JavaScript Class

**Summary**: in this tutorial, you’ll learn about the JavaScript class and how to use it effectively.

A JavaScript class is a blueprint for creating [objects](https://www.javascripttutorial.net/javascript-objects/). A class encapsulates data and functions that manipulate data.

Unlike other programming languages such as Java and [C#](https://www.csharptutorial.net/), JavaScript classes are syntactic sugar over the [prototypal inheritance](https://www.javascripttutorial.net/javascript-prototypal-inheritance/). In other words, ES6 classes are just special [functions](https://www.javascripttutorial.net/javascript-function/).

Classes prior to ES6 revisited

Prior to ES6, JavaScript had no concepts of classes. To mimic a class, you often use the [constructor/prototype pattern](https://www.javascripttutorial.net/javascript-constructor-prototype/) as shown in the following example:

function Person(name) {

this.name = name;

}

Person.prototype.getName = function () {

return this.name;

};

var john = new Person("John Doe");

console.log(john.getName());

Code language: JavaScript (javascript)

Output:

John Doe

How it works.

First, create the Person as a constructor function that has a property name called name. The getName() function is assigned to the prototype so that it can be shared by all instances of the Person type.

Then, create a new instance of the Person type using the new operator. The john object, hence, is an instance of the Person and Object through [prototypal inheritance](https://www.javascripttutorial.net/javascript-prototypal-inheritance/).

The following statements use the instanceof operator to check if john is an instance of the Person and Object type:

console.log(john instanceof Person); *// true*

console.log(john instanceof Object); *// true*

Code language: JavaScript (javascript)

ES6 class declaration

ES6 introduced a new syntax for declaring a class as shown in this example:

class Person {

constructor(name) {

this.name = name;

}

getName() {

return this.name;

}

}

Code language: JavaScript (javascript)

This Person class behaves like the Person type in the previous example. However, instead of using a constructor/prototype pattern, it uses the class keyword.

In the Person class, the constructor() is where you can initialize the properties of an instance. JavaScript automatically calls the constructor() method when you instantiate an object of the class.

The following creates a new Person object, which will automatically call the constructor() of the Person class:

let john = new Person("John Doe");

Code language: JavaScript (javascript)

The getName() is called a method of the Person class. Like a constructor function, you can call the methods of a class using the following syntax:

objectName.methodName(args)

Code language: CSS (css)

For example:

let name = john.getName();

console.log(name); *// "John Doe"*

Code language: JavaScript (javascript)

To verify the fact that classes are special functions, you can use the typeof operator of to check the type of the Person class.

console.log(typeof Person); *// function*

Code language: JavaScript (javascript)

It returns function as expected.

The john object is also an instance of the Person and Object types:

console.log(john instanceof Person); *// true*

console.log(john instanceof Object); *// true*

Code language: JavaScript (javascript)

Class vs. Custom type

Despite the similarities between a class and a custom type defined via a constructor function, there are some important differences.

First, class declarations are not [hoisted](https://www.javascripttutorial.net/javascript-hoisting/) like function declarations.

For example, if you place the following code above the Person class declaration section, you will get a ReferenceError.

let john = new Person("John Doe");

Code language: JavaScript (javascript)

Error:

Uncaught ReferenceError: Person is not defined

Code language: JavaScript (javascript)

Second, all the code inside a class automatically executes in the strict mode. And you cannot change this behavior.

Third, class methods are [non-enumerable](https://www.javascripttutorial.net/javascript-enumerable-properties/). If you use a constructor/prototype pattern, you have to use the Object.defineProperty() method to make a property non-enumerable.

Finally, calling the class constructor without the new operator will result in an error as shown in the following example.

let john = Person("John Doe");

Code language: JavaScript (javascript)

Error:

Uncaught TypeError: Class constructor Person cannot be invoked without 'new'

Code language: JavaScript (javascript)

Note that it’s possible to call the constructor function without the new operator. In this case, the constructor function behaves like a regular function.

Summary

* Use the JavaScript class keyword to declare a new class.
* A class declaration is syntactic sugar over [prototypal inheritance](https://www.javascripttutorial.net/javascript-prototypal-inheritance/) with additional enhancements.

JavaScript Getters and Setters

**Summary**: in this tutorial, you will learn about JavaScript getters and setters and how to use them effectively.

Introduction to the JavaScript getters and setters

The following example [defines a class](https://www.javascripttutorial.net/es6/javascript-class/) called Person:

class Person {

constructor(name) {

this.name = name;

}

}

let person = new Person("John");

console.log(person.name); *// John*

Code language: JavaScript (javascript)

The Person class has a property name and a constructor. The constructor initializes the name property to a string.

