which points either to another object or null. Envision an object C with a [[Prototype]] property that points to object B. Object B’s [[Prototype]] property points to prototype object A. This continues onward, forming a kind of chain called the prototype chain.

This concept is used when searching our code. When we need to find a property in an object, it is first searched for in the object, and if not found, it is searched for on that object’s prototype, and so on. Thus, the entire prototype chain is traversed until the property is found or null is reached.

In the following sections, we’ll take a look at some implementations using the handling of accounts in a streaming service.

**Keep learning about front-end JavaScript.**

Master JavaScript and other front-end technologies with hands-on practice.

Educative Paths let you access all our best front-end lessons pulled from across our course library.

[**Become a Front End Developer**](https://www.educative.io/path/become-front-end-developer)

**Example Code 1: Setting up Prototypical Relationships**

For this first example, we’ll write a simple prototypical relationship between two objects, user and premiumUser, using the .\_proto\_ function. Each of these objects has their own properties which would be shared among all accounts at that tier: all users have access to stream shows, showAccess = true, and all premiumUsers have advertisements disabled, ads = false.

Ace Editor

The prototypical relationship here ensures that premiumUser inherits the showAccess property set from user without having to set it manually at the premium tier.

To check that this has inherited properly, we add a line to have the console print the current value of showAccess for premiumUser. As it returns true, we can see that premiumUser has inherited this property from user.

**Example Code 2: Inheriting Methods**

Prototypal Inheritance can be used not only to inherit properties from other objects but methods as well.

In the example below, we build off our previous code and now add email and IDnumber properties to user, tracking account info for this user, as well as a setter method, accountInfo which when called will parse a passed string, setting email and IDnumber to the new passed values.

Ace Editor

The key section of this example is the calling of the three methods at the bottom. Each of these methods are defined under the user object and therefore would usually be inaccessible by premiumUser. However, because user is the prototype of premiumUser, all methods and properties of user are shared with any inheritor objects.

From the final two methods, we also see how the value of the properties shared are not stagnant but can be set to new values regardless of the properties’ values in the prototype.

**Example Code 3: Three-Tier Inheritance and Scalability**

As you might have noticed, the examples above allow for only one account in user and one account in premiumUser. To introduce much needed scalability, we pivot from using these objects as variables and instead use them as an equivalent to classes.

Instead of changing properties’ values, we create new objects for each account, setting the prototype for that new object based on the tier of the account.

In the example below, the object me will be my account. This object then calls the inherited setter method to set values for the email and IDnumber property exclusive to this account, and set its tier by making the newly added familyPremium object as its prototype.

While this is an example using a single account object, this procedure could be scaled to assign the correct properties to any number of objects.

Ace Editor

Even with three levels of inheritance, we can see that me has access to data throughout the chain, from the immediately inherited multipleDevices property to the inherited accountInfo method, defined at the top of its chain in user.

Regardless of how many levels the inheritance chain has, all information from previous levels are retained and accessible.

Through the use of prototypical inheritance, we’re able to create a program in which new accounts can be added and assigned established properties in only a few lines of code rather than having to set manually.

It also allows for ease of adaptability of those properties. If we could change properties of all inheritor accounts by only changing the properties in the prototype.

Prototype chaining is a fundamental concept in JavaScript, playing a crucial role in how objects inherit properties and methods. Let's dive into the basics and its significance.

**Object Creation in JavaScript**

In JavaScript, objects can be created in two primary ways:

1. **Using Constructor Functions**
2. **Using the** Object.create Method

**Constructor Functions**

Constructor functions are regular functions but are used differently when called with the new keyword. When a constructor function is invoked, it creates an empty object, and within this function, the this keyword refers to this new object. After the function finishes execution, it returns this object.

**Example:**

function Person(name) {

this.name = name;

}

const person1 = new Person('Alice');

console.log(person1.name); // Output: Alice

**Prototype Chaining**

In JavaScript, every object has a prototype, which can be another object or null. When you try to access a property or method on an object, JavaScript first looks at the object itself. If it doesn't find it there, it looks at the object's prototype, and this process continues up the prototype chain until it reaches null.

