<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics>

**Object:**

An object is a collection of related data and/or functionality. These usually consist of several variables and functions (which are called properties and methods when they are inside objects).

**Creating Object using Object Literal:**

const objectName = {

member1Name: member1Value,

member2Name: member2Value,

member3Name: member3Value,

};

The value of an object member can be pretty much anything. If the member is a function, it allows the object to do something with that data, and are referred to as the object's **methods**.

When the object's members are functions there's a simpler syntax. Instead of bio: function () we can write bio().

const person = {

name: ["Bob", "Smith"],

age: 32,

bio() {

console.log(`${this.name[0]} ${this.name[1]} is ${this.age} years old.`);

},

};

An object like this is referred to as an **object literal** — we've literally written out the object contents as we've come to create it. This is different compared to objects instantiated from classes, which we'll look at later on.

It is very common to create an object **using an object literal** when you want to **transfer a series of structured, related data items in some manner**, for example **sending a request to the server to be put into a database**. Sending a single object is much more efficient than sending several items individually, and it is easier to work with than an array, when you want to identify individual items by name.

An object property can itself be an object:

const person = {

name: ["Bob", "Smith"],

};

Change the above object to:

const person = {

name: {

first: "Bob",

last: "Smith",

},

// …

};

To access these items you just need to chain the extra step onto the end with another dot.

person.name.first;

person.name.last;

Objects are sometimes called **associative arrays** — they map strings to values in the same way that arrays map numbers to values.

**Bracket notation and Dot notation to access object’s properties and methods:**

You can also **set** (update) the value of object members by declaring the member you want to set (using dot or bracket notation), like this:

person.age = 45; //dot notation –uses literal member name.

person["eyes"] = "hazel"; //bracket notation.

One useful aspect of **bracket notation** is that it can be used to set not only member values dynamically, but member names too.

Example:

const myDataName = "height";

const myDataValue = "1.75m";

person[myDataName] = myDataValue; // we added a height property whose value is 1.75m to the person object.

Adding a property to an object using the method above isn't possible with **dot notation**, which can only accept a literal member name, not a variable value pointing to a name.

**Creating Object using a Function:**

Using object literals is fine when you only need to create one object, but if you have to create more than one, they're seriously inadequate. We have to write out the same code for every object we create, and if we want to change some properties of the object - like adding a height property - then we have to remember to update every object.

The first version of this is just a **function:**

function createPerson(name) {

**const obj = {};**

obj.name = name;

obj.introduceSelf = function () {

console.log(`Hi! I'm ${this.name}.`);

};

return obj;

}

This function creates and returns a new object each time we call it. The object will have two members:

* a property name
* a method introduceSelf().

Note that createPerson() takes a parameter name to set the value of the name property, but the value of the introduceSelf() method will be the same for all objects created using this function. This is a very common pattern for creating objects.

Now we can create as many objects as we like, reusing the definition:

const salva = createPerson("Salva");

const frankie = createPerson("Frankie");

This works fine but is a bit long-winded: **we have to create an empty object, initialize it, and return it**. A better way is to use a **constructor**.

**Creating Object using a Constructor function:**

A constructor is just a function called using the [new](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) keyword. When you call a constructor, it will:

* create a **new** object
* bind this to the new object, so you can refer to this in your constructor code
* run the code in the constructor
* return the new object.

**Constructors,** by convention, start with a **capital letter** and are named for the type of object they create. So we could rewrite our example like this:

function **P**erson(name) {

this.name = name;

this.introduceSelf = function () {

console.log(`Hi! I'm ${this.name}.`);

};

}

To call Person() as a constructor, we use **new**:

const salva = new Person("Salva");

salva.name;

salva.introduceSelf(); // "Hi! I'm Salva."

**‘new’ Operator:**

new constructor is same as new constructor() since both don’t have argument(s).

When a function is called with the **new** keyword, the function will be used as a constructor. new will do the following things:

1. Creates a blank, plain JavaScript object. For convenience, let's call it newInstance.
2. Points newInstance's [[Prototype]] to the constructor function's prototype property, if the prototype is an [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object). Otherwise, newInstance stays as a plain object with Object.prototype as its [[Prototype]].

**Note:** Properties/objects added to the constructor function's prototype property are therefore accessible to all instances created from the constructor function.

1. Executes the constructor function with the given arguments, binding newInstance as the [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) context (i.e. all references to this in the constructor function now refer to newInstance).
2. If the constructor function returns a [non-primitive](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#primitive_values), this return value becomes the result of the whole new expression. Otherwise, if the constructor function doesn't return anything or returns a primitive, newInstance is returned instead. (Normally constructors don't return a value, but they can choose to do so to override the normal object creation process.)

To add the new property to all objects of the same type, you must add the property to the constructor's prototype property.

So far, we have learned 3 ways to create an object:

1. Object literal
2. Function: inside a function - create empty object, initialize it and return it
3. Constructor function: A constructor is just a function called using the [**new**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) keyword

Every time you create a string in your code, that string is automatically created as an instance of String, and therefore has several common methods and properties available on it.

For each webpage loaded, an instance of Document is created, called **document**, which represents the entire page's structure, content, and other features such as its URL. Again, this means that it has several common methods and properties available on it.

The same is true of pretty much any other built-in object or API you've been using — [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array), [Math](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math), and so on.

Note that built in objects and APIs **don't always** create object instances automatically. As an example, the [Notifications API](https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API) — which allows modern browsers to fire system notifications — requires you to instantiate a new object instance using the constructor for each notification you want to fire.

const myNotification = new Notification("Hello!");

**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}`);

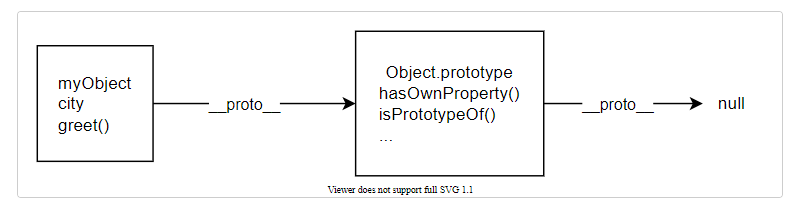
  },

};

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);

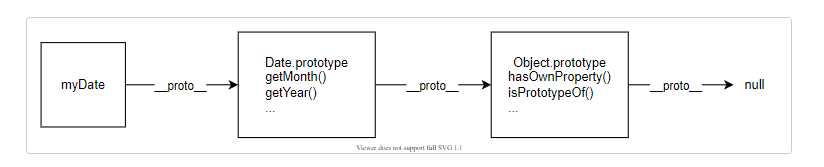
} while (object);

// 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.

