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 but also have some syntax and semantics that are unique to classes.

**Defining classes**

Classes are in fact "special functions", and just as you can define function expressions and function declarations, a class can be defined in two ways: a class expression or a class declaration.

JS

// 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;

}

};

Like function expressions, class expressions may be anonymous, or have a name that's different from the variable that it's assigned to. However, unlike function declarations, class declarations have the same temporal dead zone restrictions as let or const and behave as if they are not hoisted.

JavaScript is a prototype-based language — an object's behaviors are specified by its own properties and its prototype's properties. However, with the addition of classes, the creation of hierarchies of objects and the inheritance of properties and their values are much more in line with other object-oriented languages such as Java. In this section, we will demonstrate how objects can be created from classes.

**Class body**

The body of a class is the part that is in curly brackets {}. This is where you define class members, such as methods or constructor.

The body of a class is executed in strict mode even without the "use strict" directive.

A class element can be characterized by three aspects:

1. Kind: Getter, setter, method, or field
2. Location: Static or instance
3. Visibility: Public or private

Together, they add up to 16 possible combinations. To divide the reference more logically and avoid overlapping content, the different elements are introduced in detail in different pages:

1. Method definitions

Public instance method

1. getter

Public instance getter

1. setter

Public instance setter

1. Public class fields

Public instance field

1. static

Public static method, getter, setter, and field

1. Private class features

Everything that's private

**Constructor**

The 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 is thrown if the class contains more than one occurrence of a constructor method.

A constructor can use the super keyword to call the constructor of the super class.

You can create instance properties inside the constructor:

JS

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.

**Static initialization blocks**

Static initialization blocks allow flexible initialization of static properties, including the evaluation of statements during initialization, while granting access to the private scope.

Multiple static blocks can be declared, and these can be interleaved with the declaration of static fields and methods (all static items are evaluated in declaration order).

**Methods**

Methods are defined on the prototype of each class instance and are shared by all instances. Methods can be plain functions, async functions, generator functions, or async generator functions. For more information, see method definitions.

JS

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

// Getter

get area() {

return this.calcArea();

}

// Method

calcArea() {

return this.height \* this.width;

}

\*getSides() {

yield this.height;

yield this.width;

yield this.height;

yield this.width;

}

}

const square = new Rectangle(10, 10);

console.log(square.area); // 100

console.log([...square.getSides()]); // [10, 10, 10, 10]

**Static methods and fields**

The static keyword defines a static method or field for a class. Static properties (fields and methods) are defined on the class itself instead of each instance. Static methods are often used to create utility functions for an application, whereas static fields are useful for caches, fixed-configuration, or any other data that don't need to be replicated across instances.

JS

class Point {

constructor(x, y) {

this.x = x;

this.y = y;

}

static displayName = "Point";

static distance(a, b) {

const dx = a.x - b.x;

const dy = a.y - b.y;

return Math.hypot(dx, dy);

}

}

const p1 = new Point(5, 5);

const p2 = new Point(10, 10);

p1.displayName; // undefined

p1.distance; // undefined

p2.displayName; // undefined

p2.distance; // undefined

console.log(Point.displayName); // "Point"

console.log(Point.distance(p1, p2)); // 7.0710678118654755

**Field declarations**

With the class field declaration syntax, the constructor example can be written as:

JS

class Rectangle {

height = 0;

width;

constructor(height, width) {

this.height = height;

this.width = width;

}

}

Class fields are similar to object properties, not variables, so we don't use keywords such as const to declare them. In JavaScript, private features use a special identifier syntax, so modifier keywords like public and private should not be used either.

As seen above, the fields can be declared with or without a default value. Fields without default values default to undefined. By declaring fields up-front, class definitions become more self-documenting, and the fields are always present, which help with optimizations.

**Private class features**

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

JS

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.

**Inheritance**

The extends keyword is used in class declarations or class expressions to create a class as a child of another constructor (either a class or a function).

JS

class Animal {

constructor(name) {

this.name = name;

}

speak() {

console.log(`${this.name} makes a noise.`);

}

}

class Dog extends Animal {

constructor(name) {

super(name); // call the super class constructor and pass in the name parameter

}

speak() {

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

}

}

const d = new Dog("Mitzie");

d.speak(); // Mitzie barks.

If there is a constructor present in the subclass, it needs to first call super() before using this. The super keyword can also be used to call corresponding methods of super class.

JS

class Cat {

constructor(name) {

this.name = name;

}

speak() {

console.log(`${this.name} makes a noise.`);

}

}

class Lion extends Cat {

speak() {

super.speak();

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

}

}

const l = new Lion("Fuzzy");

l.speak();

// Fuzzy makes a noise.

// Fuzzy roars.

**Evaluation order**

When a class declaration or class expression is evaluated, its various components are evaluated in the following order:

The extends clause, if present, is first evaluated. It must evaluate to a valid constructor function or null, or a TypeError is thrown.

The constructor method is extracted, substituted with a default implementation if constructor is not present. However, because the constructor definition is only a method definition, this step is not observable.

The class elements' property keys are evaluated in the order of declaration. If the property key is computed, the computed expression is evaluated, with the this value set to the this value surrounding the class (not the class itself). None of the property values are evaluated yet.

Methods and accessors are installed in the order of declaration. Instance methods and accessors are installed on the prototype property of the current class, and static methods and accessors are installed on the class itself. Private instance methods and accessors are saved to be installed on the instance directly later. This step is not observable.

The class is now initialized with the prototype specified by extends and implementation specified by constructor. For all steps above, if an evaluated expression tries to access the name of the class, a ReferenceError is thrown because the class is not initialized yet.

The class elements' values are evaluated in the order of declaration:

For each instance field (public or private), its initializer expression is saved. The initializer is evaluated during instance creation, at the start of the constructor (for base classes) or immediately before the super() call returns (for derived classes).

For each static field (public or private), its initializer is evaluated with this set to the class itself, and the property is created on the class.

Static initialization blocks are evaluated with this set to the class itself.

The class is now fully initialized and can be used as a constructor function.

For how instances are created, see the constructor reference.

**Examples**

Binding this with instance and static methods

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. This behavior is the same even if the "use strict" directive isn't present, because code within the class body is always executed in strict mode.

JS

class Animal {

speak() {

return this;

}

static eat() {

return this;

}

}

const obj = new Animal();

obj.speak(); // the Animal object

const speak = obj.speak;

speak(); // undefined

Animal.eat(); // class Animal

const eat = Animal.eat;

eat(); // undefined

If we rewrite the above using traditional function-based syntax in non–strict mode, then this method calls are automatically bound to globalThis. In strict mode, the value of this remains as undefined.

JS

function Animal() {}

Animal.prototype.speak = function () {

return this;

};

Animal.eat = function () {

return this;

};

const obj = new Animal();

const speak = obj.speak;

speak(); // global object (in non–strict mode)

const eat = Animal.eat;

eat(); // global object (in non-strict mode)

**private fields in classes**

