**Class Kinds, Location, and Visibility**

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

**Kind**

**Method definitions**

Method definition is a shorter syntax for defining a function property in an object initializer. It can also be used in classes.

**get**

The get syntax binds an object property to a function that will be called when that property is looked up. It can also be used in classes.

const obj = {

log: ['a', 'b', 'c'],

get latest() {

return this.log[this.log.length - 1];

}

};

console.log(obj.latest);

// Expected output: "c"

Syntax

JS

{ get prop() { /\* … \*/ } }

{ get [expression]() { /\* … \*/ } }

**Parameters**

1. prop

The name of the property to bind to the given function. In the same way as other properties in object initializers, it can be a string literal, a number literal, or an identifier.

1. expression

You can also use expressions for a computed property name to bind to the given function.

Examples

Defining a getter on new objects in object initializers

This will create a pseudo-property latest for object obj, which will return the last array item in log.

JS

const obj = {

log: ["example", "test"],

get latest() {

if (this.log.length === 0) return undefined;

return this.log[this.log.length - 1];

},

};

console.log(obj.latest); // "test"

**Using getters in classes**

You can use the exact same syntax to define public instance getters that are available on class instances. In classes, you don't need the comma separator between methods.

JS

class ClassWithGetSet {

#msg = "hello world";

get msg() {

return this.#msg;

}

set msg(x) {

this.#msg = `hello ${x}`;

}

}

const instance = new ClassWithGetSet();

console.log(instance.msg); // "hello world"

instance.msg = "cake";

console.log(instance.msg); // "hello cake"

Getter properties are defined on the prototype property of the class and are thus shared by all instances of the class. Unlike getter properties in object literals, getter properties in classes are not enumerable.

Static setters and private setters use similar syntaxes, which are described in the static and private class features pages.

**Deleting a getter using the delete operator**

If you want to remove the getter, you can just delete it:

JS

delete obj.latest;

**set**

The set syntax binds an object property to a function to be called when there is an attempt to set that property. It can also be used in classes.

Syntax

JS

{ set prop(val) { /\* … \*/ } }

{ set [expression](val) { /\* … \*/ } }

There are some additional syntax restrictions:

A setter must have exactly one parameter.

**Parameters**

1. prop

The name of the property to bind to the given function. In the same way as other properties in object initializers, it can be a string literal, a number literal, or an identifier.

1. val

An alias for the variable that holds the value attempted to be assigned to prop.

1. expression

You can also use expressions for a computed property name to bind to the given function.

In JavaScript, a setter can be used to execute a function whenever a specified property is attempted to be changed. Setters are most often used in conjunction with getters to create a type of pseudo-property. It is not possible to simultaneously have a setter on a property that holds an actual value.

**Examples**

Defining a setter on new objects in object initializers

The following example define a pseudo-property current of object language. When current is assigned a value, it updates log with that value:

JS

const language = {

set current(name) {

this.log.push(name);

},

log: [],

};

language.current = "EN";

console.log(language.log); // ['EN']

language.current = "FA";

console.log(language.log); // ['EN', 'FA']

Note that current is not defined, and any attempts to access it will result in undefined.

**Using setters in classes**

You can use the exact same syntax to define public instance setters that are available on class instances. In classes, you don't need the comma separator between methods.

JS

class ClassWithGetSet {

#msg = "hello world";

get msg() {

return this.#msg;

}

set msg(x) {

this.#msg = `hello ${x}`;

}

}

const instance = new ClassWithGetSet();

console.log(instance.msg); // "hello world"

instance.msg = "cake";

console.log(instance.msg); // "hello cake"

Setter properties are defined on the prototype property of the class and are thus shared by all instances of the class. Unlike setter properties in object literals, setter properties in classes are not enumerable.

Static setters and private setters use similar syntaxes, which are described in the static and private class features pages.

Removing a setter with the delete operator

If you want to remove the setter, you can just delete it:

JS

delete language.current;

Defining a setter on existing objects using defineProperty

To append a setter to an existing object, use Object.defineProperty().

