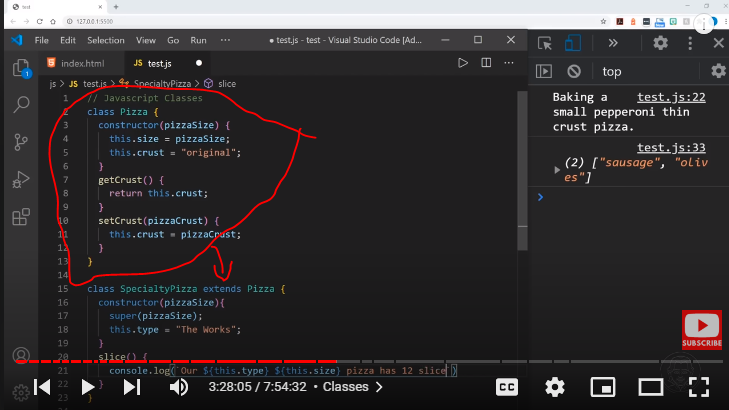
Inheritance

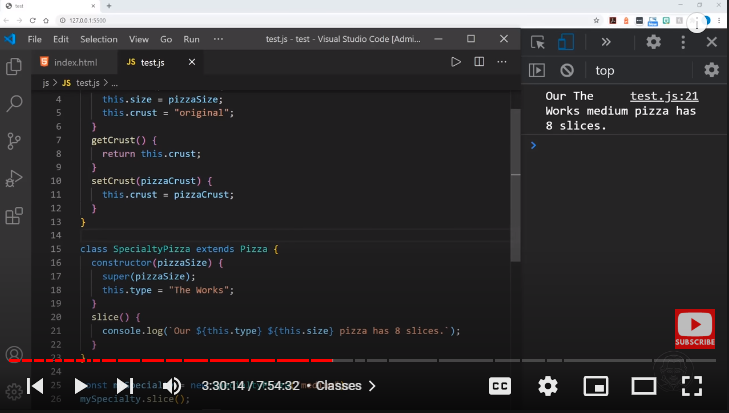
Link for more : <https://javascript.info/class-inheritance>

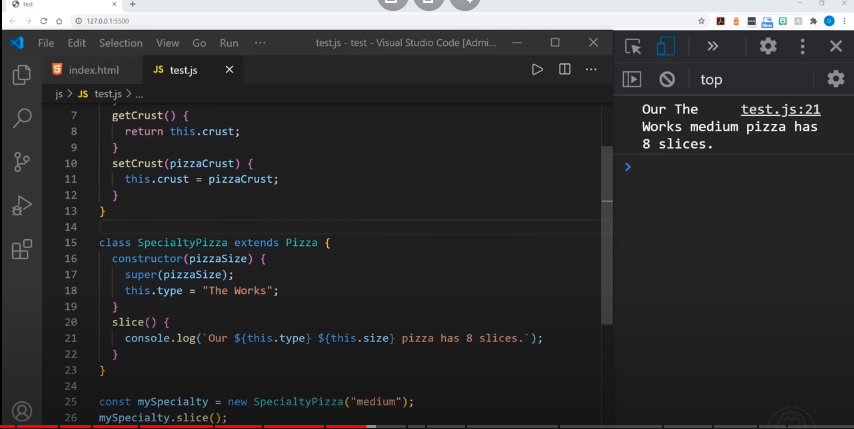
Class inheritance is a way for one class to extend another class.

So we can create new functionality on top of the existing.

The “extends” keyword







Let’s say we have class Animal:

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;

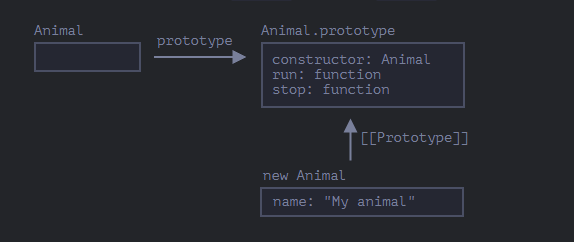
alert(`${this.name} stands still.`);

}

}

let animal = new Animal("My animal");

Here’s how we can represent animal object and Animal class graphically:



…And we would like to create another class Rabbit.

As rabbits are animals, Rabbit class should be based on Animal, have access to animal methods, so that rabbits can do what “generic” animals can do.

The syntax to extend another class is: class Child extends Parent.

