



Classes

English ▼

JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax *does not* introduce a new object-oriented inheritance model to JavaScript.

Defining classes

Classes are in fact "special [functions](#)", and just as you can define [function expressions](#) and [function declarations](#), the class syntax has two components: [class expressions](#) and [class declarations](#).

Class declarations

One way to define a class is using a **class declaration**. To declare a class, you use the `class` keyword with the name of the class ("Rectangle" here).

```
1  class Rectangle {  
2    constructor(height, width) {  
3      this.height = height;  
4      this.width = width;  
5    }  
6  }
```

Hoisting

An important difference between **function declarations** and **class declarations** is that function declarations are **hoisted** and class declarations are not. You first need to declare your class and then access it, otherwise code like the following will throw a **ReferenceError**:


```
1 | const p = new Rectangle(); // ReferenceError
2 |
3 | class Rectangle {}
```



Class expressions

A **class expression** is another way to define a class. Class expressions can be named or unnamed. The name given to a named class expression is local to the class's body. (it can be retrieved through the class's (not an instance's) **name** property, though).

```
1 | // unnamed
2 | let Rectangle = class {
3 |     constructor(height, width) {
4 |         this.height = height;
5 |         this.width = width;
6 |     }
7 | };
8 | console.log(Rectangle.name);
9 | // output: "Rectangle"
10 |
11 | // named
12 | let Rectangle = class Rectangle2 {
13 |     constructor(height, width) {
14 |         this.height = height;
15 |         this.width = width;
16 |     }
17 | };
18 | console.log(Rectangle.name);
19 | // output: "Rectangle2"
```

 **Note:** Class **expressions** are subject to the same hoisting restrictions as described in the [Class declarations](#) section.

Class body and method definitions

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.

Strict mode

The body of a class is executed in [strict mode](#), i.e., code written here is subject to stricter syntax for increased performance, some otherwise silent errors will be thrown, and certain keywords are reserved for future versions of ECMAScript.

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](#) will be 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.

Prototype methods

See also [method definitions](#).

```
1  class Rectangle {
2    constructor(height, width) {
3      this.height = height;
4      this.width = width;
5    }
   // Getter
```

```

6   get area() {
7       return this.calcArea();
8   }
9   // Method
10  calcArea() {
11      return this.height * this.width;
12  }
13 }
14
15 const square = new Rectangle(10, 10);
16
17 console.log(square.area); // 100
18

```

Static methods

The `static` keyword defines a static method for a class. Static methods are called without [instantiating](#) their class and **cannot** be called through a class instance. Static methods are often used to create utility functions for an application.

```

1   class Point {
2       constructor(x, y) {
3           this.x = x;
4           this.y = y;
5       }
6
7       static distance(a, b) {
8           const dx = a.x - b.x;
9           const dy = a.y - b.y;
10
11          return Math.hypot(dx, dy);
12      }
13  }
14
15  const p1 = new Point(5, 5);
16  const p2 = new Point(10, 10);
17  p1.distance; //undefined
18  p2.distance; //undefined
19

```

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

Boxing with prototype and static methods

When a static or prototype method is called without a value for *this*, the *this* value will be `undefined` inside the method. This behavior will be the same even if the `"use strict"` directive isn't present, because code within the `class` body's syntactic boundary is always executed in strict mode.

```
1  class Animal {
2    speak() {
3      return this;
4    }
5    static eat() {
6      return this;
7    }
8  }
9
10 let obj = new Animal();
11 obj.speak(); // the Animal object
12 let speak = obj.speak;
13 speak(); // undefined
14
15 Animal.eat() // class Animal
16 let eat = Animal.eat;
17 eat(); // undefined
```

If the above is written using traditional function-based syntax, then autoboxing in method calls will happen in non-strict mode based on the initial *this* value. If the initial value is `undefined`, *this* will be set to the global object.

Autoboxing will not happen in strict mode, the *this* value remains as passed.

```
1  function Animal() { }
2
3  Animal.prototype.speak = function() {
```

```

4     return this;
5 }
6
7 Animal.eat = function() {
8     return this;
9 }
10
11 let obj = new Animal();
12 let speak = obj.speak;
13 speak(); // global object
14
15 let eat = Animal.eat;
16 eat(); // global object

```

Instance properties

Instance properties must be defined inside of class methods:

```

1 class Rectangle {
2     constructor(height, width) {
3         this.height = height;
4         this.width = width;
5     }
6 }

```

Static (class-side) data properties and prototype data properties must be defined outside of the ClassBody declaration:

```

1 Rectangle.staticWidth = 20;
2 Rectangle.prototype.prototypeWidth = 25;

```

Field declarations

❗ Public and private field declarations are an [experimental feature \(stage 3\)](#) proposed at TC39, the JavaScript standards committee. Support in browsers is limited, but the feature can be

used through a build step with systems like [Babel](#).