Sometimes, you don’t want the name property to be accessed directly like this:

person.name

Code language: CSS (css)

To do that, you may come up with a pair of methods that manipulate the name property. For example:

class Person {

constructor(name) {

this.setName(name);

}

getName() {

return this.name;

}

setName(newName) {

newName = newName.trim();

if (newName === '') {

throw 'The name cannot be empty';

}

this.name = newName;

}

}

let person = new Person('Jane Doe');

console.log(person); *// Jane Doe*

person.setName('Jane Smith');

console.log(person.getName()); *// Jane Smith*

Code language: JavaScript (javascript)

In this example, the Person class has the name property. Also, it has two additional methods getName() and setName().

The getName() method returns the value of the name property.

The setName() method assigns an argument to the name property. The setName() removes the whitespaces from both ends of the newName argument and throws an exception if the newName is empty.

The constructor() calls the setName() method to initialize the name property:

constructor(name) {

this.setName(name);

}

Code language: JavaScript (javascript)

The getName() and setName() methods are known as getter and setter in other programming languages such as Java and C++.

ES6 provides specific syntax for defining the getter and setter using the get and set keywords. For example:

class Person {

constructor(name) {

this.name = name;

}

get name() {

return this.\_name;

}

set name(newName) {

newName = newName.trim();

if (newName === '') {

throw 'The name cannot be empty';

}

this.\_name = newName;

}

}

Code language: JavaScript (javascript)

How it works.

First, the name property is changed to \_name to avoid the name collision with the getter and setter.

Second, the getter uses the get keyword followed by the method name:

get name() {

return this.\_name;

}

Code language: JavaScript (javascript)

To call the getter, you use the following syntax:

let name = person.name;

Code language: JavaScript (javascript)

When JavaScript sees the access to name property of the Person class, it checks if the Person class has any name property.

If not, JavaScript checks if the Person class has any method that binds to the name property. In this example, the name() method binds to the name property via the get keyword. Once JavaScript finds the getter method, it executes the getter method and returns a value.

Third, the setter uses the set keyword followed by the method name:

set name(newName) {

newName = newName.trim();

if (newName === '') {

throw 'The name cannot be empty';

}

this.\_name = newName;

}

Code language: JavaScript (javascript)

JavaScript will call the name() setter when you assign a value to the name property like this:

person.name = 'Jane Smith';

Code language: JavaScript (javascript)

If a class has only a getter but not a setter and you attempt to use the setter, the change won’t take any effect. See the following example:

class Person {

constructor(name) {

this.\_name = name;

}

get name() {

return this.\_name;

}

}

let person = new Person("Jane Doe");

console.log(person.name);

*// attempt to change the name, but cannot*

person.name = 'Jane Smith';

console.log(person.name); *// Jane Doe*

Code language: JavaScript (javascript)

In this example, the Person class has the name getter but not the name setter. It attempts to call the setter. However, the change doesn’t take effect since the Person class doesn’t have the name setter.

Using getter in an object literal

The following example defines a getter called latest to return the latest attendee of the meeting object:

let meeting = {

attendees: [],

add(attendee) {

console.log(`${attendee} joined the meeting.`);

this.attendees.push(attendee);

return this;

},

get latest() {

let count = this.attendees.length;

return count == 0 ? undefined : this.attendees[count - 1];

}

};

meeting.add('John').add('Jane').add('Peter');

console.log(`The latest attendee is ${meeting.latest}.`);

Code language: JavaScript (javascript)

Output:

John joined a meeting.

Jane joined a meeting.

Peter joined a meeting.

The latest attendee is Peter.

Summary

* Use the get and set keywords to define the JavaScript getters and setters for a class or an object.
* The get keyword binds an object property to a method that will be invoked when that property is looked up.
* The set keyword binds an object property to a method that will be invoked when that property is assigned.

JavaScript Class Expressions

**Summary**: in this tutorial, you’ll learn how to use JavaScript class expressions to declare new classes.

Introduction to JavaScript class expressions

Similar to [functions](https://www.javascripttutorial.net/javascript-function/), [classes](https://www.javascripttutorial.net/es6/javascript-class/) have expression forms. A class expression provides you with an alternative way to define a new class.

A class expression doesn’t require an identifier after the class keyword. And you can use a class expression in a [variable declaration](https://www.javascripttutorial.net/javascript-variables/) and pass it into a function as an argument.

For example, the following defines a class expression:

let Person = class {

constructor(name) {

this.name = name;

}

getName() {

return this.name;

}

}

Code language: Python (python)

How it works.

On the left side of the expression is the Person variable. It’s assigned to a class expression.

The class expression starts with the keyword class followed by the class definition.

A class expression may have a name or not. In this example, we have an unnamed class expression.

If a class expression has a name, its name can be local to the class body.

The following creates an instance of the Person class expression. Its syntax is the same as if it were a class declaration.

let person = new Person('John Doe');

Code language: Python (python)

Like a [class declaration](https://www.javascripttutorial.net/es6/javascript-class/), the type of a class expression is also a [function](https://www.javascripttutorial.net/javascript-function/):

console.log(typeof Person); // function

Code language: Python (python)

Similar to function expressions, class expressions are not [hoisted](https://www.javascripttutorial.net/javascript-hoisting/). It means that you cannot create an instance of the class before defining the class expression.

