**CLASSES**

* 1. Classes are a template for creating objects. They encapsulate data with code to work on that data. Classes in JS are built on [prototypes](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain) but also have some syntax and semantics that are unique to classes.

A class element can be characterized by three aspects:

* Kind: Getter, setter, method, or field
* Location: Static or instance
* Visibility: Public or private
  1. Classes declaration

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}}

// Expression; the class is anonymous but assigned to a variableconst

Rectangle = class {

constructor(height, width) {

this.height = height;

this.width = width;

}};

// Expression; the class has its own nameconst

Rectangle = class Rectangle2 {

constructor(height, width) {

this.height = height;

this.width = width;

}};

* 1. The body of a class is executed in [strict mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode) even without the "use strict" directive.
  2. Constructor
     1. The [constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError) is thrown if the class contains more than one occurrence of a constructor method.
     2. A constructor can use the [super](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super) keyword to call the constructor of the super class.
     3. If you don't provide your own constructor, then a default constructor will be supplied for you.
     4. If your class is a base class, the default constructor is empty.
     5. If your class is a derived class, the default constructor calls the parent constructor, passing along any arguments that were provided:
     6. The difference between an explicit constructor like the one above and the default constructor is that the latter doesn't actually invoke [the array iterator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/@@iterator) through [argument spreading](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax).
     7. The constructor method may have a return value. While the base class may return anything from its constructor, the derived class must return an object or undefined, or a [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError) will be thrown.

class ParentClass {

constructor() {

return 1;

}

}

console.log(new ParentClass()); // ParentClass {}

// The return value is ignored because it's not an object

// This is consistent with function constructors

class ChildClass extends ParentClass {

constructor() {

return 1;

}

}

console.log(new ChildClass()); // TypeError: Derived constructors may only return object or undefined

* + 1. Private names cannot be called #constructor. Any member named constructor must be a plain method. To prevent classes from being constructed outside of the class, you have to [use a private flag](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Private_class_fields" \l "simulating_private_constructors).
    2. Calling super in a constructor bound to a different prototype
       1. super() calls the constructor that's the prototype of the current class. If you change the prototype of the current class itself, super() will call the constructor that's the new prototype. Changing the prototype of the current class's prototype property doesn't affect which constructor super() calls.

class Polygon {

constructor() {

this.name = "Polygon";

}

}

class Rectangle {

constructor() {

this.name = "Rectangle";

}

}

class Square extends Polygon {

constructor() {

super();

}

}

// Make Square extend Rectangle (which is a base class) instead of Polygon

Object.setPrototypeOf(Square, Rectangle);

const newInstance = new Square();

// newInstance is still an instance of Polygon, because we didn't

// change the prototype of Square.prototype, so the prototype chain

// of newInstance is still

// newInstance --> Square.prototype --> Polygon.prototype

console.log(newInstance instanceof Polygon); // true

console.log(newInstance instanceof Rectangle); // false

// However, because super() calls Rectangle as constructor, the name property

// of newInstance is initialized with the logic in Rectangle

console.log(newInstance.name); // Rectangle

* 1. Methods
     1. Methods can be plain functions, async functions, generator functions, or async generator functions. For more information, see [method definitions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Method_definitions).

Eg-// Getter

get area() {

return this.calcArea();

}

// Method

calcArea() {

return this.height \* this.width;

* 1. Static fields and methods
     1. Static fields can have an initializer. Static fields without initializers are initialized to undefined.
     2. The [static](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/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.

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

Iii. class ClassWithStaticField {

static baseStaticField = "base static field";

static anotherBaseStaticField = this.baseStaticField;

static baseStaticMethod() {

return "base static method output";

}

}

class SubClassWithStaticField extends ClassWithStaticField {

static baseStaticMethod() {

return "sub static method output";

}

static subStaticField = this.baseStaticMethod();

}

console.log(ClassWithStaticField.anotherBaseStaticField); // "base static field"

console.log(SubClassWithStaticField.subStaticField); // "sub static method output"

//if static subStaticField=super.baseStaticMethod();

//output->base static method output

* 1. Fields
     1. 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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes" \l "private_class_features) use a special identifier syntax(#), so modifier keywords like public and private should not be used either.
     2. the fields can be declared with or without a default value. Fields without default values default to undefined.
     3. It's an error to reference private fields from outside of the class; they can only be read or written within the class body.
     4. 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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete).

class ClassWithPrivateField {

#privateField;

constructor() {;

delete this.#privateField; // Syntax error bcz declared

this.#undeclaredField = 42; // Syntax error bcz not declared

}

}

const instance = new ClassWithPrivateField();

instance.#privateField; // Syntax error bcz accessing outside

* + 1. If you access a private property from an object that doesn't have the property, a [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError) is thrown, instead of returning undefined as normal properties do.

class C {

#x;

static getX(obj) {

return obj.#x;

}

}

console.log(C.getX(new C())); // undefined

console.log(C.getX({})); // TypeError: Cannot read private member #x from an object whose class did not declare it

* + 1. You can use the [in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/in) operator to check whether an externally defined object possesses a private property. This will return true if the private field or method exists, and false otherwise.

class C {

#x;

constructor(x) {

this.#x = x;

}

static getX(obj) {

if (#x in obj) return obj.#x;

return "obj must be an instance of C";

}

}

console.log(C.getX(new C("foo"))); // "foo"

console.log(C.getX(new C(0.196))); // 0.196

console.log(C.getX(new C(new Date()))); // the current date and time

console.log(C.getX({})); // "obj must be an instance of C"

* + 1. There is a restriction on private static fields: only the class which defines the private static field can access the field.
    2. 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.

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

* + 1. In the following example we will get 42 as output

class ClassWithPrivateStaticField {

static #privateStaticField = 42;

static publicStaticMethod() {

return ClassWithPrivateStaticField .#privateStaticField;

}

}

class Subclass extends ClassWithPrivateStaticField {}

Console.log(Subclass.publicStaticMethod()); // 42

* 1. Inheritence
     1. The [extends](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/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).

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.

* 1. Binding this with instance and static methods
     1. When a static or instance method is called without a value for [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/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"](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode) directive isn't present, because code within the class body is always executed in strict mode.

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

* + 1. If we rewrite the above using traditional function-based syntax in non–strict mode, then this method calls are automatically bound to [globalThis](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/globalThis). In strict mode, the value of this remains as undefined.

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)

* 1. Static Initialization blocks
     1. The scope of the variables declared inside the static block is local to the block. This includes var, function, const, and let declarations. var declarations in the block are not hoisted.

var y = "Outer y";

class A {

static field = "Inner y";

static {

var y = this.field;

}

}

// var defined in static block is not hoisted

console.log(y); // 'Outer y'

* + 1. The this inside a static block refers to the constructor object of the class. This code shows how to access a public static field.

class A {

static field = "static field";

static {

console.log(this.field);

}

}

// 'static field'

* + 1. The [super.property](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super) syntax can be used inside a static block to reference static properties of a super class.

class A {

static field = "static field";

}

class B extends A {

static {

console.log(super.field);

}

}

// 'static field'

* + 1. This example below shows how access can be granted to a private instance field of a class from an object outside the class

let getDPrivateField;

class D {

#privateField;

constructor(v) {

this.#privateField = v;

}

static {

getDPrivateField = (d) => d.#privateField;

}

}

console.log(getDPrivateField(new D("private"))); // 'private'