JS

Copy to Clipboard

const o = { a: 0 };

Object.defineProperty(o, "b", {

set(x) {

this.a = x / 2;

},

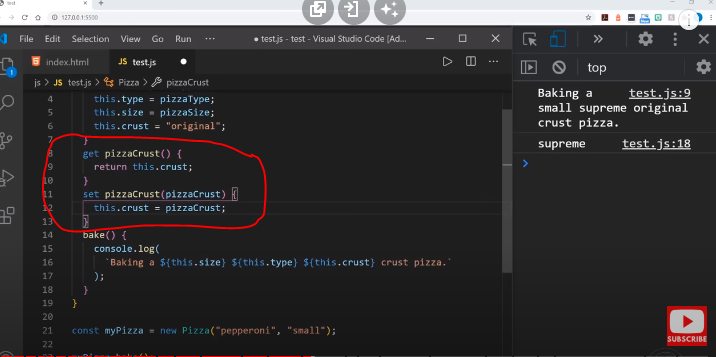
});

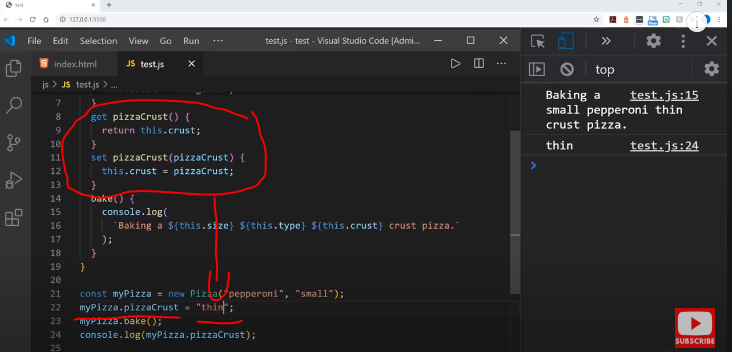
o.b = 10;