First-class citizen

[JavaScript classes are first-class citizens](https://www.javascripttutorial.net/javascript-functions-are-first-class-citizens/). It means that you can pass a class into a function, return it from a function, and assign it to a variable.

See the following example:

function factory(aClass) {

return new aClass();

}

let greeting = factory(class {

sayHi() { console.log('Hi'); }

});

greeting.sayHi(); // 'Hi'

Code language: Python (python)

How it works.

First, define a factory() function that takes a class expression as an argument and return the instance of the class:

function factory(aClass) {

return new aClass();

}

Code language: Python (python)

Second, pass an unnamed class expression to the factory() function and assign its result to the greeting variable:

let greeting = factory(class {

sayHi() { console.log('Hi'); }

});

Code language: Python (python)

The class expression has a method called sayHi(). And the greeting variable is an instance of the class expression.

Third, call the sayHi() method on the greeting object:

greeting.sayHi(); // 'Hi'

Code language: Python (python)

Singleton

Singleton is a design pattern that limits the instantiation of a class to a single instance. It ensures that only one instance of a class can be created throughout the system.

Class expressions can be used to create a singleton by calling the class constructor immediately.

To do that, you use the new operator with a class expression and include the parentheses at the end of class declaration as shown in the following example:

let app = new class {

constructor(name) {

this.name = name;

}

start() {

console.log(`Starting the ${this.name}...`);

}

}('Awesome App');

app.start(); // Starting the Awesome App...

Code language: Python (python)

How it works.

The following is an unnamed class expresion:

new class {

constructor(name) {

this.name = name;

}

start() {

console.log(`Starting the ${this.name}...`);

}

}

Code language: Python (python)

The class has a constructor() that accepts an argument. It aslo has a method called start().

The class expression evaluates to a class. Therefore, you can call its constructor immediately by placing parentheses after the expression:

new class {

constructor(name) {

this.name = name;

}

start() {

console.log(`Starting the ${this.name}...`);

}

}('Awesome App')

Code language: Python (python)

This expression returns an instance of the class expression which is assigned to the app variable.

The following calls the start() method on the app object:

app.start(); // Starting the Awesome App...

Code language: Python (python)

Summary

* ES6 provides you with an alternative way to defining a new class using a class expression.
* Class expressions can be named or unnamed.
* The class expression can be used to create a singleton object.

JavaScript Computed Property

**Summary**: in this tutorial, you’ll learn about the JavaScript computed properties introduced in ES6.

Introduction to JavaScript Computed Property

ES6 allows you to use an expression in brackets []. It’ll then use the result of the expression as the property name of an object. For example:

let propName = 'c';

const rank = {

a: 1,

b: 2,

[propName]: 3,

};

console.log(rank.c); *// 3*

Code language: JavaScript (javascript)

In this example, the [propName] is a computed property of the rank object. The property name is derived from the value of the propName variable.

When you access c property of the rank object, JavaScript evaluates propName and returns the property’s value.

Like an [object literal](https://www.javascripttutorial.net/javascript-objects/), you can use computed properties for [getters and setters](https://www.javascripttutorial.net/es6/javascript-getters-and-setters/) of a [class](https://www.javascripttutorial.net/es6/javascript-class/). For example:

let name = 'fullName';

class Person {

constructor(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

get [name]() {

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

}

}

let person = new Person('John', 'Doe');

console.log(person.fullName);

Code language: JavaScript (javascript)

Output:

John Doe

How it works:

The get[name] is a computed property name of a getter of the Person class. At runtime, when you access the fullName property, the person object calls the getter and returns the full name.

Summary

* Computed properties allow you to use the values of expressions as property names of an object.

JavaScript Inheritance Using extends & super

**Summary**: in this tutorial, you will learn how to implement JavaScript inheritance by using extends and super in ES6.

Implementing JavaScript inheritance using extends and super

Prior to ES6, implementing a proper inheritance required multiple steps. One of the most commonly used strategies is [prototypal inheritance](https://www.javascripttutorial.net/javascript-prototypal-inheritance/).

The following illustrates how the Bird inherits properties from the Animal using the prototypal inheritance technique:

function Animal(legs) {

this.legs = legs;

}

Animal.prototype.walk = function() {

console.log('walking on ' + this.legs + ' legs');

}

function Bird(legs) {

Animal.call(this, legs);

}

Bird.prototype = Object.create(Animal.prototype);

Bird.prototype.constructor = Animal;

Bird.prototype.fly = function() {

console.log('flying');

}

var pigeon = new Bird(2);

pigeon.walk(); *// walking on 2 legs*

pigeon.fly(); *// flying*

Code language: JavaScript (javascript)

ES6 simplified these steps by using the extends and super keywords.

