# **Clases**

Klasės

## The fundamental part of most classes is its constructor, which sets up each instance's initial state and handles any parameters that were passed when calling new.

## It's defined in a class block as though you're defining a method named constructor, though it's actually handled as a special case.

| class MyClass {  constructor(option) {  console.log(`Creating instance using ${option} option.`);  this.option = option;  } } |
| --- |

| const foo = new MyClass('speedy'); *// logs: "Creating instance using speedy option"* |
| --- |

## Class Inheritance

Inheritance works just like it does in other object-oriented languages: methods defined on the superclass are accessible in the extending subclass.

If the subclass declares its own constructor then it must invoke the parents constructor via super() before it can access this.

| class SuperClass {  constructor() {  this.logger = console.log;  }  log() {  this.logger(`Hello ${this.name}`);  } } class SubClass extends SuperClass {  constructor() {  super();  this.name = 'subclass';  } } const subClass = new SubClass(); subClass.log(); *// logs: "Hello subclass"* |
| --- |

## Static Methods

Static methods and properties are defined on the class/constructor itself, not on instance objects. These are specified in a class definition by using the static keyword.

| class MyClass {  static myStaticMethod() {  return 'Hello';  }  static get myStaticProperty() {  return 'Goodbye';  } } console.log(MyClass.myStaticMethod()); *// logs: "Hello" console.log(MyClass.myStaticProperty); // logs: "Goodbye"* |
| --- |

We can see that static properties are not defined on object instances:

| const myClassInstance = new MyClass(); console.log(myClassInstance.myStaticProperty); *// logs: undefined* |
| --- |

However, they are defined on subclasses:

| class MySubClass extends MyClass {}; console.log(MySubClass.myStaticMethod()); *// logs: "Hello" console.log(MySubClass.myStaticProperty); // logs: "Goodbye"* |
| --- |

## Getters and Setters

Getters and setters allow you to define custom behaviour for reading and writing a given property on your class. To the user, they appear the same as any typical property. However, internally a custom function you provide is used to determine the value when the property is accessed (the getter), and to perform any necessary changes when the property is assigned (the setter).

In a class definition, a getter is written like a no-argument method prefixed by the get keyword. A setter is similar, except that it accepts one argument (the new value being assigned) and the set keyword is used instead.

Here's an example class which provides a getter and setter for its .name property. Each time it's assigned, we'll record the new name in an internal .names\_ array. Each time it's accessed, we'll return the latest name.

| class MyClass {  constructor() {  this.names\_ = [];  }  set name(value) {  this.names\_.push(value);  }  get name() {  return this.names\_[this.names\_.length - 1];  } }  const myClassInstance = new MyClass(); myClassInstance.name = 'Joe'; myClassInstance.name = 'Bob'; console.log(myClassInstance.name); *// logs: "Bob" console.log(myClassInstance.names\_); // logs: ["Joe", "Bob"]* |
| --- |

If you only define a setter, attempting to access the property will always return undefined.

| const classInstance = new class {  set prop(value) {  console.log('setting', value);  } }; classInstance.prop = 10; *// logs: "setting", 10 console.log(classInstance.prop); // logs: undefined* |
| --- |

If you only define a getter, attempting to assign the property will have no effect.

| const classInstance = new class {  get prop() {  return 5;  } }; classInstance.prop = 10; console.log(classInstance.prop); *// logs: 5* |
| --- |

## Private Members

JavaScript does not technically support private members as a language feature. Privacy - described by Douglas Crockford - gets emulated instead via closures (preserved function scope) that will be generated each with every instantiation call of a constructor function.

The Queue example demonstrates how, with constructor functions, local state can be preserved and made accessible too via privileged methods.

| class Queue {  constructor() {  const list = [];  this.enqueue = function(type) {  list.push(type);  return type;  }  this.dequeue = function() {  return list.shift();  }  } }  var q = new Queue;  q.enqueue(9); *// First in* q.enqueue(8); console.log(q.dequeue()); *// 9 first out* console.log(q.dequeue()); *// 8* console.log(q.dequeue()); *// 7* console.log(q); *// {}* console.log(Object.keys(q)); *// ["enqueue", "dequeue"]* |
| --- |

With every instantiation of a Queue type the constructor generates a closure.

Thus both of a Queue type's own methods enqueue and dequeue (see Object.keys(q)) still do have access to list

that continues to live in its enclosing scope that, at construction time, has been preserved.

Making use of this pattern - emulating private members via privileged public methods - one should keep in mind that, with every instance, additional memory will be consumed for every own property method (for it is code that can't be shared/reused). The same is true for the amount/size of state that is going to be stored within such a closure.

## 

## Methods

Methods can be defined in classes to perform a function and optionally return a result. They can receive arguments from the caller.

| class Something {  constructor(data) {  this.data = data  }  doSomething(text) {  return {  data: this.data,  text  }  } } var s = new Something({}) s.doSomething("hi") *// returns: { data: {}, text: "hi" }* |
| --- |

## Dynamic Methods Names

There is also the ability to evaluate expressions when naming methods similar to how you can access an objects' properties with []. This can be useful for having dynamic property names, however is often used in conjunction with Symbols.

| let METADATA = Symbol('metadata'); class Car {  constructor(make, model) {  this.make = make;  this.model = model;  }  [METADATA]() {  return {  make: this.make,  model: this.model  };  }   ["add"](a, b) {  return a + b;  }   [1 + 2]() {  return "three";  } }   let MazdaMPV = new Car("Mazda", "MPV"); MazdaMPV.add(4, 5); *// 9* MazdaMPV[3](); *// "three"* MazdaMPV[METADATA](); *// { make: "Mazda", model: "MPV" }* |
| --- |

## Class Name binding

Class Declaration's Name is bound in different ways in different scopes -

1. The scope in which the class is defined - let binding

2. The scope of the class itself - within { and } in class {} - const binding

| class Foo {  *// Foo inside this block is a const binding* }  *// Foo here is a let binding* |
| --- |

| class A {  foo() {  A = null; *// will throw at runtime as A inside the class is a `const` binding }*  } } A = null; *// will NOT throw as A here is a `let` binding* |
| --- |

This is not the same for a Function -

| function A() {  A = null; *// works* }  A.prototype.foo = function foo() {  A = null; *// works* } A = null; *// works* |
| --- |