Public field declarations

With the JavaScript field declaration syntax, the above example can be written as:

```
1 class Rectangle {  
2   height = 0;  
3   width;  
4   constructor(height, width) {  
5     this.height = height;  
6     this.width = width;  
7   }  
8 }
```

By declaring fields up-front, class definitions become more self-documenting, and the fields are always present.

As seen above, the fields can be declared with or without a default value.

Private field declarations

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

```
1 class Rectangle {  
2   #height = 0;  
3   #width;  
4   constructor(height, width) {  
5     this.#height = height;  
6     this.#width = width;  
7   }  
8 }
```

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 which are not visible outside of the class, you ensure that your classes' users can't depend on internals, which may change version to version.



Private fields can only be declared up-front in a field declaration.

Private fields cannot be created later through assigning to them, the way that normal properties can.

For more information, see also [class fields](#).

Sub classing with `extends`

The `extends` keyword is used in *class declarations* or *class expressions* to create a class as a child of another class.

```
1  class Animal {
2    constructor(name) {
3      this.name = name;
4    }
5
6    speak() {
7      console.log(`${this.name} makes a noise.`);
8    }
9  }
10
11  class Dog extends Animal {
12    constructor(name) {
13      super(name); // call the super class constructor and pass in the name
14    }
15
16    speak() {
17      console.log(`${this.name} barks.`);
18    }
19  }
20
21  let d = new Dog('Mitzie');
22  d.speak(); // Mitzie barks.
```


If there is a constructor present in the subclass, it needs to first call `super()` before using "this".

One may also extend traditional function-based "classes":

```
function Animal (name) {
  this.name = name;
}

Animal.prototype.speak = function () {
  console.log(`${this.name} makes a noise.`);
}

class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks.`);
  }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.

//NB: For similar methods, the child's method takes precedence over parent's met
```

Note that classes cannot extend regular (non-constructible) objects. If you want to inherit from a regular object, you can instead use `Object.setPrototypeOf()`:

```
1  const Animal = {
2    speak() {
3      console.log(`${this.name} makes a noise.`);
4    }
5  };
6
7  class Dog {
8    constructor(name) {
9      this.name = name;
10   }
11 }
12
```

```
13 // If you do not do this you will get a TypeError when you invoke speak
14 Object.setPrototypeOf(Dog.prototype, Animal);
15
16 let d = new Dog('Mitzie');
17 d.speak(); // Mitzie makes a noise.
```

Species

You might want to return `Array` objects in your derived array class `MyArray`. The species pattern lets you override default constructors.

For example, when using methods such as `map()` that returns the default constructor, you want these methods to return a parent `Array` object, instead of the `MyArray` object. The `Symbol.species` symbol lets you do this:

```
1 class MyArray extends Array {
2   // Overwrite species to the parent Array constructor
3   static get [Symbol.species]() { return Array; }
4 }
5
6 let a = new MyArray(1,2,3);
7 let mapped = a.map(x => x * x);
8
9 console.log(mapped instanceof MyArray); // false
10 console.log(mapped instanceof Array);   // true
```

Super class calls with `super`

The `super` keyword is used to call corresponding methods of super class. This is one advantage over prototype-based inheritance.

```
1 class Cat {
2   constructor(name) {
3     this.name = name;
4   }
5
6   speak() {
7     console.log(`${this.name} makes a noise.`);
8   }
9 }
10
11 class Lion extends Cat {
12   speak() {
13     super.speak();
14     console.log(`${this.name} roars.`);
15   }
16 }
17
18 let l = new Lion('Fuzzy');
19 l.speak();
20 // Fuzzy makes a noise.
21 // Fuzzy roars.
```

Mix-ins

Abstract subclasses or *mix-ins* are templates for classes. An ECMAScript class can only have a single superclass, so multiple inheritance from tooling classes, for example, is not possible. The functionality must be provided by the superclass.