The following example defines the Animal and Bird classes and establishes the inheritance through the extends and super keywords.

class Animal {

constructor(legs) {

this.legs = legs;

}

walk() {

console.log('walking on ' + this.legs + ' legs');

}

}

class Bird extends Animal {

constructor(legs) {

super(legs);

}

fly() {

console.log('flying');

}

}

let bird = new Bird(2);

bird.walk();

bird.fly();

Code language: JavaScript (javascript)

How it works.

First, use the extends keyword to make the Bird class inheriting from the Animal class:

class Bird extends Animal {

*// ...*

}

Code language: JavaScript (javascript)

The Animal class is called a **base class** or **parent class** while the Bird class is known as a **derived class** or **child class**. By doing this, the Bird class inherits all methods and properties of the Animal class.

Second, in the Bird‘s constructor, call super() to invoke the Animal‘s constructor with the legs argument.

JavaScript requires the child class to call super() if it has a constructor. As you can see in the Bird class, the super(legs) is equivalent to the following statement in ES5:

Animal.call(this, legs);

Code language: JavaScript (javascript)

If the Bird class doesn’t have a constructor, you can don’t need to do anything else:

class Bird extends Animal {

fly() {

console.log('flying');

}

}

Code language: JavaScript (javascript)

It is equivalent to the following class:

class Bird extends Animal {

constructor(...args) {

super(...args);

}

fly() {

console.log('flying');

}

}

Code language: JavaScript (javascript)

However, the child class has a constructor, it needs to call super(). For example, the following code results in an error:

class Bird extends Animal {

constructor(legs) {

}

fly() {

console.log('flying');

}

}

Code language: JavaScript (javascript)

Error:

ReferenceError: Must call super constructor in derived class before accessing 'this' or returning from derived constructor

Code language: JavaScript (javascript)

Because the super() initializes the this object, you need to call the super() before accessing the this object. Trying to access this before calling super() also results in an error.

For example, if you want to initialize the color property of the Bird class, you can do it as follows:

class Bird extends Animal {

constructor(legs, color) {

super(legs);

this.color = color;

}

fly() {

console.log("flying");

}

getColor() {

return this.color;

}

}

let pegion = new Bird(2, "white");

console.log(pegion.getColor());

Code language: JavaScript (javascript)

Shadowing methods

ES6 allows the child class and parent class to have methods with the same name. In this case, when you call the method of an object of the child class, the method in the child class will shadow the method in the parent class.

The following Dog class extends the Animal class and redefines the walk() method:

class Dog extends Animal {

constructor() {

super(4);

}

walk() {

console.log(`go walking`);

}

}

let bingo = new Dog();

bingo.walk(); *// go walking*

Code language: JavaScript (javascript)

To call the method of the parent class in the child class, you use super.method(arguments) like this:

class Dog extends Animal {

constructor() {

super(4);

}

walk() {

super.walk();

console.log(`go walking`);

}

}

let bingo = new Dog();

bingo.walk();

*// walking on 4 legs*

*// go walking*

Code language: JavaScript (javascript)

Inheriting static members

Besides the properties and methods, the child class also inherits all static properties and methods of the parent class. For example:

class Animal {

constructor(legs) {

this.legs = legs;

}

walk() {

console.log('walking on ' + this.legs + ' legs');

}

static helloWorld() {

console.log('Hello World');

}

}

class Bird extends Animal {

fly() {

console.log('flying');

}

}

Code language: JavaScript (javascript)

In this example, the Animal class has the helloWorld() static method and this method is available as Bird.helloWorld() and behaves the same as the Animal.helloWorld() method:

Bird.helloWorld(); *// Hello World*

Code language: JavaScript (javascript)

Inheriting from built-in types

JavaScript allows you to extend a built-in type such as [Array](https://www.javascripttutorial.net/javascript-array/), String, [Map](https://www.javascripttutorial.net/es6/javascript-map/), and [Set](https://www.javascripttutorial.net/es6/javascript-set/)through inheritance.

The following Queue class extends the Array reference type. The syntax is much cleaner than the [Queue](https://www.javascripttutorial.net/javascript-queue/) implemented using the [constructor/prototype pattern](https://www.javascripttutorial.net/create-objects-in-javascript/#constructor_prototype_pattern).

class Queue extends Array {

enqueue(e) {

super.push(e);

}

dequeue() {

return super.shift();

}

peek() {

return !this.empty() ? this[0] : undefined;

}

empty() {

return this.length === 0;

}

}

var customers = new Queue();

customers.enqueue('A');

customers.enqueue('B');

customers.enqueue('C');

while (!customers.empty()) {

console.log(customers.dequeue());

}

Code language: JavaScript (javascript)

Summary

* Use the extends keyword to implement the inheritance in ES6. The class to be extended is called a base class or parent class. The class that extends the base class or parent class is called the derived class or child class.
* Call the super(arguments) in the child class’s constructor to invoke the parent class’s constructor.
* Use super keyword to call methods of the parent class in the methods of the child class.

Introduction to JavaScript new.target Metaproperty

**Summary**: in this tutorial, you learn about the JavaScript new.target metaproperty that detects whether a function or constructor was called using the new operator.