## [Shadowing properties](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes#shadowing_properties)

What happens if you define a property in an object, when a property with the same name is defined in the object's prototype? When searched for that property, the object’s own property is picked and not the one from the prototype. This is called **Property Shadowing**.

## [**Setting a prototype**](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes#setting_a_prototype)**:**

There are various ways of setting an object's prototype in JavaScript, and here we'll describe two: Object.create() and constructors.

### [Using Object.create](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes#using_object.create)

The Object.create() method creates a new object and allows you to specify an object that will be used as the new object's prototype.

Here's an example:

const personPrototype = {

greet() {

console.log("hello!");

},

};

const carl = Object.create(personPrototype); //create ‘carl’ object and make personPrototype object its prototype.

carl.greet(); // hello!

Here we create an object personPrototype, which has a greet() method. We then use Object.create() to create a new object with personPrototype as its prototype. Now we can call greet() on the new object, and the prototype provides its implementation.

### [Using a constructor](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes#using_a_constructor)

In JavaScript, all functions have a property named prototype. When you call a function as a constructor, this property is set as the prototype of the newly constructed object (by convention, in the property named \_\_proto\_\_).

*So if we set the prototype of a constructor, we can ensure that all objects created with that constructor are given that prototype*:

const personPrototype = {

  greet() {

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

  },

};

//constructor function below:

function Person(name) {

  this.name = name;

}

Object.assign(Person.prototype, personPrototype);

// or

// Person.prototype.greet = personPrototype.greet;

//Here we create: 1) an object personPrototype, which has a greet() method

//2) a Person() constructor function which initializes the name of the person to create.

//We then put the methods defined in personPrototype onto the Person function's prototype property using Object.assign.

//After this code, objects created using Person() will get Person.prototype as their prototype, which automatically contains the greet method.

const reuben = new Person("Reuben");

reuben.greet(); // hello, my name is Reuben!

This also explains why we said earlier that the prototype of myDate object is called Date.prototype: it's the prototype property of the Date constructor.

### [Own properties](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes#own_properties)

VVI:

The objects we create using the Person constructor above have two properties:

* a name property, which is set in the constructor, so it appears directly on Person objects
* a greet() method, which is set in the prototype.

It's common to see this pattern, in which methods are defined on the prototype, but data properties are defined in the constructor. *That's because methods are usually the same for every object we create*, while we often want each object to have its own value for its data properties (just as here where every person has a different name).

\*\*\*\*\*\*\*\*The above statement is the most important\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Properties that are defined directly in the object, like name here, are called **own properties**, and you can check whether a property is an own property using the static [Object.hasOwn()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn) method

**In JavaScript, there is no explicit concept of static and non-static methods** like in some other object-oriented programming languages such as Java or C#. However, we can achieve similar functionality using JavaScript's prototypical inheritance.

A method defined on an object's prototype is shared among all instances of the object. This is similar to a static method in other languages as it does not belong to a specific instance of the object. On the other hand, a method defined inside an object's constructor or directly on the instance belongs to that specific instance only, and is not shared among other instances. This is similar to a non-static method in other languages.

**JavaScript is a prototype-based language, which means that objects inherit properties and methods from their prototype objects.** In other words, in JavaScript, objects are created from a prototype object, and any changes made to the prototype object will affect all objects that inherit from it.

function Car(make, model, year) {

  this.make = make;

  this.model = model;

  this.year = year;

  // non-static method

  this.getAge = function () {

    return new Date().getFullYear() - this.year;

  };

}

// static method

Car.getInfo = function () {

  return "This is a car object";

};

// prototype method

Car.prototype.startEngine = function () {

  return "Engine started for " + this.make + " " + this.model;

};

// creating objects of Car

let car1 = new Car("Toyota", "Corolla", 2015);

let car2 = new Car("Honda", "Civic", 2018);

console.log(Car.getInfo()); // This is a car object. getInfo() is a static/class function

console.log(car1.startEngine()); // Engine started for Toyota Corolla

console.log(car2.startEngine()); // Engine started for Honda Civic

console.log(car1.getAge()); // 8

console.log(car2.getAge()); // 5

console.log(car2.getInfo()); // Error. getInfo() is not an instance method, nor it exists in this object's prototype. It is a class method defined directly on the Constructor function itself. SO, it can only be accessed using Constructor Function's name. Like this: Car.getInfo()

## [Prototypes and inheritance](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes#prototypes_and_inheritance)

Prototypes are a powerful and very flexible feature of JavaScript, making it possible to reuse code and combine objects.

In particular they support a version of **inheritance**. Inheritance is a feature of object-oriented programming languages that lets programmers express the idea that some objects in a system are more specialized versions of other objects.

You can see how in JavaScript, if Professor and Student objects can have Person prototypes, then they can inherit the common properties, while adding and redefining those properties which need to differ.

## [Different ways of creating and mutating prototype chains](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#different_ways_of_creating_and_mutating_prototype_chains)

**Q: In javascript prototype, why is there double square bracket in '[[Prototype]] '?**

ChatGPT: In JavaScript, the **[[Prototype]] is an internal property** that is used to implement the prototype-based inheritance mechanism. It is a reference to another object that is used as a fallback source of properties.

The reason for the d**ouble square bracket notation is to indicate that this property is an internal property**, which cannot be accessed directly from outside the object. This is known as an **"internal slot"** in the ECMAScript specification, which is a way of describing properties that are used by the language implementation but are **not directly accessible by user code**.

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.

### [**Objects created with syntax constructs**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#objects_created_with_syntax_constructs)

Difference between ‘Object.prototype.\_proto\_’ and ‘\_Proto\_’:

Both ‘Object.prototype.\_\_proto\_\_’ and ‘\_\_proto\_\_’ refer to the same internal property of an object, which is its prototype.