Let’s create class Rabbit that inherits from Animal:

class Rabbit extends Animal {

hide() {

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

}

}

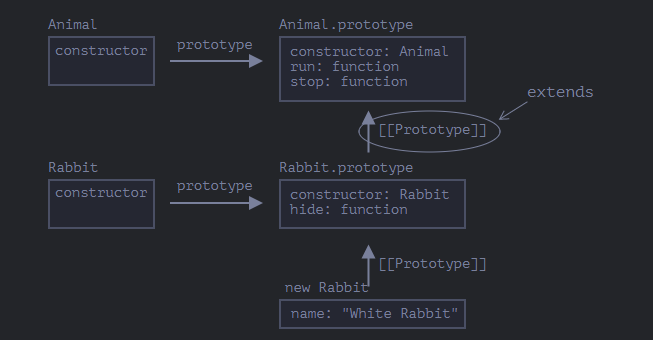
let rabbit = new Rabbit("White Rabbit");

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

rabbit.hide(); // White Rabbit hides!

Object of Rabbit class have access both to Rabbit methods, such as rabbit.hide(), and also to Animal methods, such as rabbit.run().

Internally, extends keyword works using the good old prototype mechanics. It sets Rabbit.prototype.[[Prototype]] to Animal.prototype. So, if a method is not found in Rabbit.prototype, JavaScript takes it from Animal.prototype.



For instance, to find rabbit.run method, the engine checks (bottom-up on the picture):

The rabbit object (has no run).

Its prototype, that is Rabbit.prototype (has hide, but not run).

Its prototype, that is (due to extends) Animal.prototype, that finally has the run method.

As we can recall from the chapter Native prototypes, JavaScript itself uses prototypal inheritance for built-in objects. E.g. Date.prototype.[[Prototype]] is Object.prototype. That’s why dates have access to generic object methods.

Any expression is allowed after extends

Class syntax allows to specify not just a class, but any expression after extends.

For instance, a function call that generates the parent class:

function f(phrase) {

return class {

sayHi() { alert(phrase); }

};

}

class User extends f("Hello") {}

new User().sayHi(); // Hello

Here class User inherits from the result of f("Hello").

That may be useful for advanced programming patterns when we use functions to generate classes depending on many conditions and can inherit from them.

**Overriding a method**

Now let’s move forward and override a method. By default, all methods that are not specified in class Rabbit are taken directly “as is” from class Animal.

But if we specify our own method in Rabbit, such as stop() then it will be used instead:

class Rabbit extends Animal {

stop() {

// ...now this will be used for rabbit.stop()

// instead of stop() from class Animal

}

}

Usually, however, we don’t want to totally replace a parent method, but rather to build on top of it to tweak or extend its functionality. We do something in our method, but call the parent method before/after it or in the process.

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**

As was mentioned in the chapter Arrow functions revisited, 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.

According to the specification, 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 the 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, the 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 a difference?

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

Before constructor for the base class (that doesn’t extend anything),

Immediately 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.

Super: internals, [[HomeObject]]

Advanced information

If you’re reading the tutorial for the first time – this section may be skipped.

It’s about the internal mechanisms behind inheritance and super.

Let’s get a little deeper under the hood of super. We’ll see some interesting things along the way.

First to say, from all that we’ve learned till now, it’s impossible for super to work at all!

Yeah, indeed, let’s ask ourselves, how it should technically work? When an object method runs, it gets the current object as this. If we call super.method() then, the engine needs to get the method from the prototype of the current object. But how?

The task may seem simple, but it isn’t. The engine knows the current object this, so it could get the parent method as this.\_\_proto\_\_.method. Unfortunately, such a “naive” solution won’t work.

Let’s demonstrate the problem. Without classes, using plain objects for the sake of simplicity.

You may skip this part and go below to the [[HomeObject]] subsection if you don’t want to know the details. That won’t harm. Or read on if you’re interested in understanding things in-depth.

In the example below, rabbit.\_\_proto\_\_ = animal. Now let’s try: in rabbit.eat() we’ll call animal.eat(), using this.\_\_proto\_\_:

let animal = {

name: "Animal",

eat() {

alert(`${this.name} eats.`);

}

};

let rabbit = {

\_\_proto\_\_: animal,

name: "Rabbit",

eat() {

// that's how super.eat() could presumably work

this.\_\_proto\_\_.eat.call(this); // (\*)

}

};

rabbit.eat(); // Rabbit eats.

At the line (\*) we take eat from the prototype (animal) and call it in the context of the current object. Please note that .call(this) is important here, because a simple this.\_\_proto\_\_.eat() would execute parent eat in the context of the prototype, not the current object.

And in the code above it actually works as intended: we have the correct alert.