// Runs the setter, which assigns 10 / 2 (5) to the 'a' property

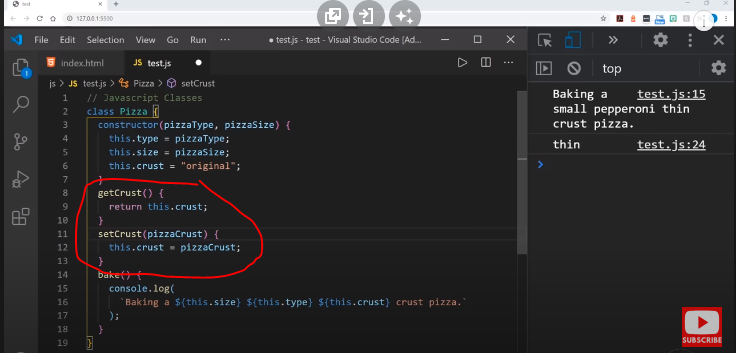
console.log(o.a); // 5

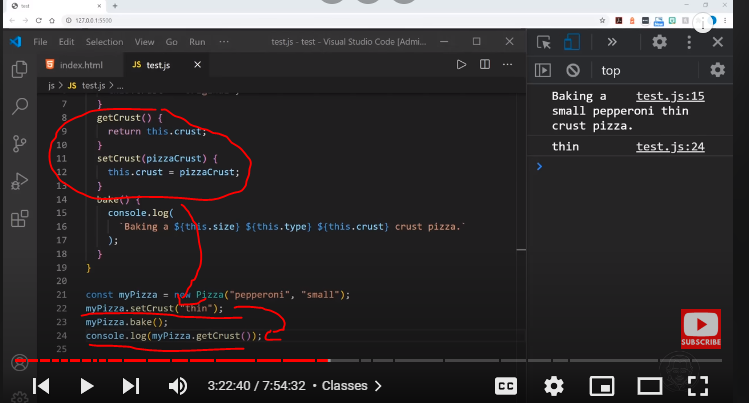
**getter and setter example:**



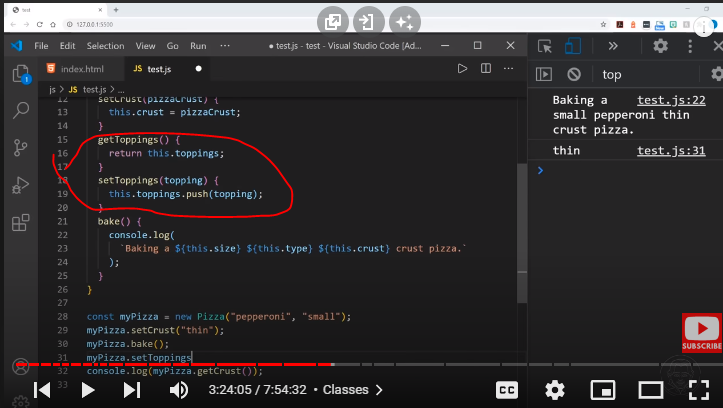


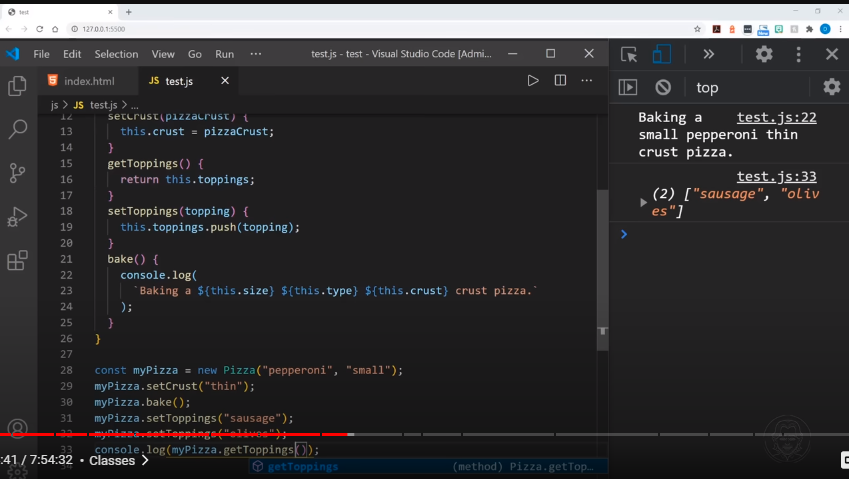
**Getter and setter (using method alternate technique)**

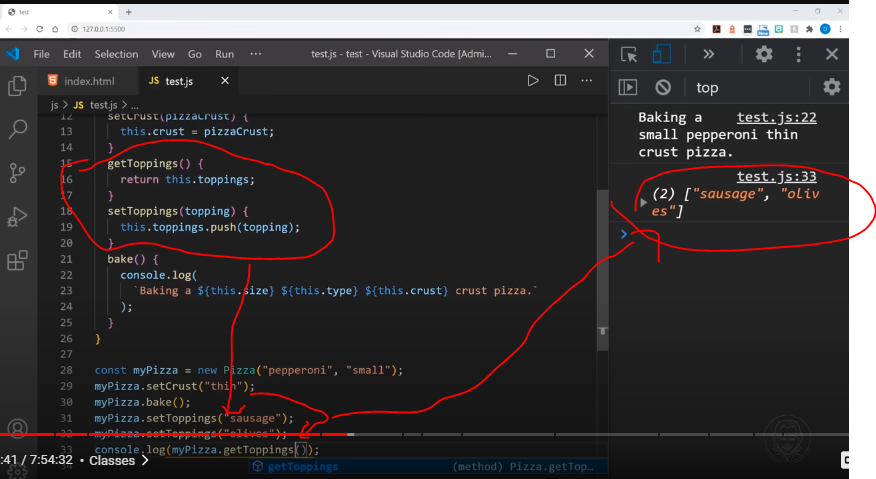




**Another example:**







**Public class fields**

Public fields are writable, enumerable, and configurable properties. As such, unlike their private counterparts, they participate in prototype inheritance.

Syntax

JS

class ClassWithField {

instanceField;

instanceFieldWithInitializer = "instance field";

static staticField;

static staticFieldWithInitializer = "static field";

}

ublic instance fields exist on every created instance of a class. By declaring a public field, you can ensure the field is always present, and the class definition is more self-documenting.

Public instance fields are added to the instance either at construction time in the base class (before the constructor body runs), or just after super() returns in a subclass. Fields without initializers are initialized to undefined. Like properties, field names may be computed.

JS

Copy to Clipboard

const PREFIX = "prefix";

class ClassWithField {

field;

fieldWithInitializer = "instance field";

[`${PREFIX}Field`] = "prefixed field";

}

const instance = new ClassWithField();

console.log(Object.hasOwn(instance, "field")); // true

console.log(instance.field); // undefined

console.log(instance.fieldWithInitializer); // "instance field"

console.log(instance.prefixField); // "prefixed field"

Computed field names are only evaluated once, at class definition time. This means that each class always has a fixed set of field names, and two instances cannot have different field names via computed names. The this value in the computed expression is the this surrounding the class definition, and referring to the class's name is a ReferenceError because the class is not initialized yet. await and yield work as expected in this expression.