However, there is a key difference between the two:

* **Object.prototype.\_\_proto\_\_** is a getter/setter property that allows you to access the prototype of an object. This property is inherited by all objects from the **Object.prototype** object. Therefore, you can access the prototype of an object using **Object.prototype.\_\_proto\_\_** if it is not defined on the object itself.
* **\_\_proto\_\_** is a **shorthand notation** that allows you to **set the prototype of an object** when you create it using **object literal** notation. This property is **not inherited from any** object prototype and is **only available in object literals**. When you use **\_\_proto\_\_** in an object literal, you are directly defining the prototype of the object being created.
* const o = { a: 1 };
* // The newly created object o has Object.prototype as its [[Prototype]]
* // Object.prototype has null as its prototype.
* // o ---> Object.prototype ---> null
* const p = { b: 2, \_\_proto\_\_: o };
* // It is possible to point the newly created object's [[Prototype]] to
* // another object via the \_\_proto\_\_ literal property. (Not to be confused
* // with Object.prototype.\_\_proto\_\_ accessors)
* // p ---> o ---> Object.prototype ---> null
* const b = ["yo", "whadup", "?"];
* // Arrays inherit from Array.prototype
* // (which has methods indexOf, forEach, etc.)
* // The prototype chain looks like:
* // b ---> Array.prototype ---> Object.prototype ---> null
* function f() {
* return 2;
* }
* // Functions inherit from Function.prototype
* // (which has methods call, bind, etc.)
* // f ---> Function.prototype ---> Object.prototype ---> null

### [**With constructor functions**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#with_constructor_functions)

* function Graph() {
* this.vertices = [];
* this.edges = [];
* }
* Graph.prototype.addVertex = function (v) {
* this.vertices.push(v);
* };
* const g = new Graph();
* // g is an object with own properties 'vertices' and 'edges'.
* // g.[[Prototype]] is the value of Graph.prototype when new Graph() is executed.

Pros: supported by all browser versions.

Cons:

* In order to use this method, the function in question must be initialized. During this initialization, the constructor may store unique information that must be generated per-object. This unique information would only be generated once, potentially leading to problems.
* The initialization of the constructor may put unwanted methods onto the object.

### [**With Object.create()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#with_object.create)

Calling [Object.create()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create) creates a new object. The [[Prototype]] of this object is the first argument of the function:

const a = { a: 1 };

// a ---> Object.prototype ---> null

const b = Object.create(a);

// b ---> a ---> Object.prototype ---> null

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

const c = Object.create(b);

// c ---> b ---> a ---> Object.prototype ---> null

const d = Object.create(null);

// d ---> null (d is an object that has null directly as its prototype)

console.log(d.hasOwnProperty);

// undefined, because d doesn't inherit from Object.prototype

**Pros:**   
Supported in all modern engines. Allows directly setting [[Prototype]] of an object at creation time, which permits the runtime to further optimize the object. Also allows the creation of objects without a prototype, using Object.create(null).

### [**With classes**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#with_classes)

1 class Polygon {

1. constructor(height, width) {
2. this.height = height;
3. this.width = width;
4. }
5. }
6. class Square extends Polygon {
7. constructor(sideLength) {
8. super(sideLength, sideLength);
9. }
10. get area() {
11. return this.height \* this.width;
12. }
13. set sideLength(newLength) {
14. this.height = newLength;
15. this.width = newLength;
16. }
17. }
18. const square = new Square(2);
19. // square ---> Square.prototype ---> Polygon.prototype ---> Object.prototype ---> null

**Pros**: Supported in all modern engines. Very high readability and maintainability. [Private properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Private_class_fields) are a feature with no trivial replacement in prototypical inheritance. This means, we can make our data private using class-based OOP but it can’t be done in prototypical OOP.

**Cons**: Classes, especially with private properties, are less optimized than traditional ones (although engine implementers are working to improve this). Not supported in older environments and transpilers are usually needed to use classes in production.

### [**With Object.setPrototypeOf()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#with_object.setprototypeof)

While all methods above will set the prototype chain at object creation time, [Object.setPrototypeOf()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf) allows mutating the [[Prototype]] internal property of an existing object.

**Pros:** Supported in all modern engines. Allows the dynamic manipulation of an object's prototype and can even force a prototype on a prototype-less object created with Object.create(null).

**Cons**: Ill-performing. **Should be avoided** if it's possible to set the prototype at object creation time. Many engines optimize the prototype and try to guess the location of the method in memory when calling an instance in advance; but setting the prototype dynamically disrupts all those optimizations.

### [**With the \_\_proto\_\_ accessor**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#with_the___proto___accessor)

All objects inherit the [Object.prototype.\_\_proto\_\_](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/proto) setter, which can be used to set the [[Prototype]] of an **existing object** (if the \_\_proto\_\_ key is not overridden on the object).

**Warning:** Object.prototype.\_\_proto\_\_ accessors are **non-standard** and **deprecated**. You should almost always **use Object.setPrototypeOf instead**.

**Cons**: Non-performant and deprecated.

## [**Performance**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#performance)

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

Also, when iterating over the properties of an object, **every** enumerable property that is on the prototype chain will be enumerated. To check whether an object has a property defined on itself and not somewhere on its prototype chain, it is necessary to use the [hasOwnProperty](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty) or [Object.hasOwn](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn) methods. All objects, except those with null as [[Prototype]], inherit [hasOwnProperty](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty) from Object.prototype — unless it has been overridden further down the prototype chain.

**Note**: It is **not** enough to check whether a property is [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined). The property might very well exist, but its value just happens to be set to undefined.

## [**Conclusion**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#conclusion)

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

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

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

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

From the internet:

**Getters and setters** allow you to define Object **Accessors** (Computed Properties).

JavaScript can secure better data quality when using getters and setters.

## **Why Using Getters and Setters?**

* It gives simpler syntax
* It allows equal syntax for properties and methods
* It can secure better data quality
* It is useful for doing things behind-the-scenes

The Object.defineProperty() method can also be used to add Getters and Setters:

# JavaScript Object Constructors: It is considered good practice to name constructor functions with an upper-case first letter.

# In a constructor function (function that is used to create an object) this does not have a value. It is a substitute for the new object. The value of this will become the new object when a new object is created.

# You cannot add a new property to an object constructor the same way you add a new property to an existing object.

# To add a new property to a constructor, you must add it to the constructor function:

# function Person(first, last, age, eyecolor) {   this.firstName = first;   this.lastName = last;   this.age = age;   this.eyeColor = eyecolor;   this.nationality = "English"; }

# This way object properties can have default values.

You **cannot add a new method** to an object constructor the same way you add a new method to an existing object.

Adding methods to an object constructor must be done inside the constructor function:

function Person(firstName, lastName, age, eyeColor) {  
  this.firstName = firstName;   
  this.lastName = lastName;  
  this.age = age;  
  this.eyeColor = eyeColor;  
  **this.changeName = function (name) {  
    this.lastName = name;**  
  };  
**}**

JavaScript knows which person you are talking about by "**substituting**" this with myMother.

