JavaScript Objects and Classes

.NET CORE

Classes are in fact "special functions", and just as you can define function expressions and function declarations, the class syntax has two components: class expressions and class declarations.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

JS objects

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/A_first_splash
https://javascript.info/object

In JavaScript, everything is an object. An ***object*** is a collection of related functionality stored in a single grouping. You can create your own ***objects***. ***Objects*** are used to store ***properties*** (***key:value*** collections) of various data and more complex entities. In JavaScript, ***objects*** penetrate almost every aspect of the language. Objects are stored by ***reference***, so the variable representing the object merely holds the memory location of the object on the heap.

An empty ***object*** can be created with figure brackets {…} with an optional list of properties in two ways.

```
1  let user = new Object(); // "object constructor" syntax
2  let user = {};  // "object literal" syntax
```

An ***Object Literal*** is created immediately with properties. Property values are accessible using dot (.) notation.

```
1  let user = {     // an object
2    name: "John",  // by key "name" store value "John"
3    age: 30        // by key "age" store value 30
4  };
```

JS objects – Common Commands

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/A_first_splash
https://javascript.info/object

Command	Usage
Delete user.age	Remove a property (age) from an object (user).

JS Objects – Property Values and Shorthand

https://javascript.info/object#property-value-shorthand

JavaScript has a shorthand for declaring and setting object variables.
The below examples are all equivalent objects.

```
function makeUser(name, age) {
  return {
    name: name,
    age: age,
    // ...other properties
  };
}

let user = makeUser("John", 30);
alert(user.name); // John
```

This is reusable

```
function makeUser(name, age) {
  return {
    name, // same as name: name
    age,  // same as age: age
    // ...
  };
}
```

This is reusable

```
let user = {
  name,  // same as name:name
  age: 30
};
```

This is not reusable

JS Objects - Accessing Properties

https://javascript.info/object#property-existence-test-in-operator
https://javascript.info/object#the-for-in-loop

It's possible to access any property of an ***object***.

user.noSuchProperty returns ***undefined*** if the property exists, ***true*** if does not exist.

The ***in*** operator can also be used.
"propertyName" in objectName returns ***true*** of the property exists, ***false*** if it doesn't.

Use the ***for...in*** loop to access each property of an object in sequence.

The ***this*** keyword can be used to specify which variable to access

```
let user = {
  name: "John",
  age: 30,
  isAdmin: true
};

for (let key in user) {
  // keys
  alert( key );  // name, age, isAdmin
  // values for the keys
  alert( user[key] ); // John, 30, true
}
```

```
let user = {
  name: "John",
  age: 30,

  sayHi() {
    // "this" is the "current object"
    alert(this.name);
  }

};

user.sayHi(); // John
```

JS Objects – Objects in Objects

https://javascript.info/object#cloning-and-merging-object-assign

An object can contain another object. In this example, you would access ***height*** with

let height = user.sizes.height

```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};

alert( user.sizes.height ); // 182
```

An object can be assigned another ***object*** or ***function*** after being created.

```
let user = {
  // ...
};

// first, declare
function sayHi() {
  alert("Hello!");
};

// then add as a method
user.sayHi = sayHi;

user.sayHi(); // Hello!
```

JS Objects – Constructors and new

https://javascript.info/constructor-new

A ***constructor function*** in JavaScript serves the same purpose as a ***Class constructor*** in C#.

Constructor functions technically are regular functions. There are two conventions though:

- They are named with capital letter first.
- They should be executed only with "new" operator.

When a function is executed with ***new***, it does the following steps

1. A new empty object is created and assigned to ***this***.
2. The function body executes. Usually it modifies ***this***, adds new properties to it.
3. The value of ***this*** is returned.

The main purpose of constructors is to implement reusable object creation code. Like for Classes!

```
1  function User(name) {
2    this.name = name;
3    this.isAdmin = false;
4  }
5
6  let user = new User("Jack");
7
8  alert(user.name); // Jack
9  alert(user.isAdmin); // false
```

```
1  function User(name) {
2    // this = {};  (implicitly)
3
4    // add properties to this
5    this.name = name;
6    this.isAdmin = false;
7
8    // return this;  (implicitly)
9  }
```

JavaScript Classes

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

JavaScript ***classes*** are primarily *syntactical sugar* over JavaScript's existing ***prototype-based inheritance***. The ***class*** syntax does not introduce a new object-oriented inheritance model to JavaScript. Classes are "special functions" and just as you can define ***function expressions*** and ***function declarations***, the class syntax has two components: ***class expressions*** and ***class declarations***.

Class Declaration	Class Expression

class Rectangle {
  constructor(height, width) {
  this.height = height;
  this.width = width;
  }
}
	Class expressions can be named or unnamed. The name given to a named class expression is local to the class's body. (it can be retrieved through the class's (not an instance's) name property.

A class must be declared before they can be accessed. (no ***Hoisting***)	

```
1  // unnamed
2  let Rectangle = class {
3    constructor(height, width) {
4      this.height = height;
5      this.width = width;
6    }
7  };
8  console.log(Rectangle.name);
9  // output: "Rectangle"
10
11 // named
12 let Rectangle = class Rectangle2 {
13   constructor(height, width) {
14     this.height = height;
15     this.width = width;
16   }
17 };
18 console.log(Rectangle.name);
19 // output: "Rectangle2"
```

JS Class Parts

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

The ***constructor method*** creates and initializes an object created with a class template. **There can be only one...** (constructor in a class).