**Constructor Function Prototypes**

Every constructor function has a prototype property, which is an object. This prototype object becomes the prototype of the objects created by the constructor function.

**Example:**

function Person(name) {

this.name = name;

}

Person.prototype.greet = function() {

console.log(`Hello, my name is ${this.name}`);

};

const person1 = new Person('Alice');

person1.greet(); // Output: Hello, my name is Alice

In this example, the greet method is defined on Person.prototype. When greet is called on person1, JavaScript looks up the prototype chain and finds the method on Person.prototype.

**Functions and Arrays**

Functions and arrays in JavaScript are also objects, and they are created using their respective constructor functions. The prototype chaining mechanism applies to them as well. For instance, arrays have Array.prototype and functions have Function.prototype, which are used for method inheritance.

**Example:**

const arr = [1, 2, 3];

console.log(arr.hasOwnProperty('length')); // Output: true

console.log(Array.prototype.hasOwnProperty('push')); // Output: true

function greet() {

console.log('Hello');

}

console.log(greet.hasOwnProperty('call')); // Output: false

console.log(Function.prototype.hasOwnProperty('call')); // Output: true

**Benefits of Prototype Chaining**

The primary benefit of prototype chaining is **memory efficiency**. Methods are not duplicated across each instance; instead, they are defined once on the prototype and shared by all instances, saving memory.

**Setting Prototype to Null**

If you set the prototype of a constructor function to null, the created objects will have Object.prototype as their prototype.

**Object.create Method**

The Object.create method creates a new object with the specified prototype object or null.

**Example:**

const personPrototype = {

greet: function() {

console.log('Hello!');

}

};

const person1 = Object.create(personPrototype);

person1.greet(); // Output: Hello!

**ES6 Classes**

Behind the scenes, ES6 classes use the same prototype mechanism as constructor functions, making them syntactical sugar over the traditional constructor function and prototype pattern.

**A Whole New Object**

In JavaScript, objects are pairs of keys and values (in Ruby, this structure is called a Hash; in Python, it's called a dictionary). For example, if I wanted to describe my name, I could have an object with two keys: firstName would point to "Yehuda" and lastName would point to "Katz". Keys in a JavaScript object are Strings.

To create the simplest new object in JavaScript, you can use Object.create:

var person = Object.create(null); // this creates an empty objects

Why didn't we just use var person = {};? Stick with me! To look up a value in the object by key, use bracket notation. If there is no value for the key in question, JavaScript will return undefined.

person['name'] // undefined

If the String is a valid identifier[1], you can use the dot form:

person.name // undefined

*[1] in general, an* [*identifier*](http://es5.github.com/?ref=yehudakatz.com#x7.6) *starts with a unicode letter, $, \_, followed by any of the starting characters or numbers. A valid identifier must also not be a* [*reserved word*](http://es5.github.com/?ref=yehudakatz.com#x7.6.1)*. There are other allowed characters, such as unicode combining marks, unicode connecting punctuation, and unicode escape sequences. Check out the spec for the full details*

**Adding values**

So now you have an empty object. Not that useful, eh? Before we can add some properties, we need to understand what a property (what the spec calls a "named data property") looks like in JavaScript.

Obviously, a property has a name and a value. In addition, a property can be **enumerable**, **configurable** and **writable**. If a value is enumerable, it will show up when enumerating over an object using a for(prop in obj) loop. If a property is writable, you can replace it. If a property is configurable, you can delete it or change its other attributes.

In general, when we create a new property, we will want it to be enumerable, configurable, and writable. In fact, prior to ECMAScript 5, that was the only kind of property a user could create directly.

We can add a property to an object using Object.defineProperty. Let's add a first name and last name to our empty object:

var person = Object.create(null);

Object.defineProperty(person, 'firstName', {

value: "Yehuda",

writable: true,

enumerable: true,

configurable: true

});

Object.defineProperty(person, 'lastName', {

value: "Katz",

writable: true,

enumerable: true,

configurable: true

});

Obviously, this is extremely verbose. We can make it a bit less verbose by eliminating the common defaults:

var config = {

writable: true,

enumerable: true,

configurable: true

};

var defineProperty = function(obj, name, value) {

config.value = value;

Object.defineProperty(obj, name, config);

}

var person = Object.create(null);

defineProperty(person, 'firstName', "Yehuda");

defineProperty(person, 'lastName', "Katz");

Still, this is pretty ugly to create a simple property list. Before we can get to a prettier solution, we will need to add another weapon to our JavaScript object arsenal.