## **Built-in JavaScript Constructors**

JavaScript has built-in constructors for native objects:

new String()    // A new String object  
new Number()    // A new Number object  
new Boolean()   // A new Boolean object  
new Object()    // A new Object object  
new Array()     // A new Array object  
new RegExp()    // A new RegExp object  
new Function()  // A new Function object  
new Date()      // A new Date object

The Math() object is not in the list. Math is a global object. The new keyword cannot be used on Math.

There is no reason to create complex objects. Primitive values are much faster:

Use string literals "" instead of new String().

Use number literals 50 instead of new Number().

Use boolean literals true / false instead of new Boolean().

Use object literals {} instead of new Object().

Use array literals [] instead of new Array().

Use pattern literals /()/ instead of new RegExp().

Use function expressions () {} instead of new Function().

# JavaScript Object Prototypes

All JavaScript objects inherit properties and methods from a prototype.

We also learned that you can**not** add a new property to an **existing object constructor:**

To add a new property to a constructor, you must add it to the constructor function:

## **Prototype Inheritance**

All JavaScript objects inherit properties and methods from a prototype:

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

The Object.prototype is on the top of the prototype inheritance chain:

Date objects, Array objects, and Person objects inherit from Object.prototype.

## **Adding Properties and Methods to Object**

Sometimes you want to add new properties (or methods) to all existing objects of a given type.

Sometimes you want to add new properties (or methods) to an object constructor.

## **Using the prototype Property:**

The JavaScript prototype property allows you to add new properties and methods to object constructors:

function Person(first, last, age, eyecolor) {  
  this.firstName = first;  
  this.lastName = last;  
  this.age = age;  
  this.eyeColor = eyecolor;  
}  
  
Person.prototype.nationality = "English";

Person.prototype.name = function() {  
  return this.firstName + " " + this.lastName;  
};

Only modify your **own** prototypes. Never modify the prototypes of standard JavaScript objects.

**Iterable** objects are objects that can be iterated over with for..of.

Technically, iterables must implement the Symbol.iterator method.

**Enumerable**: that can be counted.

# JavaScript Classes

ECMAScript 2015, also known as ES6, introduced JavaScript Classes.

JavaScript Classes are templates for JavaScript Objects.

Use the keyword class to create a class.

Always add a method named constructor():

class Car {  
  constructor(name, year) {  
    this.name = name;  
    this.year = year;  
  }  
}

The example above creates a class named "Car".

The class has two initial properties: "name" and "year".

A JavaScript class is **not** an object.

It is a **template** for JavaScript objects.

The constructor method is called automatically when a new object is created.

## **The Constructor Method**

The constructor method is a special method:

* It has to have the exact name "constructor"
* It is executed automatically when a new object is created
* It is used to initialize object properties

If you do not define a constructor method, JavaScript will add an empty constructor method.

## **Class Methods**

Class methods are created with the same syntax as object methods.

Use the keyword class to create a class.

Always add a constructor() method.

Then add any number of methods.

class ClassName {  
  constructor() { ... }  
  method\_1() { ... }  
}

**Create a Class method named "age", that returns the Car age:**

**You can send parameters to Class methods:**

class Car {  
  constructor(name, year) {  
    this.name = name;  
    this.year = year;  
  }  
  age(x) {  
    return x - this.year;  
  }  
}

## **"use strict"**

The syntax in classes must be written in "strict mode".

In "strict mode" you will get an error if you use a variable without declaring it:

class Car {  
  constructor(name, year) {  
    this.name = name;  
    this.year = year;  
  }  
  age() {  
    // date = new Date();  // This will not work  
    const date = new Date(); // This will work  
    return date.getFullYear() - this.year;  
  }

# JavaScript Class Inheritance

To create a class inheritance, use the extends keyword.

A class created with a class inheritance inherits all the methods from another class:

The super() method refers to the parent class.

By calling the super() method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.

Inheritance is useful for code reusability: reuse properties and methods of an existing class when you create a new class.

Classes also allows you to use getters and setters.

It can be smart to use getters and setters for your properties, especially if you want to do something special with the value before returning them, or before you set them.

To add getters and setters in the class, use the get and set keywords.

**Note:** even if the getter is a method, you do not use parentheses when you want to get the property value.

To use a setter, use the same syntax as when you set a property value, without parentheses:

The name of the getter/setter method **cannot be the same** as the name of the property.

Many programmers use an underscore character \_ before the **property name** to separate the getter/setter from the actual property:

class Car {  
  constructor(brand) {  
    this.\_carname = brand;  
  }  
  get carname() {  
    return this.\_carname;  
  }  
  set carname(x) {  
    this.\_carname = x;  
  }  
}  
  
const myCar = new Car("Ford");

## **Hoisting**

Unlike functions, and other JavaScript declarations, class declarations are not hoisted.

That means that you must declare a class before you can use it:

**Note:** For other declarations, like functions, you will NOT get an error when you try to use it before it is declared, because the default behavior of JavaScript declarations are hoisting (moving the declaration to the top).

# JavaScript Static Methods

Static class methods are defined on the class itself.

You cannot call a static method on an object, only on an object class.

# JavaScript Callbacks

*"I will call back later!"*

A callback is a function passed as an argument to another function

This technique allows a function to call another function

A callback function can run after another function has finished

# Object-oriented programming in General

We'll describe these concepts without reference to JavaScript in particular, so all the examples are given in [pseudocode](https://developer.mozilla.org/en-US/docs/Glossary/Pseudocode). To be precise, the features described here are of a particular style of OOP called **class-based** or "classical" OOP. JavaScript is a prototype-based language, which means that objects inherit properties and methods from their prototype objects.

Object-oriented programming is about modeling a system as a collection of objects, where each object represents some particular aspect of the system. Objects contain both functions (and methods) and data. An object provides a public interface to other code that wants to use it but maintains its own private, internal state; other parts of the system don't have to care about what is going on inside the object.

## [Classes and instances](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_programming#classes_and_instances)

The definition of the class lists the data and methods. The process of creating an instance is performed by a special function called a **constructor**. We pass values to the constructor for any internal state that we want to initialize in the new instance.

Generally, the constructor is written out as part of the class definition, and it usually has the same name as the class itself:

## [Inheritance](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_programming#inheritance)

When a method has the same name but a different implementation in different classes - is called **polymorphism**. When a method in a subclass replaces the superclass's implementation, we say that the subclass **overrides** the version in the superclass.