JS

class C {

[Math.random()] = 1;

}

console.log(new C());

console.log(new C());

// Both instances have the same field name

In the field initializer, this refers to the class instance under construction, and super refers to the prototype property of the base class, which contains the base class's instance methods, but not its instance fields.

JS

class Base {

baseField = "base field";

anotherBaseField = this.baseField;

baseMethod() {

return "base method output";

}

}

class Derived extends Base {

subField = super.baseMethod();

}

const base = new Base();

const sub = new Derived();

console.log(base.anotherBaseField); // "base field"

console.log(sub.subField); // "base method output"

**Locations**

**static**

The static keyword defines a static method or field for a class, or a static initialization block. Static properties cannot be directly accessed on instances of the class. Instead, they're accessed on the class itself.

Static methods are often utility functions, such as functions to create or clone objects, whereas static properties are useful for caches, fixed-configuration, or any other data you don't need to be replicated across instances.

Syntax

JS

class ClassWithStatic {

static staticField;

static staticFieldWithInitializer = value;

static staticMethod() {

// …

}

}

1. The name of a static property (field or method) cannot be prototype.
2. The name of a class field (static or instance) cannot be constructor.

Description

This page introduces public static properties of classes, which include static methods, static accessors, and static fields.

For private static features, see private class features.

For instance features, see methods definitions, getter, setter, and public class fields.

Public static features are declared using the static keyword. They are added to the class constructor at the time of class evaluation using the [[DefineOwnProperty]] semantic (which is essentially Object.defineProperty()). They are accessed again from the class constructor.

Static methods are often utility functions, such as functions to create or clone instances. Public static fields are useful when you want a field to exist only once per class, not on every class instance you create. This is useful for caches, fixed-configuration, or any other data you don't need to be replicated across instances.

Static field names can be computed. The this value in the computed expression is the this surrounding the class definition, and referring to the class's name is a ReferenceError because the class is not initialized yet. await and yield work as expected in this expression.

Static fields can have an initializer. Static fields without initializers are initialized to undefined. Public static fields are not reinitialized on subclasses, but can be accessed via the prototype chain.

JS

Copy to Clipboard

class ClassWithStaticField {

static staticField;

static staticFieldWithInitializer = "static field";

}

class SubclassWithStaticField extends ClassWithStaticField {

static subStaticField = "subclass field";

}

console.log(Object.hasOwn(ClassWithStaticField, "staticField")); // true

console.log(ClassWithStaticField.staticField); // undefined

console.log(ClassWithStaticField.staticFieldWithInitializer); // "static field"

console.log(SubclassWithStaticField.staticFieldWithInitializer); // "static field"

console.log(SubclassWithStaticField.subStaticField); // "subclass field"

The name of a static property (field or method) cannot be prototype.

The name of a class field (static or instance) cannot be constructor.

Description

This page introduces public static properties of classes, which include static methods, static accessors, and static fields.

For private static features, see private class features.

For instance features, see methods definitions, getter, setter, and public class fields.

Public static features are declared using the static keyword. They are added to the class constructor at the time of class evaluation using the [[DefineOwnProperty]] semantic (which is essentially Object.defineProperty()). They are accessed again from the class constructor.

Static methods are often utility functions, such as functions to create or clone instances. Public static fields are useful when you want a field to exist only once per class, not on every class instance you create. This is useful for caches, fixed-configuration, or any other data you don't need to be replicated across instances.

Static field names can be computed. The this value in the computed expression is the this surrounding the class definition, and referring to the class's name is a ReferenceError because the class is not initialized yet. await and yield work as expected in this expression.

Static fields can have an initializer. Static fields without initializers are initialized to undefined. Public static fields are not reinitialized on subclasses, but can be accessed via the prototype chain.

