**Object Prototypes:**

From ChatGPT: In JavaScript, objects can be created using constructor functions. This is one of the three ways we can create objects that we discussed above. A constructor function is a function that is used to create objects with a specific "shape" or structure. The constructor function contains the definition of the properties and methods that the objects will have.

Prototypes are a way to share methods and properties among objects. Each object in JavaScript has a prototype property, which points to an object that provides it with its properties and methods. When a method or property is called on an object, JavaScript looks for that property or method in the object itself, and if it can't find it, it looks for it in the object's prototype (which is an object).

Using prototypes, we can **define methods and properties on a constructor's prototype property instead of defining them in the constructor function itself**. This allows all objects created using that constructor to inherit those methods and properties from the prototype object, without having to define them again for each object. This can save memory and make our code more efficient.

So, in summary, the constructors function in JavaScript can be used to define the blueprint of an object, including any methods it contains, in a single place. However, we can also use prototypes to define methods on the prototype property of the constructor function, which will be inherited by all objects created using that constructor.

There are a few default properties that are available on any JavaScript object:

1. **constructor**: A reference to the constructor function that created the object.
2. **\_\_proto\_\_**: A reference to the prototype object of the object.
3. **toString()**: A method that returns a string representation of the object.
4. **valueOf()**: A method that returns the primitive value of the object.

These default properties exist as part of the JavaScript language specification and are used to provide basic functionality and behavior to all objects in JavaScript.

For example, the **constructor** property is used to identify the constructor function that created an object, which can be useful for determining the type of an object or for creating new objects based on an existing one.

The **\_\_proto\_\_** property is used to access the prototype object of an object, which allows you to access properties and methods that are defined on the prototype. This is the mechanism that JavaScript uses to implement inheritance.

The **toString()** and **valueOf()** methods are used to convert an object to a string or a primitive value, respectively. These methods are used by JavaScript's built-in functions and operators to convert objects to primitive values when necessary.

Overall, these default properties exist to provide a basic set of functionality and behavior to all objects in JavaScript, which allows developers to work with objects in a consistent and predictable way.

**Note:** The property of an object that points to its prototype is **not** called prototype. Its name is not standard, but in practice all browsers use [\_\_proto\_\_](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/proto). *The standard way to access an object's prototype is the [Object.getPrototypeOf(nameOfTheOBject)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getPrototypeOf) method.*

**VVI:** \*\*\*\*\*\*\*When you try to access a property of an object: if the property can't be found in the object itself, the prototype is searched for the property. If the property still can't be found, then the prototype's prototype is searched, and so on until either the property is found, or the end of the chain is reached, in which case undefined is returned.

const myObject = {

  city: "Madrid",

  greet() {

    console.log(`Greetings from ${this.city}`);

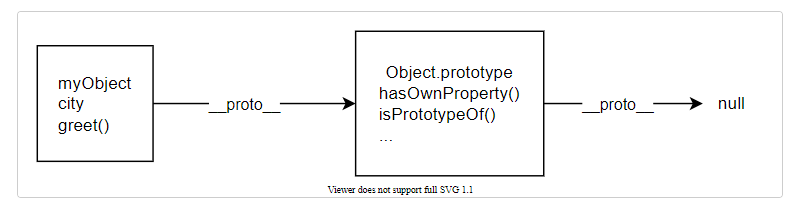
  },

};

myObject.greet(); // Greetings from Madrid

console.log(Object.getPrototypeOf(myObject)); //[Object: null prototype] {}

This is an **object** called **Object.prototype**, and it is the most basic prototype, that all objects have by default. The prototype of Object.prototype object is null, so it's at the end of the prototype chain:



The prototype of an object is **not always** Object.prototype. Try this:

const myDate = new Date();

let object = myDate;

do {

object = Object.getPrototypeOf(object);

console.log(object);

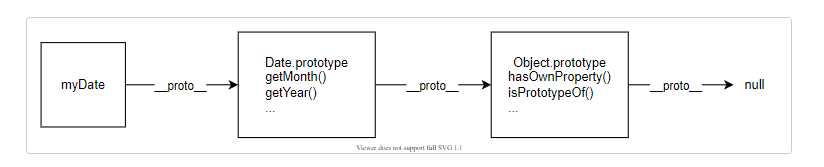
} while (object);

// iteration 1: Date.prototype

// iteration 2: Object { }

// iteration 3: null

This code creates a Date object, then walks up the prototype chain, logging the prototypes. It shows us that the prototype of myDate is a Date.prototype object, and the prototype of *that* is Object.prototype.



In fact, when you call familiar methods, like myDate2.getMonth(), you are calling a method that's defined on Date.prototype.

Classical OOP in JS: It's worth keeping in mind that the features described here are not a new way of combining objects: under the hood, **they still use prototypes. T**hey're just a way to make it easier to set up a prototype chain.

Q: In Javascript summarize the Object oriented programming concept and how OOP is different between class-based OOP and Prototype-based OOP.