## [Encapsulation](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_programming#encapsulation)

Objects provide an interface to other code that wants to use them but maintain their own internal state. The object's internal state is kept **private**, meaning that it can only be accessed by the object's own methods, not from other objects.

Keeping an object's internal state private, and generally making a clear division between its public interface and its private internal state, is called **encapsulation**. Encapsulation is achieved by putting data and methods inside a class.

It enables the programmer to change the internal implementation of an object without having to find and update all the code that uses it: it creates a kind of firewall between this object and the rest of the system. The internal implementation of an object is kept private from rest of the system/other objects. So, they have/had no access to the internal implementation. So, any changes made to the internal implementation will have no impact to rest of the system.

In many OOP languages, we can prevent other code from accessing an object's internal state by marking some properties as **private**. This will generate an error if code outside the object tries to access them.

class Student : extends Person

properties

private year

constructor

Student(name, year)

methods

introduceSelf()

canStudyArchery() { return this.year > 1 }

student = new Student('Weber', 1)

student.year // error: 'year' is a private property of Student

In languages that don't enforce access like this, programmers use **naming conventions, such as starting the name with an underscore**, to indicate that the property should be considered private.

## [OOP and JavaScript](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_programming#oop_and_javascript)

JavaScript features: [constructors](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics) and [prototypes](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes) have some relation to some of the OOP concepts described above.

* **constructors** in JavaScript provide us with something like a class definition, enabling us to define the "shape" of an object, including any methods it contains, in a single place. But prototypes can be used here, too. For example, if a method is defined on a constructor's prototype property, then all objects created using that constructor get that method via their prototype, and we don't need to define it in the constructor.
* **the prototype chain** seems like a natural way to implement inheritance. For example, if we can have a Student object whose prototype is Person, then it can inherit name and override introduceSelf().

But it's worth understanding the differences between these features and the "classical" OOP concepts described above. We'll highlight a couple of them here.

1. First, in class-based OOP, classes and objects are **two separate constructs**, and objects are always created as instances of classes. Also, there is a distinction between the feature used to define a class (the class syntax itself) and the feature used to instantiate an object (a constructor).

In JavaScript, we can and often do **create objects** without any separate class definition, either using **a function** or an **object literal**. This can make working with objects much more **lightweight** than it is in classical OOP.

1. Second, although a prototype chain looks like an inheritance hierarchy and behaves like it in some ways, it's different in others.

In classical-OOP, when a subclass is instantiated, a single object is created which combines properties defined in the subclass with properties defined further up the hierarchy.

With prototyping, each level of the hierarchy is represented by a separate object, and they are linked together via the \_\_proto\_\_ property. *The prototype chain's behavior is less like inheritance and more like* **delegation**. Delegation is a programming pattern where an object, when asked to perform a task, can perform the task itself or ask another object (its **delegate**) to perform the task on its behalf. In many ways, delegation is a more flexible way of combining objects than inheritance (for one thing, it's possible to change or completely replace the delegate at run time).

That said, **constructors and prototypes can be used to implement class-based OOP patterns in JavaScript**. But using them directly to implement features like inheritance is tricky, so *JavaScript provides extra features, layered on top of the prototype model, that map more directly to the concepts of class-based OOP. See below.*

# Classes in JavaScript

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.

## [**Classes and constructors**](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Classes_in_JavaScript#classes_and_constructors)in Javascript:

You can declare a class using the [class](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/class) keyword.

class Person {

  name;

  constructor(name) {

    this.name = name;

  }

  introduceSelf() {

    console.log(`Hi! I'm ${this.name}`);

  }

}

This declares a class called Person, with:

* a name property.
* a constructor that takes a name parameter that is used to initialize the new object's name property
* an introduceSelf() method that can refer to the object's properties using this.

The name; declaration is optional: you could omit it, and the line this.name = name; in the constructor will create the name property before initializing it. However, listing properties explicitly in the class declaration might make it easier for people reading your code to see which properties are part of this class.

You could also initialize the property to a default value when you declare it, with a line like name = '';.

The constructor is defined using the [constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor) keyword.

It will:

* create a new object
* bind this to the new object, so you can refer to this in your constructor code
* run the code in the constructor
* Return the new object.

We call the constructor using the name of the class, Person.

const giles = new Person("Giles");

giles.introduceSelf(); // Hi! I'm Giles

### [Omitting constructors](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Classes_in_JavaScript#omitting_constructors)

If you don't need to do any special initialization, you can omit the constructor, and a default constructor will be generated for you.

## [**Inheritance**](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Classes_in_JavaScript#inheritance)in Javascript:

Given our Person class above, let's define the Professor subclass.

The Professor class adds a new property teaches, so we declare that. Since we want to set teaches when a new Professor is created, we define a constructor, which takes the name and teaches as arguments. The first thing this constructor does is call the superclass constructor using [super()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super), passing up the name parameter. The superclass constructor takes care of setting name. After that, the Professor constructor sets the teaches property.

**Note:** If a subclass has any of its own initialization to do, it **must** first call the superclass constructor using super(), passing up any parameters that the superclass constructor is expecting. If there is a constructor present in the subclass, it needs to first call super() before using ‘this’. The [super](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super) keyword can also be used to call corresponding methods of super class.

We've also overridden the introduceSelf() method from the superclass, and added a new method grade(), to grade a paper (our professor isn't very good, and just assigns random grades to papers).

//Given our Person class above, let's define the Professor subclass.

class Person {

  name; //can be ommited but it is easier for other to understand if you keep it.

  constructor(name) {

    this.name = name;

  }

  introduceSelf() {

    console.log(`Hi! I'm ${this.name}`);

  }

}

class Professor extends Person {

  teaches; //can be ommited but it is easier for other to understand if you keep it.

  constructor(name, teaches) {

    super(name); //Calls the constructor of the Super Class. initializes the name.

    this.teaches = teaches; //initializes the 'teaches' property.