Introduction to JavaScript new.target



ES6 provides a metaproperty named new.target that allows you to detect whether a [function](https://www.javascripttutorial.net/javascript-function/)or constructor was called using the new operator.

The new.target consists of the new keyword, a dot, and target property. The new.target is available in all functions.

However, in [arrow functions](https://www.javascripttutorial.net/es6/javascript-arrow-function/), the new.target is the one that belongs to the surrounding function.

The new.target is very useful to inspect at runtime whether a function is being executed as a function or as a constructor. It is also handy to determine a specific derived class that was called by using the new operator from within a parent class.

JavaScript new.target in functions

Let’s see the following Person constructor function:

function Person(name) {

this.name = name;

}

Code language: JavaScript (javascript)

You can create a new object from the Person function by using the new operator as follows:

let john = new Person('John');

console.log(john.name); *// john*

Code language: JavaScript (javascript)

Or you can call the Person as a function:

Person('Lily');

Code language: JavaScript (javascript)

Because the this is set to the [global object](https://www.javascripttutorial.net/es-next/javascript-globalthis/) i.e., the window object when you run JavaScript in the web browser, the name property is added to the window object as follows:

console.log(window.name); *//Lily*

Code language: JavaScript (javascript)

To help you detect whether a function was called using the new operator, you use the new.target metaproperty.

In a regular function call, the new.target returns undefined. If the function was called with the new operator, the new.target returns a reference to the function.

Suppose you don’t want the Person to be called as a function, you can use the new.target as follows:

function Person(name) {

if (!new.target) {

throw "must use new operator with Person";

}

this.name = name;

}

Code language: JavaScript (javascript)

Now, the only way to use Person is to instantiate an object from it by using the new operator. If you try to call it as a regular function, you will get an error.

JavaScript new.target in constructors

In a [class](https://www.javascripttutorial.net/es6/javascript-class/)constructor, the new.target refers to the constructor that was invoked directly by the new operator. It is true if the constructor is in the parent class and was delegated from the constructor of the child class:

class Person {

constructor(name) {

this.name = name;

console.log(new.target.name);

}

}

class Employee extends Person {

constructor(name, title) {

super(name);

this.title = title;

}

}

let john = new Person('John Doe'); *// Person*

let lily = new Employee('Lily Bush', 'Programmer'); *// Employee*

Code language: JavaScript (javascript)

In this example, new.target.name is the human-friendly name of the constructor reference of new.target

In this tutorial, you have learned how to use the JavaScript new.target metaproperty to detect whether a function or constructor was called using the new operator.

JavaScript Static Methods

**Summary**: in this tutorial, you’ll learn about the JavaScript static methods and how to use them effectively.

Introduction to the JavaScript static methods

By definition, static methods are bound to a [class](https://www.javascripttutorial.net/es6/javascript-class/), not the instances of that class. Therefore, static methods are useful for defining helper or utility methods.

To define a static method before ES6, you add it directly to the constructor of the class. For example, suppose you have a Person type as follows:

function Person(name) {

this.name = name;

}

Person.prototype.getName = function () {

return this.name;

};

Code language: JavaScript (javascript)

The following adds a static method called createAnonymous() to the Person type:

Person.createAnonymous = function (gender) {

let name = gender == "male" ? "John Doe" : "Jane Doe";

return new Person(name);

};

Code language: JavaScript (javascript)

The createAnonymous() method is considered a static method because it doesn’t depend on any instance of the Person type for its property values.

To call the createAnonymous() method, you use the Person type instead of its instances:

var anonymous = Person.createAnonymous();

Code language: JavaScript (javascript)

JavaScript static methods in ES6

In ES6, you define static methods using the static keyword. The following example defines a static method called createAnonymous() for the Person class:

class Person {

constructor(name) {

this.name = name;

}

getName() {

return this.name;

}

static createAnonymous(gender) {

let name = gender == "male" ? "John Doe" : "Jane Doe";

return new Person(name);

}

}

Code language: JavaScript (javascript)

To invoke the static method, you use the following syntax:

let anonymous = Person.createAnonymous("male");

Code language: JavaScript (javascript)

If you attempt to call the static method from an instance of the class, you’ll get an error. For example:

let person = new Person('James Doe');

let anonymous = person.createAnonymous("male");

Code language: JavaScript (javascript)

Error:

TypeError: person.createAnonymous is not a function

Code language: JavaScript (javascript)

Calling a static method from the class constructor or an instance method

To call a static method from a class constructor or an instance method, you use the class name, followed by the . and the static method:

className.staticMethodName();

Code language: CSS (css)

Alternatively, you can use the following syntax:

this.constructor.staticMethodName();

Code language: CSS (css)

Summary

* JavaScript static methods are shared among instances of a class. Therefore, they are bound to the class.
* Call the static methods via the class name, not the instances of that class.
* Use the className.staticMethodName() or this.constructor.staticMethodName() to call a static method in a class constructor or an instance method.