A function with a superclass as input and a subclass extending that superclass as output can be used to implement mix-ins in ECMAScript:

```
1 let calculatorMixin = Base => class extends Base {
2   calc() { }
3 };
4
```

```
5 | let randomizerMixin = Base => class extends Base {  
6 |     randomize() { }  
7 | };
```

A class that uses these mix-ins can then be written like this:

```
1 | class Foo { }  
2 | class Bar extends calculatorMixin(randomizerMixin(Foo)) { }
```

Specifications

Specification

ECMAScript Latest Draft (ECMA-262)

The definition of 'Class definitions' in that specification.

Browser compatibility

[Update compatibility data on GitHub](#)

classes	
Chrome	49★ <div></div>
Edge	13
Firefox	45
IE	No <div></div>
Opera	36★ <div></div>
Safari	9
WebView Android	49★ <div></div>

Chrome Android	49★	▼
Firefox Android	45	
Opera Android	36★	▼
Safari iOS	9	
Samsung Internet Android	5.0★	▼
nodejs	6.0.0	▼
constructor		
Chrome	49★	▼
Edge	13	
Firefox	45	
IE	No	✗
Opera	36★	▼
Safari	9	
WebView Android	49★	▼
Chrome Android	49★	▼
Firefox Android	45	
Opera Android	36★	▼
Safari iOS	9	
Samsung Internet Android	5.0	
nodejs	6.0.0	▼
extends		
Chrome	49★	▼
Edge	13	
Firefox	45	
IE	No	✗
Opera	36★	▼

Safari	9
WebView Android	49★
Chrome Android	49★
Firefox Android	45
Opera Android	36★
Safari iOS	9
Samsung Internet Android	5.0
nodejs	6.0.0
Private class fields	
Chrome	74
Edge	79
Firefox	No
IE	No
Opera	62
Safari	No
WebView Android	74
Chrome Android	74
Firefox Android	No
Opera Android	53
Safari iOS	No
Samsung Internet Android	No
nodejs	12.0.0
Public class fields	
Chrome	72
Edge	79
Firefox	69

IE	No
Opera	60
Safari	No
WebView Android	72
Chrome Android	72
Firefox Android	No
Opera Android	51
Safari iOS	No
Samsung Internet Android	No
nodejs	12.0.0
static	
Chrome	49★
Edge	13
Firefox	45
IE	No
Opera	36★
Safari	9
WebView Android	49★
Chrome Android	49★
Firefox Android	45
Opera Android	36★
Safari iOS	9
Samsung Internet Android	5.0
nodejs	6.0.0
Static class fields	
Chrome	72

Edge	79
Firefox	75
IE	No
Opera	60
Safari	No
WebView Android	72
Chrome Android	72
Firefox Android	No
Opera Android	51
Safari iOS	No
Samsung Internet Android	No
nodejs	12.0.0

What are we missing?



Full support



No support



See implementation notes.



User must explicitly enable this feature.

Re-running a class definition

A class can't be redefined. Attempting to do so produces a `SyntaxError`.

If you're experimenting with code in a web browser, such as the Firefox Web Console (**Tools** > **Web Developer** > **Web Console**) and you 'Run' a definition of a class with the same name twice, you'll get a `SyntaxError: redeclaration of let ClassName;`. (See further

discussion of this issue in [bug 1428672](#).) Doing something similar in Chrome Developer Tools gives you a message like `Uncaught SyntaxError: Identifier 'ClassName' has already been declared at <anonymous>:1:1`.

See also

- [Functions](#)
 - [class](#) declaration
 - [class](#) expression
 - [Class elements](#)
 - [super](#)
 - [Blog post: "ES6 In Depth: Classes"](#)
 - [Fields and public/private class properties proposal \(stage 3\)](#)
-

 **Last modified:** Mar 13, 2020, by [MDN contributors](#)

[Defining classes](#)

[Class body and method definitions](#)

[Sub classing with `extends`](#)

[Species](#)

[Super class calls with `super`](#)

[Mix-ins](#)

[Specifications](#)

[Browser compatibility](#)

[Re-running a class definition](#)

[See also](#)

Related Topics

JavaScript

Tutorials:

- ▶ [Complete beginners](#)
- ▶ [JavaScript Guide](#)
- ▶ [Intermediate](#)
- ▶ [Advanced](#)

References:

- ▶ [Built-in objects](#)
- ▶ [Expressions & operators](#)
- ▶ [Statements & declarations](#)
- ▶ [Functions](#)
- ▼ [Classes](#)
 - [Class fields](#)
 - [constructor](#)
 - [extends](#)
 - [static](#)
- ▶ [Errors](#)
- ▶ [Misc](#)

Learn the best of web development

Get the latest and greatest from MDN delivered straight to your inbox.

Sign up now