  }

  introduceSelf() {

    console.log(

      `My name is ${this.name}, and I will be your ${this.teaches} professor.`

    );

  }

  grade(paper) {

    const grade = Math.floor(Math.random() \* (5 - 1) + 1);

    console.log(grade);

  }

}

//With this declaration we can now create and use professors:

const walsh = new Professor("Walsh", "Psychology");

walsh.introduceSelf(); // 'My name is Walsh, and I will be your Psychology professor'

walsh.grade("my paper"); // some random grade

## [**Encapsulation**](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Classes_in_JavaScript#encapsulation) in Javascript:

We would like to make the year property of Student private, so we could change the rules about archery classes without breaking any code that uses the Student class.

class Person {

  name; //can be ommited but it is easier for other to understand if you keep it.

  constructor(name) {

    this.name = name;

  }

  introduceSelf() {

    console.log(`Hi! I'm ${this.name}`);

  }

}

class Student extends Person {

  #year; //# makes it private

  constructor(name, year) {

    super(name);

    this.#year = year;

  }

  introduceSelf() {

    console.log(`Hi! I'm ${this.name}, and I'm in year ${this.#year}.`);

  }

  canStudyArchery() {

    return this.#year > 1;

  }

}

const summers = new Student("Summers", 2);

summers.introduceSelf(); // Hi! I'm Summers, and I'm in year 2.

summers.canStudyArchery(); // true

summers.#year; // SyntaxError: Property '#year' is not accessible outside of class 'Student' because it has a private identifier

Private data properties must be declared in the class declaration, and their names start with #.

### [**Private methods**](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Classes_in_JavaScript#private_methods)

### You can have private methods as well as private data properties. Just like private data properties, their names start with #, and they can only be called by the **object's own methods:**

class Example {

  somePublicMethod() {

    this.#somePrivateMethod();

  }

  #somePrivateMethod() {

    console.log("You called me?");

  }

}

const myExample = new Example();

myExample.somePublicMethod(); // 'You called me?'

myExample.#somePrivateMethod(); // SyntaxError: Property '#somePrivateMethod' is not accessible outside of class 'Example' because it has a private identifier

For more on Classes, lets go to a different page.

**Advanced topic in Javascript Class:**

**https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes**

Classes are a template for creating objects. They encapsulate data with code to work on that data. Classes in JS are built on [prototypes](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain) but also have some syntax and semantics that are unique to classes.

Classes are in fact "special [functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions)", and just as you can define [function expressions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function) and [function declarations](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function), a class can be defined in two ways: a [class expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/class) or a [class declaration](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/class).

**// Declaration**

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

}

**// Expression;** the class is anonymous but assigned to a variable

const Rectangle = class {

constructor(height, width) {

this.height = height;

this.width = width;

}

};

// **Expression;** the class has its own name

const Rectangle = class Rectangle2 {

constructor(height, width) {

this.height = height;

this.width = width;

}

};

Unlike function declarations, class declarations have the same [temporal dead zone](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#temporal_dead_zone_tdz) restrictions as let or const and behave as if they are [not hoisted](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_Classes#class_declaration_hoisting). **Meaning, you can’t use them before declaring them.**

The body of a class is executed in [strict mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode) even without the "use strict" directive.

A **class element** can be characterized by **three aspects**:

* **Kind:** Getter, setter, method, or field
* **Location**: Static or instance
* **Visibility**: Public or private

Together, they add up to 16 possible combinations.

### [Method definitions in classes](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Method_definitions#method_definitions_in_classes)

You can use the exact same syntax to define public instance methods that are available on class instances.

class ClassWithPublicInstanceMethod {

  publicMethod() {

    return "hello world";

  }

}

Public instance methods are defined on the prototype property of the class and are thus shared by all instances of the class. They are writable, non-enumerable, and configurable.

Inside instance methods, [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) and [super](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super) work like in normal methods. Usually, this refers to the instance itself. In subclasses, super lets you access the prototype of the object that the method is attached to, allowing you to call methods from the superclass.

class BaseClass {

  msg = "hello world";

  basePublicMethod() {

    return this.msg;

  }

}

class SubClass extends BaseClass {

  subPublicMethod() {

    return super.basePublicMethod();

  }

}

const instance = new SubClass();

console.log(instance.subPublicMethod()); // "hello world"

# Public class fields

The term Class fields (fields inside a class) include both static field and instance fields.

Public class fields participate in prototype inheritance.

Private class fields DO NOT participate in prototype inheritance.

Class fields are similar to object properties, not variables, so we don't use keywords such as const to declare them.

class ClassWithField {

  instanceField;

  instanceFieldWithInitializer = "instance field";

  static staticField;

  static staticFieldWithInitializer = "static field";

}

There are some additional syntax restrictions:

* The **name** of a static property (field or method) **cannot be prototype**.
* The **name** of a class field (static or instance) **cannot be constructor**.

Public instance fields are added to the instance either at construction time in the base class (before the constructor body runs), or just after super() returns in a subclass. Fields without initializers are initialized to **undefined**.

# ‘Static’ properties (field/data, method):

You can use static properties (data/method) to perform any initialization that you need to do when the class is first loaded. For example, you might use a static method to load data from a database or initialize a shared resource that all instances of the class will use.

Static properties cannot be directly accessed on instances of the class. Instead, they're accessed on the class itself.

Static methods are often utility functions, such as functions to create or clone objects.

Static field are useful for caches, fixed-configuration, or any other data you don't need to be replicated across instances.

There are some additional syntax restrictions:

* The name of a static property (field or method) cannot be prototype.
* The name of a class field (static or instance) cannot be constructor.

Static fields without initializers are initialized to undefined. Public static fields are not reinitialized on subclasses, but can be accessed via the prototype chain.

### [Calling static members from another static method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static#calling_static_members_from_another_static_method)

In order to call a static method or property within another static method of the same class, you can use the [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) keyword.

### [Calling static members from a class constructor and other methods](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static#calling_static_members_from_a_class_constructor_and_other_methods)

Static members are not directly accessible using the [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) keyword from non-static methods. You need to call them using the class name: CLASSNAME.STATIC\_METHOD\_NAME() / CLASSNAME.STATIC\_PROPERTY\_NAME or by calling the method as a property of the constructor: **this**.constructor.STATIC\_METHOD\_NAME() / **this**.constructor.STATIC\_PROPERTY\_NAME

#### **Constructor**

The [constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor) method is a special method for creating and initializing an object created with a class. There can only be one special method with the name "constructor" in a class — a [SyntaxError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError) is thrown if the class contains more than one occurrence of a constructor method.

A constructor can use the [super](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super) keyword to call the constructor of the super class.

You can create instance properties inside the constructor:

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

}

**Alternatively, if your instance properties' values do not depend on the constructor's arguments, you can define them as**[**class fields**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes#field_declarations)**.**

### [Binding ‘this’ with instance and static methods](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes#binding_this_with_instance_and_static_methods)

When a static or instance method is called without a value for [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this), such as by assigning the method to a variable and then calling it, the ‘this’ value will be undefined inside the method.

