Classes provide "super" keyword for that.

* super.method(...) to call a parent method.
* super(...) to call a parent constructor (inside our constructor only).

For instance, let our rabbit autohide when stopped:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

run(speed) {

this.speed = speed;

alert(`${this.name} runs with speed ${this.speed}.`);

}

stop() {

this.speed = 0;

alert(`${this.name} stands still.`);

}

}

class Rabbit extends Animal {

hide() {

alert(`${this.name} hides!`);

}

stop() {

super.stop(); // call parent stop

this.hide(); // and then hide

}

}

let rabbit = new Rabbit("White Rabbit");

rabbit.run(5); // White Rabbit runs with speed 5.

rabbit.stop(); // White Rabbit stands still. White rabbit hides!

Now Rabbit has the stop method that calls the parent super.stop() in the process.

**Arrow functions have no super**

Arrow functions do not have super.

If accessed, it’s taken from the outer function. For instance:

class Rabbit extends Animal {

stop() {

setTimeout(() => super.stop(), 1000); // call parent stop after 1sec

}

}

The super in the arrow function is the same as in stop(), so it works as intended. If we specified a “regular” function here, there would be an error:

// Unexpected super

setTimeout(function() { super.stop() }, 1000);

**Overriding constructor**

With constructors it gets a little bit tricky.

Until now, Rabbit did not have its own constructor.

If a class extends another class and has no constructor, then the following “empty” constructor is generated:

class Rabbit extends Animal {

// generated for extending classes without own constructors

constructor(...args) {

super(...args);

}

}

As we can see, it basically calls the parent constructor passing it all the arguments. That happens if we don’t write a constructor of our own.

Now let’s add a custom constructor to Rabbit. It will specify the earLength in addition to name:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

// ...

}

class Rabbit extends Animal {

constructor(name, earLength) {

this.speed = 0;

this.name = name;

this.earLength = earLength;

}

// ...

}

// Doesn't work!

let rabbit = new Rabbit("White Rabbit", 10); // Error: this is not defined.

Whoops! We’ve got an error. Now we can’t create rabbits. What went wrong?

The short answer is:

* **Constructors in inheriting classes must call super(...), and (!) do it before using this.**

…But why? What’s going on here? Indeed, the requirement seems strange.

Of course, there’s an explanation. Let’s get into details, so you’ll really understand what’s going on.

In JavaScript, there’s a distinction between a constructor function of an inheriting class (so-called “derived constructor”) and other functions. A derived constructor has a special internal property [[ConstructorKind]]:"derived". That’s a special internal label.

That label affects its behavior with new.

* When a regular function is executed with new, it creates an empty object and assigns it to this.
* But when a derived constructor runs, it doesn’t do this. It expects the parent constructor to do this job.

So a derived constructor must call super in order to execute its parent (base) constructor, otherwise the object for this won’t be created. And we’ll get an error.

For the Rabbit constructor to work, it needs to call super() before using this, like here:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

// ...

}

class Rabbit extends Animal {

constructor(name, earLength) {

super(name);

this.earLength = earLength;

}

// ...

}

// now fine

let rabbit = new Rabbit("White Rabbit", 10);

alert(rabbit.name); // White Rabbit

alert(rabbit.earLength); // 10

**Overriding class fields: a tricky note**

**Advanced note**

This note assumes you have a certain experience with classes, maybe in other programming languages.

It provides better insight into the language and also explains the behavior that might be a source of bugs (but not very often).

If you find it difficult to understand, just go on, continue reading, then return to it some time later.

We can override not only methods, but also class fields.

Although, there’s a tricky behavior when we access an overridden field in parent constructor, quite different from most other programming languages.

Consider this example:

class Animal {

name = 'animal'

constructor() {

alert(this.name); // (\*)

}

}

class Rabbit extends Animal {

name = 'rabbit';

}

new Animal(); // animal

new Rabbit(); // animal

Here, class Rabbit extends Animal and overrides name field with its own value.

There’s no own constructor in Rabbit, so Animal constructor is called.

What’s interesting is that in both cases: new Animal() and new Rabbit(), the alert in the line (\*) shows animal.

**In other words, parent constructor always uses its own field value, not the overridden one.**

What’s odd about it?

If it’s not clear yet, please compare with methods.

Here’s the same code, but instead of this.name field we call this.showName() method:

class Animal {

showName() { // instead of this.name = 'animal'

alert('animal');

}

constructor() {

this.showName(); // instead of alert(this.name);

}

}

class Rabbit extends Animal {

showName() {

alert('rabbit');

}

}

new Animal(); // animal

new Rabbit(); // rabbit

Please note: now the output is different.

And that’s what we naturally expect. When the parent constructor is called in the derived class, it uses the overridden method.

…But for class fields it’s not so. As said, the parent constructor always uses the parent field.

Why is there the difference?

Well, the reason is in the field initialization order. The class field is initialized:

* Before constructor for the base class (that doesn’t extend anything),
* Imediately after super() for the derived class.

In our case, Rabbit is the derived class. There’s no constructor() in it. As said previously, that’s the same as if there was an empty constructor with only super(...args).

So, new Rabbit() calls super(), thus executing the parent constructor, and (per the rule for derived classes) only after that its class fields are initialized. At the time of the parent constructor execution, there are no Rabbit class fields yet, that’s why Animal fields are used.

This subtle difference between fields and methods is specific to JavaScript

Luckily, this behavior only reveals itself if an overridden field is used in the parent constructor. Then it may be difficult to understand what’s going on, so we’re explaining it here.

If it becomes a problem, one can fix it by using methods or getters/setters instead of fields.