**Prototypes**

So far, we've talked about objects as simple pairs of keys and values. In fact, JavaScript objects also have one additional attribute: a pointer to *another* object. We call this pointer the object's *prototype*. If you try to look up a key on an object and it is not found, JavaScript will look for it in the prototype. It will follow the "prototype chain" until it sees a null value. In that case, it returns undefined.

You'll recall that we created a new object by invoking Object.create(null). The parameter tells JavaScript what it should set as the Object's *prototype*. You can look up an object's prototype by using Object.getPrototypeOf:

var man = Object.create(null);

defineProperty(man, 'sex', "male");

var yehuda = Object.create(man);

defineProperty(yehuda, 'firstName', "Yehuda");

defineProperty(yehuda, 'lastName', "Katz");

yehuda.sex // "male"

yehuda.firstName // "Yehuda"

yehuda.lastName // "Katz"

Object.getPrototypeOf(yehuda) // returns the man object

We can also add functions that we share across many objects this way:

var person = Object.create(null);

defineProperty(person, 'fullName', function() {

return this.firstName + ' ' + this.lastName;

});

// this time, let's make man's prototype person, so all

// men share the fullName function

var man = Object.create(person);

defineProperty(man, 'sex', "male");

var yehuda = Object.create(man);

defineProperty(yehuda, 'firstName', "Yehuda");

defineProperty(yehuda, 'lastName', "Katz");

yehuda.sex // "male"

yehuda.fullName() // "Yehuda Katz"

**Setting Properties**

Since creating a new writable, configurable, enumerable property is pretty common, JavaScript makes it easy to do so using assignment syntax. Let's update the previous example using assignment instead of defineProperty:

var person = Object.create(null);

// instead of using defineProperty and specifying writable,

// configurable, and enumerable, we can just assign the

// value directly and JavaScript will take care of the rest

person['fullName'] = function() {

return this.firstName + ' ' + this.lastName;

};

// this time, let's make man's prototype person, so all

// men share the fullName function

var man = Object.create(person);

man['sex'] = "male";

var yehuda = Object.create(man);

yehuda['firstName'] = "Yehuda";

yehuda['lastName'] = "Katz";

yehuda.sex // "male"

yehuda.fullName() // "Yehuda Katz"

Just like when looking up properties, if the property you are defining is an *identifier*, you can use dot syntax instead of bracket syntax. For instance, you could say man.sex = "male" in the example above.

**Object Literals**

Still, having to set a number of properties every time can get annoying. JavaScript provides a literal syntax for creating an object and assigning properties to it at one time.

var person = { firstName: "Paul", lastName: "Irish" }

This syntax is approximately sugar for:

var person = Object.create(Object.prototype);

person.firstName = "Paul";

person.lastName = "Irish";

The most important thing about the expanded form is that object literals *always* set the newly created object's prototype to an object located at Object.prototype. Internally, the object literal looks like this:

The default Object.prototype dictionary comes with a number of the methods we have come to expect objects to contain, and through the magic of the prototype chain, all new objects created as object literal will contain these properties. Of course, objects can happily override them by defining the properties directly. Most commonly, developers will override the toString method:

var alex = { firstName: "Alex", lastName: "Russell" };

alex.toString() // "[object Object]"

var brendan = {

firstName: "Brendan",

lastName: "Eich",

toString: function() { return "Brendan Eich"; }

};

brendan.toString() // "Brendan Eich"

This is especially useful because a number of internal coercion operations use a supplied toString method.