JavaScript Static Properties

**Summary**: in this tutorial, you’ll learn about the JavaScript static properties of a class and how to access the static properties in a static method, class constructor, and other instance methods.

Introduction to the JavaScript static properties

Like a [static method](https://www.javascripttutorial.net/es6/javascript-static-method/), a static property is shared by all instances of a [class](https://www.javascripttutorial.net/es6/javascript-class/). To define static property, you use the static keyword followed by the property name like this:

class Item {

static count = 0;

}

Code language: JavaScript (javascript)

To access a static property, you use the class name followed by the . operator and the static property name. For example:

console.log(Item.count); *// 0*

Code language: JavaScript (javascript)

To access the static property in a static method, you use the class name followed by the . operator and the static property name. For example:

class Item {

static count = 0;

static getCount() {

return Item.count;

}

}

console.log(Item.getCount()); *// 0*

Code language: JavaScript (javascript)

To access a static property in a class constructor or instance method, you use the following syntax:

className.staticPropertyName;

Code language: CSS (css)

Or

this.constructor.staticPropertyName;

Code language: CSS (css)

The following example increases the count static property in the class constructor:

class Item {

constructor(name, quantity) {

this.name = name;

this.quantity = quantity;

this.constructor.count++;

}

static count = 0;

static getCount() {

return Item.count;

}

}

Code language: JavaScript (javascript)

When you create a new instance of the Item class, the following statement increases the count static property by one:

this.constructor.count++;

Code language: CSS (css)

For example:

*// Item class ...*

let pen = new Item("Pen", 5);

let notebook = new Item("notebook", 10);

console.log(Item.getCount()); *// 2*

Code language: JavaScript (javascript)

This example creates two instances of the Item class, which calls the class constructor. Since the class constructor increases the count property by one each time it’s called, the value of the count is two.

Put it all together.

class Item {

constructor(name, quantity) {

this.name = name;

this.quantity = quantity;

this.constructor.count++;

}

static count = 0;

static getCount() {

return Item.count;

}

}

let pen = new Item('Pen', 5);

let notebook = new Item('notebook', 10);

console.log(Item.getCount()); *// 2*

Code language: JavaScript (javascript)

Summary

* A static property of a class is shared by all instances of that class.
* Use the static keyword to define a static property.
* Use the className.staticPropertyName to access the static property in a static method.
* Use the this.constructor.staticPropertyName or className.staticPropertyName to access the static property in a constructor.

JavaScript Private Fields

**Summary**: in this tutorial, you’ll learn about JavaScript private fields and how to use them effectively.

Introduction to the JavaScript private fields

ES2022 allows you to define private fields for a [class](https://www.javascripttutorial.net/es6/javascript-class/). To define a private field, you prefix the field name with the # sign.

For example, the following defines the Circle class with a private field radius:

class Circle {

*#radius;*

constructor(value) {

this.*#radius = value;*

}

get area() {

return Math.PI \* Math.pow(this.*#radius, 2);*

}

}

Code language: PHP (php)

In this example:

* First, define the private field #radius in the class body.
* Second, initialize the #radius field in the constructor with an argument.
* Third, calculate the area of the circle by accessing the #radius private field in the getter method.

The following creates a new instance of the Circle class and calculates its area:

let circle = new Circle(10);

console.log(circle.area); *// 314.1592653589793*

Code language: JavaScript (javascript)

Because the #radius is a private field, you can only access it inside the Circle class. In other words, the #radius field is invisible outside of the Circle class.

Using getter and setter to access private fields

The following redefines the Circle class by adding the radius getter and setter to provide access to the #radius private field:

class Circle {

*#radius = 0;*

constructor(radius) {

this.radius = radius;

}

get area() {

return Math.PI \* Math.pow(this.radius, 2);

}

set radius(value) {

if (typeof value === 'number' && value > 0) {

this.*#radius = value;*

} else {

throw 'The radius must be a positive number';

}

}

get radius() {

return this.*#radius;*

}

}

Code language: PHP (php)

How it works.

* The radius setter validates the argument before assigning it to the #radius private field. If the argument is not a positive number, the radius setter throws an error.
* The radius getter returns the value of the #radius private field.
* The constructor calls the radius setter to assign the argument to the #radius private field.

Private fields and subclasses

Private fields are only accessible inside the class where they’re defined. Also, they’re not accessible from the subclasses. For example, the following defines the Cylinder class that [extends](https://www.javascripttutorial.net/es6/javascript-inheritance/) the Circle class:

class Cylinder extends Circle {

*#height;*

constructor(radius, height) {

super(radius);

this.*#height = height;*

*// cannot access the #radius of the Circle class here*

}

}

Code language: PHP (php)

If you attempt to access the #radius private field in the Cylinder class, you’ll get a SyntaxError.