JS

Copy to Clipboard

class ClassWithStaticField {

static staticField;

static staticFieldWithInitializer = "static field";

}

class SubclassWithStaticField extends ClassWithStaticField {

static subStaticField = "subclass field";

}

console.log(Object.hasOwn(ClassWithStaticField, "staticField")); // true

console.log(ClassWithStaticField.staticField); // undefined

console.log(ClassWithStaticField.staticFieldWithInitializer); // "static field"

console.log(SubclassWithStaticField.staticFieldWithInitializer); // "static field"

console.log(SubclassWithStaticField.subStaticField); // "subclass field"

**Examples**

Using static members in classes

The following example demonstrates several things:

How a static member (method or property) is defined on a class.

That a class with a static member can be sub-classed.

How a static member can and cannot be called.

JS

Copy to Clipboard

class Triple {

static customName = "Tripler";

static description = "I triple any number you provide";

static calculate(n = 1) {

return n \* 3;

}

}

class SquaredTriple extends Triple {

static longDescription;

static description = "I square the triple of any number you provide";

static calculate(n) {

return super.calculate(n) \* super.calculate(n);

}

}

console.log(Triple.description); // 'I triple any number you provide'

console.log(Triple.calculate()); // 3

console.log(Triple.calculate(6)); // 18

const tp = new Triple();

console.log(SquaredTriple.calculate(3)); // 81 (not affected by parent's instantiation)

console.log(SquaredTriple.description); // 'I square the triple of any number you provide'

console.log(SquaredTriple.longDescription); // undefined

console.log(SquaredTriple.customName); // 'Tripler'

// This throws because calculate() is a static member, not an instance member.

console.log(tp.calculate()); // 'tp.calculate is not a function'

Calling static members from another static method

In order to call a static method or property within another static method of the same class, you can use the this keyword.

JS

Copy to Clipboard

class StaticMethodCall {

static staticProperty = "static property";

static staticMethod() {

return `Static method and ${this.staticProperty} has been called`;

}

static anotherStaticMethod() {

return `${this.staticMethod()} from another static method`;

}

}

StaticMethodCall.staticMethod();

// 'Static method and static property has been called'

StaticMethodCall.anotherStaticMethod();

// 'Static method and static property has been called from another static method'

Calling static members from a class constructor and other methods

Static members are not directly accessible using the this keyword from non-static methods. You need to call them using the class name: CLASSNAME.STATIC\_METHOD\_NAME() / CLASSNAME.STATIC\_PROPERTY\_NAME or by calling the method as a property of the constructor: this.constructor.STATIC\_METHOD\_NAME() / this.constructor.STATIC\_PROPERTY\_NAME

JS

class StaticMethodCall {

constructor() {

console.log(StaticMethodCall.staticProperty); // 'static property'

console.log(this.constructor.staticProperty); // 'static property'

console.log(StaticMethodCall.staticMethod()); // 'static method has been called.'

console.log(this.constructor.staticMethod()); // 'static method has been called.'

}

static staticProperty = "static property";

static staticMethod() {

return "static method has been called.";

}

}

**Instance**

Static methods belong to a class and don’t act on its instances. This means that they can’t be called on instances of the class. Instead, they're called on the class itself. They are often utility functions, such as functions to create or clone objects.

Instance methods belong to the class prototype, which is inherited by all instances of the class. As such, they act on class instances and can be called on them.

const arr = [1, 2, 3]; // An Array instance

Array.isArray(arr); // Static method of Array

arr.push(4); // Instance method of Array

In the context of ES6 classes, the static keyword is used to define static methods for a class. Conversely, methods not defined as static are instance methods.

**Visibility**

**Private class features**

Class fields are public by default, but private class members can be created by using a hash # prefix. The privacy encapsulation of these class features is enforced by JavaScript itself.

Private members are not native to the language before this syntax existed. In prototypical inheritance, its behavior may be emulated with WeakMap objects or closures, but they can't compare to the # syntax in terms of ergonomics.

Syntax

JS

Copy to Clipboard

class ClassWithPrivate {

#privateField;

#privateFieldWithInitializer = 42;

#privateMethod() {

// …

}

static #privateStaticField;

static #privateStaticFieldWithInitializer = 42;

static #privateStaticMethod() {

// …

}

}

There are some additional syntax restrictions:

All private identifiers declared within a class must be unique. The namespace is shared between static and instance properties. The only exception is when the two declarations define a getter-setter pair.

The private identifier cannot be #constructor.

Description

Most class features have their private counterparts:

1. Private fields
2. Private methods
3. Private static fields
4. Private static methods
5. Private getters
6. Private setters
7. Private static getters
8. Private static setters

These features are collectively called private properties. However, constructors cannot be private in JavaScript. To prevent classes from being constructed outside of the class, you have to use a private flag.