Unfortunately, this literal syntax only works if we are willing to make the new object's prototype Object.prototype. This eliminates the benefits we saw earlier of sharing properties using the prototype. In many cases, the convenience of the simple object literal outweighs this loss. In other cases, you will want a simple way to create a new object with a particular prototype. I'll explain it right afterward:

var fromPrototype = function(prototype, object) {

var newObject = Object.create(prototype);

for (var prop in object) {

if (object.hasOwnProperty(prop)) {

newObject[prop] = object[prop];

}

}

return newObject;

};

var person = {

toString: function() {

return this.firstName + ' ' + this.lastName;

}

};

var man = fromPrototype(person, {

sex: "male"

});

var jeremy = fromPrototype(man, {

firstName: "Jeremy",

lastName: "Ashkenas"

});

jeremy.sex // "male"

jeremy.toString() // "Jeremy Ashkenas"

Let's deconstruct the fromPrototype method. The goal of this method is to create a new object with a set of properties, but with a particular prototype. First, we will use Object.create() to create a new empty object, and assign the prototype we specify. Next, we will enumerate all of the properties in the object that we supplied, and copy them over to the new object.

Remember that when you create an object literal, like the ones we are passing in to fromPrototype, it will always have Object.prototype as its prototype. By default, the properties that JavaScript includes on Object.prototype are *not enumerable*, so we don't have to worry about valueOf showing up in our loop. However, since Object.prototype is an object like any other object, anyone can define a new property on it, which may (and probably would) be marked enumerable.

As a result, while we are looping through the properties on the object we passed in, we want to restrict our copying to properties that were defined on the object itself, and not found on the prototype. JavaScript includes a method called hasOwnProperty on Object.prototype to check whether a property was defined on the object itself. Since object literals will always have Object.prototype as their prototype, you can use it in this manner.

The object we created in the example above looks like this:

**Native Object Orientation**

At this point, it should be obvious that prototypes can be used to inherit functionality, much like traditional object oriented languages. To facilitate using it in this manner, JavaScript provides a new operator.

In order to facilitate object oriented programming, JavaScript allows you to use a Function object as a combination of a prototype to use for the new object and a constructor function to invoke:

var Person = function(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

Person.prototype = {

toString: function() { return this.firstName + ' ' + this.lastName; }

}

Here, we have a single Function object that is both a constructor function and an object to use as the prototype of new objects. Let's implement a function that will create new instances from this Person object:

function newObject(func) {

// get an Array of all the arguments except the first one

var args = Array.prototype.slice.call(arguments, 1);

// create a new object with its prototype assigned to func.prototype

var object = Object.create(func.prototype);

// invoke the constructor, passing the new object as 'this'

// and the rest of the arguments as the arguments

func.apply(object, args);

// return the new object

return object;

}

var brendan = newObject(Person, "Brendan", "Eich");

brendan.toString() // "Brendan Eich"

The new operator in JavaScript essentially does this work, providing a syntax familiar to those comfortable with traditional object oriented languages:

var mark = new Person("Mark", "Miller");

mark.toString() // "Mark Miller"

In essence, a JavaScript "class" is just a Function object that serves as a constructor plus an attached prototype object. I mentioned before that earlier versions of JavaScript did not have Object.create. Since it is so useful, people often created something like it using the new operator:

var createObject = function (o) {

// we only want the prototype part of the `new`

// behavior, so make an empty constructor

function F() {}

// set the function's `prototype` property to the

// object that we want the new object's prototype

// to be.

F.prototype = o;

// use the `new` operator. We will get a new

// object whose prototype is o, and we will

// invoke the empty function, which does nothing.

return new F();

};

From ECMA Spec:

In a class-based object-oriented language, in general, state is carried by instances, methods are carried by classes, and inheritance is only of structure and behaviour. In ECMAScript, the state and methods are carried by objects, while structure, behaviour, and state are all inherited.