The in operator: check private fields exist

To check if an object has a private field inside a class, you use the in operator:

fieldName in objectName

For example, the following adds the hasRadius() static method to the Circle class that uses the in operator to check if the circle object has the #radius private field:

class Circle {

*#radius = 0;*

constructor(radius) {

this.radius = radius;

}

get area() {

return Math.PI \* Math.pow(this.radius, 2);

}

set radius(value) {

if (typeof value === 'number' && value > 0) {

this.*#radius = value;*

} else {

throw 'The radius must be a positive number';

}

}

get radius() {

return this.*#radius;*

}

static hasRadius(circle) {

return *#radius in circle;*

}

}

let circle = new Circle(10);

console.log(Circle.hasRadius(circle));

Code language: PHP (php)

Output:

true

Code language: JavaScript (javascript)

Static private fields

The following example shows how to use use a static private field:

class Circle {

*#radius = 0;*

static *#count = 0;*

constructor(radius) {

this.radius = radius;

Circle.*#count++;*

}

get area() {

return Math.PI \* Math.pow(this.radius, 2);

}

set radius(value) {

if (typeof value === 'number' && value > 0) {

this.*#radius = value;*

} else {

throw 'The radius must be a positive number';

}

}

get radius() {

return this.*#radius;*

}

static hasRadius(circle) {

return *#radius in circle;*

}

static getCount() {

return Circle.*#count;*

}

}

let circles = [new Circle(10), new Circle(20), new Circle(30)];

console.log(Circle.getCount());

Code language: PHP (php)

How it works.

First, add a private static field #count to the Circle class and initialize its value to zero:

static *#count = 0;*

Code language: PHP (php)

Second, increase the #count by one in the constructor:

Circle.#count++;

Code language: CSS (css)

Third, define a static method that returns the value of the #count private static field:

static getCount() {

return Circle.*#count;*

}

Code language: PHP (php)

Finally, create three instances of the Circle class and output the count value to the console:

let circles = [new Circle(10), new Circle(20), new Circle(30)];

console.log(Circle.getCount());

Code language: JavaScript (javascript)

Summary

* Prefix the field name with # sign to make it private.
* Private fields are accessible only inside the class, not from outside of the class or subclasses.
* Use the in operator to check if an object has a private field.

# JavaScript Private Methods

**Summary**: in this tutorial, you’ll learn about JavaScript private methods including private instance methods, private static methods, and private getter/setter.

## Introduction to JavaScript private methods

By default, members of a [class](https://www.javascripttutorial.net/es6/javascript-class/) are public. ES2020 introduced the private members that include [private fields](https://www.javascripttutorial.net/javascript-private-fields/) and methods.

To make a public method private, you prefix its name with a hash #. JavaScript allows you to define private methods for instance methods, [static methods](https://www.javascripttutorial.net/es6/javascript-static-method/), and [getter/setters](https://www.javascripttutorial.net/es6/javascript-getters-and-setters/).

The following shows the syntax of defining a private instance method:

class MyClass {

#privateMethod() {

*//...*

}

}

Code language: JavaScript (javascript)

In this syntax, the #privateMethod is a private instance method. It can only be called inside the MyClass. In other words, it cannot be called from outside the class or in the subclasses of the MyClass.

To call the #privateMethod inside the MyClass, you use the this keyword as follows:

this.#privateMethod();

Code language: JavaScript (javascript)

The following illustrates the syntax of defining a private static method:

class MyClass {

static #privateStaticMethod() {

*//...*

}

}

Code language: JavaScript (javascript)

To call the #privateStaticMethod() inside the MyClass, you use the class name instead of the this keyword:

MyClass.#privateStaticMethod();

Code language: JavaScript (javascript)

The following shows the syntax of the private getters/setters:

class MyClass {

#field;

get #myField() {

return #field;

}

set #myField(value){

#field = value;

}

}

Code language: JavaScript (javascript)

In this example, the #myField is the private getter and setter that provide access to the private field #field.

In practice, you use private methods to minimize the number of methods that the object exposes.

As a rule of thumb, you should make all class methods private by default first. And then you make a method public whenever the object needs to use that method to interact with other objects.

## JavaScript private method examples

Let’s take some examples of using private methods

### **1) Private instance method example**

The following illustrates how to define the Person class with private instance methods:

class Person {

#firstName;

#lastName;

constructor(firstName, lastName) {

this.#firstName = firstName;

this.#lastName = lastName;

}

getFullName(format = true) {

return format ? this.#firstLast() : this.#lastFirst();

}

#firstLast() {

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

}

#lastFirst() {

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

}

}

let person = new Person('John', 'Doe');

console.log(person.getFullName());

Code language: JavaScript (javascript)

Output:

John Doe

Code language: JavaScript (javascript)

In this example:

First, define two private fields #firstName and #lastName in the Person class body.

Second, define the private methods #firstLast() and #lastFirst(). These methods return the full name in different formats.

Third, define the public instance method getFullName() that returns a person’s full name. The getFullName() method calls the private method #firstLast() and #lastFirst() to return the full name.

Finally, create a new person object and output the full name to the console.