Private properties are declared with # names (pronounced "hash names"), which are identifiers prefixed with #. The hash prefix is an inherent part of the property name — you can draw relationship with the old underscore prefix convention \_privateField — but it's not an ordinary string property, so you can't dynamically access it with the bracket notation.

It is a syntax error to refer to # names from outside of the class. It is also a syntax error to refer to private properties that were not declared in the class body, or to attempt to remove declared properties with delete.

**Examples**

Private fields

Private fields include private instance fields and private static fields. Private fields are only accessible from inside the class declaration.

Private instance fields

Like their public counterparts, private instance fields:

are added before the constructor runs in a base class, or immediately after super() is invoked in a subclass, and

are only available on instances of the class.

JS

class ClassWithPrivateField {

#privateField;

constructor() {

this.#privateField = 42;

}

}

class Subclass extends ClassWithPrivateField {

#subPrivateField;

constructor() {

super();

this.#subPrivateField = 23;

}

}

new Subclass(); // In some dev tools, it shows Subclass {#privateField: 42, #subPrivateField: 23}

Note: #privateField from the ClassWithPrivateField base class is private to ClassWithPrivateField and is not accessible from the derived Subclass.

Returning overriding object

A class's constructor can return a different object, which will be used as the new this for the derived class constructor. The derived class may then define private fields on that returned object — meaning it is possible to "stamp" private fields onto unrelated objects.

JS

Copy to Clipboard

class Stamper extends class {

// A base class whose constructor returns the object it's given

constructor(obj) {

return obj;

}

} {

// This declaration will "stamp" the private field onto the object

// returned by the base class constructor

#stamp = 42;

static getStamp(obj) {

return obj.#stamp;

}

}

const obj = {};

new Stamper(obj);

// `Stamper` calls `Base`, which returns `obj`, so `obj` is

// now the `this` value. `Stamper` then defines `#stamp` on `obj`

console.log(obj); // In some dev tools, it shows {#stamp: 42}

console.log(Stamper.getStamp(obj)); // 42

console.log(obj instanceof Stamper); // false

Warning: This is a potentially very confusing thing to do. You are generally advised to avoid returning anything from the constructor — especially something unrelated to this.

Private static fields

Like their public counterparts, private static fields:

are added to the class constructor at class evaluation time, and

are only available on the class itself.

JS

Copy to Clipboard

class ClassWithPrivateStaticField {

static #privateStaticField = 42;

static publicStaticMethod() {

return ClassWithPrivateStaticField.#privateStaticField;

}

}

console.log(ClassWithPrivateStaticField.publicStaticMethod()); // 42

There is a restriction on private static fields: only the class which defines the private static field can access the field. This can lead to unexpected behavior when using this. In the following example, this refers to the Subclass class (not the ClassWithPrivateStaticField class) when we try to call Subclass.publicStaticMethod(), and so causes a TypeError.

JS

Copy to Clipboard

class ClassWithPrivateStaticField {

static #privateStaticField = 42;

static publicStaticMethod() {

return this.#privateStaticField;

}

}

class Subclass extends ClassWithPrivateStaticField {}

Subclass.publicStaticMethod(); // TypeError: Cannot read private member #privateStaticField from an object whose class did not declare it

This is the same if you call the method with super, because super methods are not called with the super class as this.

JS

Copy to Clipboard

class ClassWithPrivateStaticField {

static #privateStaticField = 42;

static publicStaticMethod() {

// When invoked through super, `this` still refers to Subclass

return this.#privateStaticField;

}

}

class Subclass extends ClassWithPrivateStaticField {

static callSuperMethod() {

return super.publicStaticMethod();

}

}

Subclass.callSuperMethod(); // TypeError: Cannot read private member #privateStaticField from an object whose class did not declare it

You are advised to always access private static fields through the class name, not through this, so inheritance doesn't break the method.

Private methods

Private methods include private instance methods and private static methods. Private methods are only accessible from inside the class declaration.

Private instance methods

Unlike their public counterparts, private instance methods:

are installed immediately before the instance fields are installed, and

are only available on instances of the class, not on its .prototype property.