Let’s see below example. Instances created by new Mold() inherits the property from its prototype. You can see different behaviors if we change different property in different levels.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

function Mold() {

}

Mold.prototype.sizes = [10, 20, 30];

Mold.prototype.brand = 'KC';

var instance = new Mold();

var instance2 = new Mold();

console.log(instance.sizes) // => [10, 20, 30];

console.log(instance2.brand) // => 'KC';

// Adds property with same name shadows the one in prototype

instance.brand = 'K & W';

console.log(instance.brand); // => 'K & W'

console.log(Mold.prototype.brand); // => 'KC'

// Changed on prototype affects instance that doesn't have property with same name

Mold.prototype.brand = 'K & W & L'

console.log(instance.brand); // => 'K & W'

console.log(instance2.brand); // => 'K & W & L'

// Changed on property which is Array or Object type in prototoype affects all

instance.sizes[0] = 40;

Mold.prototype.sizes[2] = 60;

console.log(Mold.prototype.sizes); // => [40, 20, 60]

console.log(instance2.sizes) // => [40, 20, 60]

I think most of the JSer know the concept & behavior above. However, do you know the relationship between \_\_proto\_\_, prototype and constructor?

Here is what the ECMAScript Spec said.

What is prototype chain:

Every object created by a constructor has an implicit reference (called the object’s prototype) to the value of its constructor’s “prototype” property. Furthermore, a prototype may have a non-null implicit reference to its prototype, and so on; this is called the prototype chain. When a reference is made to a property in an object, that reference is to the property of that name in the first object in the prototype chain that contains a property of that name. In other words, first the object mentioned directly is examined for such a property; if that object contains the named property, that is the property to which the reference refers; if that object does not contain the named property, the prototype for that object is examined next; and so on.

Explanation on Object and Function prototype object:

The Object prototype object is the intrinsic object %ObjectPrototype%. The Object prototype object is an ordinary object.

The value of the [[Prototype]] internal slot of the Object prototype object is null

…

The Function prototype object is the intrinsic object %FunctionPrototype%. The Function prototype object is itself a built-in function object. When invoked, it accepts any arguments and returns undefined. It does not have a [[Construct]] internal method so it is not a constructor.

NOTE The Function prototype object is specified to be a function object to ensure compatibility with ECMAScript code that was created prior to the ECMAScript 2015 specification.

The value of the [[Prototype]] internal slot of the Function prototype object is the intrinsic object %ObjectPrototype% (19.1.3).

The Function prototype object does not have a prototype property.

The value of the length property of the Function prototype object is 0.

The value of the name property of the Function prototype object is the empty String

Honestly, for me, it took me a long long time to try to understand that. I have extract some important concepts here and try to be brief:

Function instances that can be used as a constructor have a prototype property. When we use new with this function to create instances, the object pointed by the prototype property will be assigned as the prototype of the instances.

\_\_proto\_\_ is the actual object that is used in the property lookup chain. All objects have this property.

constructor is a function object that creates and initializes objects. It’s a property in the Prototype Object;

Maybe the wording is still not as expressive as the graph as below.

Firefox script not responsive warning

At first glance, this diagram might seems messy. Let’s examine it piece by piece from the easiest part and pay attention to some important points.

prototype property doesn’t exists in instances created by Mold. Only function has.

Each function’s prototype has a constructor property points to itself.

\_\_proto\_\_ points to the prototype of its constructor. The Mold instance points to Mold function, the Mold function points to Function function, the Function function points to itself and Object function points to Function function.

prototype property in Object doesn’t have \_\_proto\_\_.

\_\_proto\_\_ of the Function prototype object is Object.prototype.

Here is some script for you to understand it.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

// 1

typeof instance instance.prototype // => 'undefined'

// 2

Mold.prototype.constructor === Mold

Function.prototype.constructor === Function

Object.prototype.constructor === Object

// 3

instance.\_\_proto\_\_ === Mold.prototype

Mold.\_\_proto\_\_ === Function.prototype

Object.\_\_proto\_\_ === Function.prototype

Function.\_\_proto\_\_ === Function.prototype

// 4

Object.prototype.\_\_proto\_\_ === null

// 5

Function.prototype.\_\_proto\_\_ === Object.prototype

Mold.prototype.\_\_proto\_\_ === Object.prototype

How about take some exams to test see if you understand what I and ECMA spec said above? :)

1

2

3

4

5

6

7

8

9

typeof Function.prototype === ?

typeof Function.prototype.prototype === ?

Function.prototype.length === ?

instance.constructor === ?

instance.\_\_proto\_\_.constructor === ?

Mold.constructor === ?

Mold.\_\_proto\_\_.constructor === ?