/\*\*

 \* When a static or instance method is called without a value for this, such as by assigning the method to a variable and

 \* then calling it, the this value will be ‘undefined’ inside the method

 \*/

class Animal {

  speak() {

    return this;

  }

  static eat() {

    return this;

  }

}

// const obj = new Animal();

// console.log(obj.speak()); // Animal {} //it is an Animal object

// const speakFunction = obj.speak;

// console.log(speakFunction); //[Function: speak] -> we are just printing the function, not calling it.

// console.log(speakFunction ()); // undefined because this function is called without an object context, the this keyword does not have a value to reference.

console.log(Animal.eat()); // [class Animal]

const eat = Animal.eat(); //

console.log(eat); //[class Animal]

eat(); // undefined -> because it has no object context.

/\*\*

 \* In this example, we assign the 'speak' method of the 'Animal' object to a variable 'speakFunction' and call it without an object context.

 \* Since 'speakFunction' is just a function and not a method of an object, the 'this' keyword inside the function refers to the

 \* global object (window in the case of the browser), which does not have a 'speak' property. Therefore, the output is "undefined".

 \*/

Q: In javascript, explain why the following statement is true: 'When a static or instance method is called without a value for this, such as by assigning the method to a variable and then calling it, the this value will be undefined inside the method'

ChatGPT: In JavaScript, the **this** keyword refers to the object that the method is called on.

**This happens because when a method is called without an object context**, **the this keyword does not have a value to reference**. **When a method is called on an object, the object becomes the context for this, and the this keyword refers to the object.**

However, when a method is called without an object context, JavaScript sets the value of **this** to the global object (in the case of the browser, the **window** object). This can cause unexpected behavior if the method relies on **this** to reference properties or methods of the object.

To avoid this issue, it's important to always call methods on an object context or use **bind**, **call**, or **apply** to explicitly set the value of **this** for the method.

#### Private class features

Using private fields, the definition can be refined as below.

class Rectangle {

#height = 0;

#width;

constructor(height, width) {

this.#height = height;

this.#width = width;

}

}

It's an error to reference private fields from outside of the class; they can only be read or written within the class body. By defining things that are not visible outside of the class, you ensure that your classes' users can't depend on internals, which may change from version to version.

Private Fields can only be declared up-front in a field declaration. They cannot be created later through assigning to them, the way that normal properties can.

Using Classes in JavaScript:

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_Classes>

JavaScript is a prototype-based language — an object's behaviors are specified by its own properties and its prototype's properties.

In JavaScript, classes are mainly an abstraction over the existing prototypical inheritance mechanism — all patterns are convertible to prototype-based inheritance.

Classes themselves are normal JavaScript values as well, and have their own prototype chains. In fact, most plain JavaScript functions can be used as constructors — you use the new operator with a constructor function to create a new object.

* Classes create objects through the [new](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) operator.
* Each object has some properties (data or method) added by the class.
* The class stores some properties (data or method) itself, which are usually used to interact with instances.

These correspond to the three key features of classes:

* Constructor;
* Instance methods and instance fields;
* Static methods and static fields.

Classes are usually created with class declarations.

class MyClass {

  // Constructor

  constructor() {

    // Constructor body

  }

  // Instance field

  myField = "foo";

  // Instance method

  myMethod() {

    // myMethod body

  }

  // Static field

  static myStaticField = "bar";

  // Static method

  static myStaticMethod() {

    // myStaticMethod body

  }

  // Static block

  static {

    // Static initialization code: it is a code that runs when the class is first loaded.

  }

  // Fields, methods, static fields, and static methods all have

  // "private" forms

  #myPrivateField = "bar";

}

### [Class declaration hoisting](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_Classes#class_declaration_hoisting)

Unlike function declarations, class declarations are not [hoisted](https://developer.mozilla.org/en-US/docs/Glossary/Hoisting) (or, in some interpretations, hoisted but with the temporal dead zone restriction), which means you cannot use a class before it is declared.

new MyClass(); // ReferenceError: Cannot access 'MyClass' before initialization

class MyClass {}

There is a philosophy in object-oriented programming called "encapsulation". This means you should not access the underlying implementation of an object, but instead use well-abstracted methods to interact with it.

ChatGPT: By encapsulating the implementation details of an object, you can make your code more modular, easier to maintain, and less prone to bugs. This is because you can change the internal implementation of an object without affecting the rest of your code, as long as you keep the same interface (the set of methods and properties that are exposed to the outside world).

**Constructor:**

A function can be shared between all instances, but still have its behavior differ when different instances call it, because the value of this is different.

class Color {

constructor(r, g, b) {

this.values = [r, g, b];

}

getRed() {

return this.values[0];

}

}

const red = new Color(255, 0, 0);

console.log(red.getRed()); // 255

If you are curious where this method is stored in — it's defined on the prototype of all instances, or Color.prototype.

**Private Fields:**

Encapsulation: This means you should not access the underlying implementation of an object, but instead use well-abstracted methods to interact with it.

In order to refer to a private field anywhere in the class, you must declare it in the class body (you can't create a private property on the fly).

Private fields in JavaScript are **hard private**: if the class does not implement methods that expose these private fields, there's absolutely no mechanism to retrieve them from outside the class. This means you are safe to do any refactors to your class's private fields, as long as the behavior of exposed methods stay the same.

A class method **can read the private fields of other instances**, as long as they belong to the **same class**.

class Color {

  #values;

  constructor(r, g, b) {

    this.#values = [r, g, b];

  }

  redDifference(anotherColor) {

    // #values doesn't necessarily need to be accessed from this:

    // you can access private fields of other instances belonging

    // to the same class.

    return this.#values[0] - anotherColor.#values[0];

  }

}

const red = new Color(255, 0, 0);

const crimson = new Color(220, 20, 60);

red.redDifference(crimson); // 35

Accessing a nonexistent private property throws an error instead of returning undefined like normal properties do.

There are some limitations in using private properties: **the same name can't be declared twice in a single class**, and they **can't be deleted**. Both lead to early syntax errors.

**Methods**, [**getters, and setters**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_Classes#accessor_fields)**can be private as well.** They're useful when you have something complex that the class needs to do internally but no other part of the code should be allowed to call.