**Response 1:** Object-oriented programming (OOP) is a programming paradigm that allows you to organize your code into objects that interact with each other. The fundamental concepts of OOP are encapsulation, inheritance, and polymorphism.

In JavaScript, there are two types of OOP: class-based and prototype-based.

Class-based OOP in JavaScript was introduced in ECMAScript 2015 (ES6) and is similar to OOP in other languages such as Java or C++. In class-based OOP, you define a blueprint for an object, which is called a class. A class contains properties and methods that define the behavior of the object. You can create new objects from a class, which are called instances.

**Response 2:** Prototype-based OOP involves creating objects directly from prototypes, which are essentially blueprints for creating objects.

**An object's prototype can be accessed using the \_\_proto\_\_ property**. When a property or method is called on an object, JavaScript looks for it in the object itself first, and then in its prototype chain until it reaches the root object, **Object.prototype**.

Prototype-based OOP typically involves the use of keywords such as "Object.create," "Object.getPrototypeOf," and "Object.setPrototypeOf."

\_proto\_ = **\_\_proto\_\_** Used in object created using literal. The **\_\_proto\_\_** property is used to access the prototype object **of an object**, which allows you to access properties and methods that are defined on the prototype. This is the mechanism that JavaScript uses to implement inheritance

Object.prototype.Prop\_name = to declare a property on a prototype of a function/constructor function so the object that are created from it can inherit the method. Eg: Functionname.prototype.propName.

**Object.prototype.\_\_proto\_\_** and **\_\_proto\_\_** refer to the same internal property of an object, which is its prototype. **Object.prototype.\_\_proto\_\_** is a reference to the prototype of the **Object.prototype** object itself.

console.log(Object.prototype.\_\_proto\_\_ === null); // true, because it is the end of the prototype chain.

In the above code, we access the **\_\_proto\_\_** property of the **Object.prototype** object, which gives us a reference to its prototype. Since **Object.prototype** is the top-level object in the prototype chain, its prototype is **null**.

It's worth noting that the **\_\_proto\_\_** property is considered somewhat of a legacy feature and should be avoided in favor of the **Object.getPrototypeOf()** method. The **Object.getPrototypeOf()** method allows you to get the prototype of an object without relying on the **\_\_proto\_\_** property, which can sometimes be inconsistent across different JavaScript environments.

**[[prototype]]** = **[[Prototype]]** is an internal property that is used to implement the prototype-based inheritance mechanism. By using double square brackets, the designers of the language ensured that the **[[Prototype]]** property would be reserved for internal use only and would not be accidentally overwritten by user code. This notation is also consistent with other internal properties in JavaScript, such as **[[Class]]**, **[[Get]]**, and **[[Set]]**, which also use double square brackets.

**this** keyword refers to the current instance of a class. It is used to access instance variables and methods within a class's methods.

In Prototype-based OOP, instead of using classes, we create objects and then use them as prototypes for other objects.

* **Instance variable:** An instance variable is a variable that belongs to an instance of a class. In JavaScript, we can define instance variables by using the "this" keyword inside the constructor function.
* **Class variable:** A class variable is a variable that belongs to the class itself rather than any instance of the class. In JavaScript, we can define class variables by defining them as static fields of the class. Static field and Static method are class variables.
* **super**: The "super" keyword is used to call a parent class's constructor or method from within a subclass. It is also used in **prototype-based OOP.**
* const obj1 = {
* method1() {
* console.log("method 1");
* },
* };
* const obj2 = {
* method2() {
* super.method1();
* },
* };
* Object.setPrototypeOf(obj2, obj1);
* obj2.method2(); // Logs "method 1"
* **In prototype-based OOP**, there are **no static** /class fields and methods since there are no classes.

Q: In class-based OOP using Javascript, why can't we access super class's **variables** using the keyword 'super' from inside the subclass but we can do so for the superclass's **methods**?

Response 1: Because the methods are defined on the prototype and variables are defined on the instance (inside the constructor).

In JavaScript, the **super** keyword is used to access and call methods on an object's parent or superclass. When used inside a subclass, **super** can be used to call methods defined in the superclass. However, it cannot be used to directly access the superclass's variables.

This is because in JavaScript, **class-based inheritance is implemented using prototype-based inheritance**. When a subclass is created, a new object is created that inherits from the superclass's prototype. The **properties and methods defined in the superclass are added to the prototype of the subclass.**

When you access a method using **super**, the JavaScript engine looks up the prototype chain and calls the method on the superclass's prototype. However, when you try to access a variable using **super**, there is no equivalent lookup mechanism in JavaScript. **Variables are stored on the instance itself, not on the prototype.**

Therefore, if you want to access a superclass variable from a subclass, you will need to define a getter method on the superclass and call that method using **super**. This method can then return the value of the variable, allowing you to access it from the subclass.