Now let’s add one more object to the chain. We’ll see how things break:

let animal = {

name: "Animal",

eat() {

alert(`${this.name} eats.`);

}

};

let rabbit = {

\_\_proto\_\_: animal,

eat() {

// ...bounce around rabbit-style and call parent (animal) method

this.\_\_proto\_\_.eat.call(this); // (\*)

}

};

let longEar = {

\_\_proto\_\_: rabbit,

eat() {

// ...do something with long ears and call parent (rabbit) method

this.\_\_proto\_\_.eat.call(this); // (\*\*)

}

};

longEar.eat(); // Error: Maximum call stack size exceeded

The code doesn’t work anymore! We can see the error trying to call longEar.eat().

It may be not that obvious, but if we trace longEar.eat() call, then we can see why. In both lines (\*) and (\*\*) the value of this is the current object (longEar). That’s essential: all object methods get the current object as this, not a prototype or something.

So, in both lines (\*) and (\*\*) the value of this.\_\_proto\_\_ is exactly the same: rabbit. They both call rabbit.eat without going up the chain in the endless loop.

Here’s the picture of what happens:

Inside longEar.eat(), the line (\*\*) calls rabbit.eat providing it with this=longEar.

// inside longEar.eat() we have this = longEar

this.\_\_proto\_\_.eat.call(this) // (\*\*)

// becomes

longEar.\_\_proto\_\_.eat.call(this)

// that is

rabbit.eat.call(this);

Then in the line (\*) of rabbit.eat, we’d like to pass the call even higher in the chain, but this=longEar, so this.\_\_proto\_\_.eat is again rabbit.eat!

// inside rabbit.eat() we also have this = longEar

this.\_\_proto\_\_.eat.call(this) // (\*)

// becomes

longEar.\_\_proto\_\_.eat.call(this)

// or (again)

rabbit.eat.call(this);

…So rabbit.eat calls itself in the endless loop, because it can’t ascend any further.

The problem can’t be solved by using this alone.

[[HomeObject]]

To provide the solution, JavaScript adds one more special internal property for functions: [[HomeObject]].

When a function is specified as a class or object method, its [[HomeObject]] property becomes that object.

Then super uses it to resolve the parent prototype and its methods.

Let’s see how it works, first with plain objects:

let animal = {

name: "Animal",

eat() { // animal.eat.[[HomeObject]] == animal

alert(`${this.name} eats.`);

}

};

let rabbit = {

\_\_proto\_\_: animal,

name: "Rabbit",

eat() { // rabbit.eat.[[HomeObject]] == rabbit

super.eat();

}

};

let longEar = {

\_\_proto\_\_: rabbit,

name: "Long Ear",

eat() { // longEar.eat.[[HomeObject]] == longEar

super.eat();

}

};

// works correctly

longEar.eat(); // Long Ear eats.

It works as intended, due to [[HomeObject]] mechanics. A method, such as longEar.eat, knows its [[HomeObject]] and takes the parent method from its prototype. Without any use of this.

Methods are not “free”

As we’ve known before, generally functions are “free”, not bound to objects in JavaScript. So they can be copied between objects and called with another this.

The very existence of [[HomeObject]] violates that principle, because methods remember their objects. [[HomeObject]] can’t be changed, so this bond is forever.

The only place in the language where [[HomeObject]] is used – is super. So, if a method does not use super, then we can still consider it free and copy between objects. But with super things may go wrong.

Here’s the demo of a wrong super result after copying:

let animal = {

sayHi() {

alert(`I'm an animal`);

}

};

// rabbit inherits from animal

let rabbit = {

\_\_proto\_\_: animal,

sayHi() {

super.sayHi();

}

};

let plant = {

sayHi() {

alert("I'm a plant");

}

};

// tree inherits from plant

let tree = {

\_\_proto\_\_: plant,

sayHi: rabbit.sayHi // (\*)

};

tree.sayHi(); // I'm an animal (?!?)

A call to tree.sayHi() shows “I’m an animal”. Definitely wrong.

The reason is simple:

In the line (\*), the method tree.sayHi was copied from rabbit. Maybe we just wanted to avoid code duplication?

Its [[HomeObject]] is rabbit, as it was created in rabbit. There’s no way to change [[HomeObject]].

The code of tree.sayHi() has super.sayHi() inside. It goes up from rabbit and takes the method from animal.

Here’s the diagram of what happens:

Methods, not function properties

[[HomeObject]] is defined for methods both in classes and in plain objects. But for objects, methods must be specified exactly as method(), not as "method: function()".

The difference may be non-essential for us, but it’s important for JavaScript.

In the example below a non-method syntax is used for comparison. [[HomeObject]] property is not set and the inheritance doesn’t work:

let animal = {

eat: function() { // intentionally writing like this instead of eat() {...

// ...

}

};

let rabbit = {

\_\_proto\_\_: animal,

eat: function() {

super.eat();

}

};

rabbit.eat(); // Error calling super (because there's no [[HomeObject]])