## [Accessor fields](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_Classes#accessor_fields)

In JavaScript, Accessor fields allow us to manipulate something as if its an "actual property".

## [Public fields](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_Classes#public_fields)

Allow every instance to have a property. Fields are usually designed to be independent of the constructor's parameters.

class MyClass {

luckyNumber = Math.random();

}

console.log(new MyClass().luckyNumber); // 0.5

console.log(new MyClass().luckyNumber); // 0.3

Public fields are almost equivalent to assigning a property to this. For example, the above example can also be converted to:

class MyClass {

constructor() {

this.luckyNumber = Math.random();

}

}

## [Static properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_Classes#static_properties)

Belong to the class.

[*Static properties*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static) are a group of class features that are defined on the class itself, rather than on individual instances of the class. These features include:

* Static methods
* Static fields
* Static getters and setters

Static properties are very similar to their instance counterparts, except that:

* They are all prefixed with static, and
* They are not accessible from instances.

S[tatic initialization block](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Static_initialization_blocks), which is a block of code that runs when the class is first loaded. Static initialization blocks are almost equivalent to immediately executing some code after a class has been declared. The only difference is that they have access to static private properties.

NameSpacing: Prefixing utility methods with what they deal with is called "namespacing"

## [Extends and inheritance](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_Classes#extends_and_inheritance)

A key feature that classes bring about (in addition to ergonomic encapsulation with private fields) is inheritance, which means **one object can "borrow" a large part of another object's behaviors, while overriding or enhancing certain parts with its own logic.**

In object-oriented programming, we would create a derived class. The derived class has access to **all public properties** of the parent class. In JavaScript, derived classes are declared with an ‘[extends](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/extends)’ clause, which indicates the class it extends from.

class ColorWithAlpha extends Color {

  #alpha;

  constructor(r, g, b, a) {

    super(r, g, b);

    this.#alpha = a;

  }

  get alpha() {

    return this.#alpha;

  }

  set alpha(value) {

    if (value < 0 || value > 1) {

      throw new RangeError("Alpha value must be between 0 and 1");

    }

    this.#alpha = value;

  }

}

 In the constructor, we are calling **super(r, g, b).** It is a language requirement to call [super()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super) before accessing this. The super() call calls the parent class's constructor to initialize this — here it's roughly equivalent to this = new Color(r, g, b). You can have code before super(), but you cannot access this before super() — the language prevents you from accessing the uninitialized this.

A derived class inherits all methods from its parent.

Derived classes can also override methods from the parent class.

Within derived classes: you can **access the parent class's methods** by using **super**. This allows you to build enhancement methods and avoid code duplication.

When you use **extends**, the static methods inherit from each other as well, so you can also override or enhance them.

Derived classes don't have access to the parent class's private fields — this is another key aspect to JavaScript private fields being "hard private". Private fields are scoped to the class body itself and do not grant access to any outside code.

A class can only extend from one class. This prevents problems in multiple inheritance like the [diamond problem](https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem). However, due to the dynamic nature of JavaScript, it's still possible to achieve the effect of multiple inheritance through class composition and [mixins](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/extends" \l "mix-ins).

Instances of derived classes are also [instances of](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof) the base class.

\*\*\*\*\*\*\*\*How to access super class’s properties using ‘this’:

To access the baseField property from the Extended class, you can simply refer to it using this.baseField since the property is inherited from the Base class.

\*\*\*\*\*\*\*\*How to access super class’s properties (non-static) using ‘super: create an instance of the Base class in the child class using super().

super() in the derived class is equivalent to ‘this’ = new Color(r,g,b), ‘super.anyProp’ refers to the property from the superclass. Extended class constructor calls the super() constructor to initialize the Base class.

3) If baseField is a static class property of Base, then it can be accessed using ‘super’ from the extended class without having to create an instance of base class using super(). Shown below

\* \*/

// class Base {

//   static baseField = 10;

// }

// class Extended extends Base {

//   extendedField = super.baseField; // 10

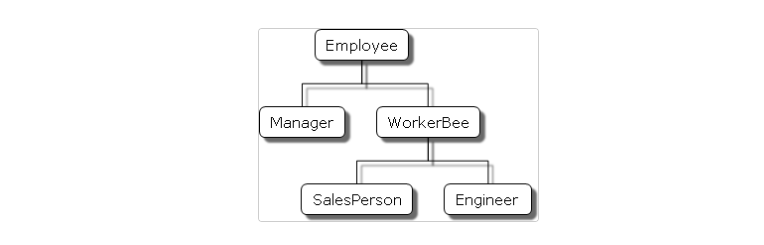
// }

## [Why classes?](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_Classes#why_classes)

Classes introduce a paradigm( fundamental style or approach to writing computer programs), or a way to organize your code. Classes are the foundations of object-oriented programming, which is built on concepts like [inheritance](https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)) and [polymorphism](https://en.wikipedia.org/wiki/Polymorphism_(computer_science)) (especially subtype polymorphism). However, many people are philosophically against certain OOP practices and don't use classes as a result.

Mutability and internal state are important aspects of object-oriented programming, but often make code hard to reason with — because any seemingly innocent operation may have unexpected side effects and change the behavior in other parts of the program.

In order to reuse code, we usually resort to extending classes, which can create big hierarchies of inheritance patterns.



However, it is often hard to describe inheritance cleanly when one class can only extend one other class. Often, we want the behavior of multiple classes. In Java, this is done through interfaces; in JavaScript, it can be done through **mixins**. But at the end of the day, it's still not very convenient.

On the brighter side, classes are a very powerful way to organize our code on a higher level.

In general, you should consider using classes when you want to create objects that store their own internal data and expose a lot of behavior. Take built-in JavaScript classes as examples:

* The [Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map) and [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set) classes store a collection of elements and allow you to access them by key using get(), set(), has(), etc.
* The [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) class stores a date as a Unix timestamp (a number) and allows you to format, update, and read individual date components.
* The [Error](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error) class stores information about a particular exception, including the error message, stack trace, cause, etc. It's one of the few classes that come with a rich inheritance structure: there are multiple built-in classes like [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError) and [ReferenceError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ReferenceError) that extend Error. In the case of errors, this inheritance allows refining the semantics of errors: each error class represents a specific type of error, which can be easily checked with [instanceof](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof).

JavaScript offers the mechanism to organize your code in a canonical (standard) object-oriented way, but whether and how to use it is entirely up to the programmer's discretion.

Private fields and methods are new features in classes with no trivial equivalent in function constructors.

# 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).**