I like the above response. Basically, variables are always stored on instance of a class. The prototype gives you an access to various methods/functions to manipulate the data but prototype does not store the data. Also super.superclassVariable is not allowed because I am trying to access data of the parent class, which is not allowed. That would go against encapsulation. You have access to superclass’s methods but you are not going to access the variable/data directly. An instance of a class, whether subclass or superclass, has access to the data/variable of its own but not to any other instance (subclass/superclass).

**Class variable and Static variable mean same thing.**

Q: In javascript, what is prototype in relation to instance variable, constructor, and a class?

A **constructor function** is a special function used to create objects with the same structure, and it is called with the new keyword. When a constructor function is called with the new keyword, it creates a new object that inherits from the constructor's prototype.

A constructor is a special function that creates and initializes an object created **from a class**. When a new object is created using the "new" keyword and a constructor function, the new object is said to be an instance of that class. The constructor function can also be used to set instance variables.

The **prototype property** is a property of a constructor function that **specifies the prototype object for objects created by that constructor.** The **prototype object contains properties and methods that are shared by all instances of the constructor.**

An instance variable is a property of an object that is specific to that object. It can be created inside the constructor function or added to the object later. Instance variables are unique to each instance of the object.

A class in JavaScript is simply a **constructor function** with the addition of a class declaration and optional static and public fields.

Here's an example to illustrate this:

// Constructor function for creating Person objects

function Person(name, age) {

  this.name = name;

  this.age = age;

}

// Adding a method to the Person prototype

Person.prototype.sayHello = function() {

  console.log("Hello, my name is " + this.name + " and I am " + this.age + " years old.");

};

// Creating a new Person instance

var person1 = new Person("John", 30);

// Calling the sayHello() method on the person1 instance

person1.sayHello(); // Output: "Hello, my name is John and I am 30 years old."

In the example above, we define a constructor function for creating Person objects. We then add a **sayHello** method to the **Person.prototype** object, which is inherited by all instances of the **Person** object. We create a new instance of **Person** called **person1** and call the **sayHello** method on it, which outputs a message to the console.

So in summary, the prototype property in JavaScript is used to define properties and methods that are inherited by all instances of an object.

# Inheritance and the prototype chain

Although classes are now widely adopted and have become a new paradigm in JavaScript, classes do not bring a new inheritance pattern.

obj.[[Prototype]]  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.

 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.

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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#defining_getters_and_setters).

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

### [Inheriting "methods"](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#inheriting_methods):

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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) points to the inheriting object, not to the prototype object where the function is an **own property**.

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.

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.

This statement is explaining two different concepts related to JavaScript constructors and prototypes:

1. **Constructor.prototype**: This property is used to add properties and methods to objects created with the constructor function. It is only useful when constructing instances of the object. For example, if you have a constructor function called **Person**, you can add a method to all instances of **Person** by defining **Person.prototype.methodName = function() { /\* method code \*/ }**.
2. **Constructor.[[Prototype]]**: This refers to the prototype of the constructor function itself. In other words, it is the object from which the constructor inherits its properties and methods. By default, the prototype of a constructor function is **Function.prototype**, which is the built-in prototype for all functions in JavaScript.

So, to summarize the statement:

* **Constructor.prototype** is only useful when constructing instances of an object.
* **Constructor.[[Prototype]]** is the prototype of the constructor function itself, and it is **Function.prototype** by default.

### [Implicit constructors of literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#implicit_constructors_of_literals)

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

// Object literals (without the `\_\_proto\_\_` key) automatically

// have `Object.prototype` as their `[[Prototype]]`

const object = { a: 1 };

Object.getPrototypeOf(object) === Object.prototype; // true

monkey patching: An example of this misfeature is, defining Array.prototype.myMethod = function () {...} and then using myMethod on all array instances.

## [Inspecting prototypes: a deeper dive](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#inspecting_prototypes_a_deeper_dive)

In JavaScript, as mentioned above, functions are able to have properties. All functions have a special property named prototype.

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.

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

function doSomething() {}

doSomething.prototype.foo = "bar";

console.log(doSomething.prototype);

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**.

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:

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 (why??), because ‘foo’ is defined on doSomething.prototype

doSomething.prototype.prop: undefined

doSomething.prototype.foo: bar

Why because: \*\*\*\*

The **doSomething** function itself doesn't have an "own" property called **foo**, so JavaScript looks up the prototype chain.

The **doSomething.prototype** object has an "own" property called **foo** with a value of **"bar"** but this object is the prototype for the **instance** of the **doSomething** function. The prototype for the **doSomething is ‘Object.prototype’. So, ‘foo’ is searched first on the function itself, it does not find the foo there, then the interpretor looks for ‘foo’ in ‘Object.prototype’, it does not find there either, hence the result is ‘undefined’**

In JavaScript, functions are objects, and like any other object, they can have properties. However, when you define a property on a function, it is not automatically added to instances of that function. Instead, you need to add the property to the function's **prototype** object.

‘foo’ was added to the **doSomething.prototype object so that all instance of doSomething would inherit the ‘foo’ via their prototype which is ‘doSomething.prototype’.**

**doSomething is just a regular function, so its prototype object is the default value (Object.prototype).**