JS

Copy to Clipboard

class ClassWithPrivateMethod {

#privateMethod() {

return 42;

}

publicMethod() {

return this.#privateMethod();

}

}

const instance = new ClassWithPrivateMethod();

console.log(instance.publicMethod()); // 42

Private instance methods may be generator, async, or async generator functions. Private getters and setters are also possible, and follow the same syntax requirements as their public getter and setter counterparts.

JS

Copy to Clipboard

class ClassWithPrivateAccessor {

#message;

get #decoratedMessage() {

return `🎬${this.#message}🛑`;

}

set #decoratedMessage(msg) {

this.#message = msg;

}

constructor() {

this.#decoratedMessage = "hello world";

console.log(this.#decoratedMessage);

}

}

new ClassWithPrivateAccessor(); // 🎬hello world🛑

Unlike public methods, private methods are not accessible on the .prototype property of their class.

JS

Copy to Clipboard

class C {

#method() {}

static getMethod(x) {

return x.#method;

}

}

console.log(C.getMethod(new C())); // [Function: #method]

console.log(C.getMethod(C.prototype)); // TypeError: Receiver must be an instance of class C

Private static methods

Like their public counterparts, private static methods:

are added to the class constructor at class evaluation time, and

are only available on the class itself.

JS

Copy to Clipboard

class ClassWithPrivateStaticMethod {

static #privateStaticMethod() {

return 42;

}

static publicStaticMethod() {

return ClassWithPrivateStaticMethod.#privateStaticMethod();

}

}

console.log(ClassWithPrivateStaticMethod.publicStaticMethod()); // 42

Private static methods may be generator, async, and async generator functions.

The same restriction previously mentioned for private static fields holds for private static methods, and similarly can lead to unexpected behavior when using this. In the following example, when we try to call Subclass.publicStaticMethod(), this refers to the Subclass class (not the ClassWithPrivateStaticMethod class) and so causes a TypeError.

JS

Copy to Clipboard

class ClassWithPrivateStaticMethod {

static #privateStaticMethod() {

return 42;

}

static publicStaticMethod() {

return this.#privateStaticMethod();

}

}

class Subclass extends ClassWithPrivateStaticMethod {}

console.log(Subclass.publicStaticMethod()); // TypeError: Cannot read private member #privateStaticMethod from an object whose class did not declare it

Simulating private constructors

Many other languages include the capability to mark a constructor as private, which prevents the class from being instantiated outside of the class itself — you can only use static factory methods that create instances, or not be able to create instances at all. JavaScript does not have a native way to do this, but it can be accomplished by using a private static flag.

JS

class PrivateConstructor {

static #isInternalConstructing = false;

constructor() {

if (!PrivateConstructor.#isInternalConstructing) {

throw new TypeError("PrivateConstructor is not constructable");

}

PrivateConstructor.#isInternalConstructing = false;

// More initialization logic

}

static create() {

PrivateConstructor.#isInternalConstructing = true;

const instance = new PrivateConstructor();

return instance;

}

}

new PrivateConstructor(); // TypeError: PrivateConstructor is not constructable

PrivateConstructor.create(); // PrivateConstructor {}

**Public class fields**

Public fields are writable, enumerable, and configurable properties. As such, unlike their private counterparts, they participate in prototype inheritance.

Syntax

JS

Copy to Clipboard

class ClassWithField {

instanceField;

instanceFieldWithInitializer = "instance field";

static staticField;

static staticFieldWithInitializer = "static field";

}

**Examples**

Using class fields

Class fields cannot depend on arguments of the constructor, so field initializers usually evaluate to the same value for each instance (unless the same expression can evaluate to different values each time, such as Date.now() or object initializers).

class Person {

// All instances of Person will have the same name

name = "Dragomir";

}

class Person {

name;

age;

constructor(name, age) {

this.name = name;

this.age = age;

}

}

The code above seems repetitive, but consider the case where this is dynamically mutated: the explicit field declaration makes it clear which fields will definitely be present on the instance.

JS

Copy to Clipboard

class Person {

name;

age;

constructor(properties) {

Object.assign(this, properties);

}

}

Because initializers are evaluated after the base class has executed, you can access properties created by the base class constructor.

JS

class Person {

name;

age;

constructor(name, age) {

this.name = name;

this.age = age;

}

}

class Professor extends Person {

name = `Professor ${this.name}`;

}

console.log(new Professor("Radev", 54).name); // "Professor Radev"