### **2) Private static method example**

The following adds the #validate() private static method to the Person class:

class Person {

#firstName;

#lastName;

constructor(firstName, lastName) {

this.#firstName = Person.#validate(firstName);

this.#lastName = Person.#validate(lastName);

}

getFullName(format = true) {

return format ? this.#firstLast() : this.#lastFirst();

}

static #validate(name) {

if (typeof name === 'string') {

let str = name.trim();

if (str.length === 3) {

return str;

}

}

throw 'The name must be a string with at least 3 characters';

}

#firstLast() {

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

}

#lastFirst() {

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

}

}

let person = new Person('John', 'Doe');

console.log(person.getFullName());

Code language: JavaScript (javascript)

How it works.

First, define the static method #validate() that returns a value if it is a string with at least three characters. The method raises an exception otherwise.

Second, call the #validate() private static method in the constructor to validate the firstName and lastName arguments before assigning them to the corresponding private attributes.

### **Summary**

* Prefix a method name with the # to make it private.
* Private methods can be called inside the class, not from outside of the class or in the subclasses.

JavaScript instanceof

**Summary**: in this tutorial, you’ll learn how to use the JavaScript instanceof operator to determine if a constructor’s prototype appears in the prototype chain of an object.

Introduction to the JavaScript instanceof operator

The instanceof operator returns true if a [prototype](https://www.javascripttutorial.net/javascript-prototype/) of a constructor (constructor.prototype) appears in the prototype chain of an object.

The following shows the syntax of the instanceof operator:

object instanceof contructor

Code language: JavaScript (javascript)

In this syntax:

* object is the object to test.
* constructor is a function to test against.

JavaScript instanceof operator example

The following example defines the Person type and uses the instanceof operator to check if an object is an instance of that type:

function Person(name) {

this.name = name;

}

let p1 = new Person('John');

console.log(p1 instanceof Person); *// true*

Code language: JavaScript (javascript)

How it works.

First, define a Person type using the [constructor function](https://www.javascripttutorial.net/javascript-constructor-function/) pattern:

function Person(name) {

this.name = name;

}

Code language: JavaScript (javascript)

Second, create a new object of the Person type:

let p1 = new Person('John Doe');

Code language: JavaScript (javascript)

Third, check if the person is an instance of the Person type:

console.log(p1 instanceof Person); *// true*

Code language: JavaScript (javascript)

It returns true because the Person.prototype appears on the prototype chain of the p1 object. The prototype chain of the p1 is the link between p1, Person.prototype, and Object.prototype:

The following also returns true because the Object.prototype appears on the prototype chain of the p1 object:

console.log(p1 instanceof Object); *// true*

Code language: JavaScript (javascript)

ES6 class and instanceof operator

The following example defines the Person class and uses the instanceof operator to check if an object is an instance of the [class](https://www.javascripttutorial.net/es6/javascript-class/):

class Person {

constructor(name) {

this.name = name;

}

}

let p1 = new Person('John');

console.log(p1 instanceof Person); *// true*

Code language: JavaScript (javascript)

How it works.

First, define the Person class:

class Person {

constructor(name) {

this.name = name;

}

}

Code language: JavaScript (javascript)

Second, create a new instance of the Person class:

let p1 = new Person('John');

Code language: JavaScript (javascript)

Third, check if p1 is an instance of the Person class:

console.log(p1 instanceof Person); *// true*

Code language: JavaScript (javascript)

The instanceof operator and inheritance

The following example defines the Employee class that extends the Person class:

class Person {

constructor(name) {

this.name = name;

}

}

class Employee extends Person {

constructor(name, title) {

super(name);

this.title = title;

}

}

let e1 = new Employee();

console.log(e1 instanceof Employee); *// true*

console.log(e1 instanceof Person); *// true*

console.log(e1 instanceof Object); *// true*

Code language: JavaScript (javascript)

Since e1 is an instance of the Employee class, it’s also an instance of the Person and Object classes (base classes).

Symbol.hasInstance

In ES6, the instanceof operator uses the Symbol.hasInstance function to check the relationship. The Symbol.hasInstance() accepts an object and returns true if a type has that object as an instance. For example:

class Person {

constructor(name) {

this.name = name;

}

}

let p1 = new Person('John');

console.log(Person[Symbol.hasInstance](p1)); *// true*

Code language: JavaScript (javascript)

Since the Symbol.hasInstance is defined on the [Function](https://www.javascripttutorial.net/javascript-function-type/) prototype, it’s automatically available by default in all functions and classes

And you can redefine the Symbol.hasInstance on a subclass as a static method. For example:

class Person {

constructor(name) {

this.name = name;

}

}

class Android extends Person {

static [Symbol.hasInstance]() {

return false;

}

}

let a1 = new Android('Sonny');

console.log(a1 instanceof Android); *// false*

console.log(a1 instanceof Person); *// false*

Code language: JavaScript (javascript)

Summary

* Use the instanceof operator to check if the constructor.protoype in object’s prototype chain.