Instance Properties - must be defined inside of class methods.

Prototype Methods – Declared in the class. Available through an instance of the class.

Static Methods - called without instantiating their class and cannot be called through a class instance. (below)

```
1  class Point {
2    constructor(x, y) {
3      this.x = x;
4      this.y = y;
5    }
6
7    static distance(a, b) {
8      const dx = a.x - b.x;
9      const dy = a.y - b.y;
```

```
15 const p1 = new Point(5, 5);
16 const p2 = new Point(10, 10);
17 p1.distance; //undefined
18 p2.distance; //undefined
19
20 console.log(Point.distance(p1, p2));
```

```
1  class Rectangle {
2    constructor(height, width) {
3      this.height = height;
4      this.width = width;
5    }
6    // Getter
7    get area() {
8      return this.calcArea();
9    }
10   // Method
11   calcArea() {
12     return this.height * this.width;
13   }
14 }
15
16 const square = new Rectangle(10, 10);
17
18 console.log(square.area); // 100
```

Getters and Setters

https://javascript.info/property-accessors#getters-and-setters

Accessor properties are new to JS. They are functions that work on ***getting*** and ***setting*** a value but look like regular ***properties*** to external code.

Getters and ***Setters*** are accessed like properties. (*instanceName.getterName*).

Getters and ***Setters*** allow validation to be written inside the class.

```
6    set name(value) {
7      if (value.length < 4) {
8        alert("Name is too shor
9        return;
10     }
11     this._name = value;
12   }
```

```
1  let user = {
2    name: "John",
3    surname: "Smith",
4
5    get fullName() {
6      return `${this.name} ${this.surname}`;
7    },
8
9    set fullName(value) {
10     [this.name, this.surname] = value.split(" ");
11   }
12 };
13
14 // set fullName is executed with the given value.
15 user.fullName = "Alice Cooper";
16
17 alert(user.name); // Alice
18 alert(user.surname); // Cooper
```

JavaScript [[Prototypes]]

https://javascript.info/prototype-inheritance

- In JavaScript, objects have a special hidden property [[Prototype]] (as named in the specification), that is either null or references another object. That object is called “a prototype”. When we want to read a property from object, and it’s missing, JavaScript automatically takes it from the prototype. In programming, such thing is called “prototypal inheritance”. The property [[Prototype]] is internal and hidden, but there are many ways to set it.
  - Multiple prototype inheritance is not allowed.
  - __proto__ doesn’t support…
    - *write* (overwriting) actions.
  - References can be chained
  - References cannot go in circles.
  - Getter/setter functions are inherited.

Inherit Methods

```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// walk is taken from the prototype
rabbit.walk(); // Animal walk
```

Hierarchical Inheritance

```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

let longEar = {
  earLength: 10,
  __proto__: rabbit
};

// walk is taken from the prototype chain
longEar.walk(); // Animal walk
alert(longEar.jumps); // true (from rabbit)
```

JavaScript Prototypes

https://javascript.info/function-prototype
https://javascript.info/prototype-methods

Prototypal Inheritance was one of the core features of JS originally, but there was no direct access to it. The only thing that worked reliably was a "prototype" property of the constructor function. There are many scripts that still use it. Remember, ***prototype*** is a default ***property*** provided in the ***constructor***.

In this example, setting ***Rabbit.prototype = animal*** sets its ***prototype*** to ***animal***.

```
1  let animal = {
2    eats: true
3  };
4
5  function Rabbit(name) {
6    this.name = name;
7  }
8
9  Rabbit.prototype = animal;
10
11 let rabbit = new Rabbit("White Rabbit");
12 //  rabbit.__proto__ == animal
13 alert( rabbit.eats ); // true
```

JavaScript Objects without __proto__

https://javascript.info/function-prototype
https://javascript.info/prototype-methods

__proto__ is considered outdated and somewhat deprecated (in browser-only parts of the JavaScript standard).

Instead of __proto__, use:

- ***Object.create(proto[, descriptors])*** – creates an empty object with given proto as [[Prototype]] and optional property descriptors.
- ***Object.getPrototypeOf(obj)*** – returns the [[Prototype]] of obj.
- ***Object.setPrototypeOf(obj, proto)*** – sets the [[Prototype]] of obj to proto.

```
let animal = {
  eats: true
};

// create a new object with animal as a prototype
let rabbit = Object.create(animal);

alert(rabbit.eats); // true

alert(Object.getPrototypeOf(rabbit) === animal); // true

Object.setPrototypeOf(rabbit, {}); // change the prototyp
                                   // change the prototype of rabbit to {}
```

JS Class Inheritance

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes#Sub_classing_with_extends

---

The ***extends*** keyword is used in class declarations or class expressions to create a ***class*** as a ***child*** of another ***class***.

If there is a ***constructor*** present in the ***subclass***, it needs to first call ***super()*** before using "***this***".

If you want to inherit from a regular object, you can instead use ***Object.setPrototypeOf()***

```
1  class Animal {
2    constructor(name) {
3      this.name = name;
4    }
5
6    speak() {
7      console.log(`${this.name} makes a noise.`);
8    }
9  }
10
11 class Dog extends Animal {
12   constructor(name) {
13     super(name); // call the super class constructor
14                  and pass in the name parameter
15   }
16
17   speak() {
18     console.log(`${this.name} barks.`);
19   }
20 }
21 let d = new Dog('Mitzie');
22 d.speak(); // Mitzie barks.
```