React Academy

JavaScript Object Cheatsheet

Creating Empty Objects

```
let emptyObject = {}; //Bracket notation
let emptyObject2 = new Object(); //constructor

{} !== {}  //each bracket creates a new object instance
```

Adding properties to an object

```
let person = new Object();

//You add or modify a property using two notations:
//1. using dot notation
person.name = 'Jim';
cosole.log(person.name);

//2. using bracket notation
person['name'] = 'Jim';
cosole.log(person['name']);
//this is useful when the prop name is stored in a
variable
```

Object literals

```
let person =  {
  name: 'Jim',
  age: 23,
  interests: ['racing', 'jogging'],
  greet: function() {
    alert(`Hello from ${this.name}`);
  }
};
//three properties and one method

//to use (with dot notation):
console.log(person.name);
console.log(person.interests[0]);
person.greet();

//to use (with bracket notation):
console.log(person['name']);
console.log(person['interests'][0]);
person['greet']();
```

Nested objects

```
let person = {
  name: 'John',
  address: {
    street: 'main',
    city: 'Montreal',
    province: 'QC'
  }
};

//modify nested data (dot notation)
person.address.city='Toronto';

//modify nested data (bracket notation):
person['address']['city']='Toronto';
```

`this` usage

In the following example, `this` refers to the person object when it is called.

```
let person = {
  name: 'John',
  planMeeting: function() {
                  return `${this.name} is meeting later`;
  }
}

person.planMeeting(); //"John is meeting later"
```

In other programming languages, `this` always refers to the current object. But this isn't always the case in JS.

`this` is different than other languages

`this` in most languages refers to the current object context. But in Javascript, `this` refers to the **calling context**, how a function is called. `this` will function similar to other languages in most cases. But it will fail when we create *alias function* or when method is called from an *event*.

```
let person = {
  name: 'John',
  planMeeting: function(time) {
    return `${this.name} is meeting at ${time}`;
  }
}

//let's create an alias called scheduleMeeting
let scheduleMeeting = person.planMeeting;

person.planMeeting('1PM');
//"John is meeting at 1PM"

scheduleMeeting('1PM');
// error: this is undefined in strict mode
```

There are a three workarounds when `this` is not in the calling context, to explicitly set the value of `this`:

```
//use call(), first param is the 'this' reference
scheduleMeeting.call(person, '1PM');

//use apply(), second param is an array with the args.
scheduleMeeting.apply(person, ['1PM']);

//use bind(), sets 'this' for all future calls
scheduleMeeting=scheduleMeeting.bind(person);
scheduleMeeting('1PM');
```

When `this` is not set.

When `this` isn't specified, a reference will be provided for `this`.

- In sloppy mode, `this` will be equal to the global object.
  - In a browser, the global object is set to `window`.
  - In Node.js, the global object is set to `global`.
  - In a Worker, the global object is set to `WorkerGlobalScope`.
- In strict mode, `this` will be `undefined`. (Recommended)
  - To access the global object (even in strict mode), use `globalThis`

`this` and arrow functions

With arrow functions, `this` always refers to its container. No need for workarounds.

```
let person = {
  name: 'John',
  planMeeting: (time) => {
    return `${this.name} is meeting at ${time}`;
  }
}
```

Classes

In Object-Oriented languages, *classes* define the structure (properties, methods) of what an object looks like. In JavaScript, there are no classes in the strict sense, but we do have *constructor functions* and *prototype* that can emulate a few of the ways a class work.

With modern JavaScript, there is a *class* structure in ES6+. But that structure is syntactic sugar, that simply hides the "ugliness" of using *constructor functions* and *prototypes*. Below the surface, *class* is using the same structures.

Class naming convention

In JavaScript, most *classes* and *constructor functions* start with an uppecase letter. It is a convention that you should follow for your own code.

Constuctor function for a class

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.greet = function() {
    return (`Hello from ${this.name}.`);
  };
}
```

Note that greet will create a new function for each instance created. This is not the right approach and will waste memory.

```
let person1 = new Person('John', 23);
let person2 = new Person('Sarah', 42);
person1.greet();
person2.greet();
//These functions aren't the same.
```

Prototypes

A prototype is a hidden property of every object. It refers to another object, which has its own props and methods.

Each time JavaScript tries to access a member (prop or method), it looks at the current object. If it cannot find the member, then it walks up the prototype chain, to look at the parent object. It walks up from one prototype to the next, until it reaches the top (the `Object` prototype.)

To get the prototype of an object instance, you can use `.__proto__`, a non-standard property. (The standard way to get the prototype is through `Object.getPrototypeOf(obj)`.)

Every *constructor function* is going to have a special property called `.prototype`, that is the prototype object used by all instances of this constructor.

If we are using a *constructor function* called `Person`, and instanciate a few people with the following code:

```
let person1 = new Person();
let person2 = new Person();
person1 !== person2; // these are two different objects
person1.__proto__ === person2.__proto__;
//true, all instances share the same prototype
person1.__proto__ === Person.prototype //true,
//person1 is an object instanciated with the Person
constructor
Person.__proto__ !== Person.prototype // These aren't the
same.
Person.__proto__  === Function.prototype //true,
//Person is an instance of a function  (constructor)
```

Prototypes with `object.create()`

You can create prototypes from existing ojects.

```
let person1 = { name: "John", age: 23 };
let person2 = Object.create(person1);
//person2 is a new empty object,
//but with the prototype pointing to person1

console.log(person2);      // {},  the object is empty
console.log(person2.name); //John.
//JS got the value by going up the prototype chain

person1.age=24;            //person1.age is now 24
console.log(person2.age);  //24.
//got the value by going up the prototype chain

person2.age=25;   //creates local prop age for person2
//(person1 is untouched)

console.log(person1);  // { name: "John", age: 24 }
console.log(person2);  // { age: 25 }

person2.__proto__ === person1              //returns true.
Object.getPrototypeOf(person2) === person1 //returns true
```

When the code asks for an object member (property or method) in person2, the JS runtime looks into the object itself, and if it can't find it, then looks up the prototype chain. So for something like `person2.notDefined`, it starts looking, in `person2`, then up the prototype chain in `person1`, then up to the `Object` prototype. Finally, it reaches the top of the chain and it returns `undefined`.

Prototypes with constructor function

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.greet = function() {
    return (`Hello from ${this.name}.`);
  };
}

let person1 = new Person('John', 23);
person1.__proto__ === Person.prototype;  //true
person1.__proto__.__proto__ === Object.prototype;  //true

person1.valueOf();  //method found in the Object prototype
```

`person1` has a prototype pointing to the `Person` prototype, which itself is pointing to the `Object` prototype.

When we try to run the method `valueOf()`, it firsts looks in `person1`. It doesn't find it so it walks up the prototype chain, to `Person` prototype, then up again to the `Object` prototype, where it finally finds it. (If it hadn't found it, it would continue up the prototype chain, until it reaches the top, where it would return `undefined`.)

Best practice: define methods in prototypes

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

Person.prototype.greet = function() {
  return (`Hello from ${this.name}.`);
};
 //greet is defined once, for all instances

let person1 = new Person('John', 23);
let person2 = new Person('Sarah', 42);
person1.greet();
person2.greet();
//These functions are the same
```

This approach is better. We should put all methods, and static properties in the prototype.

Inheritance using prototypes (hard)

A derived constructor can call a base constructor to initialize an object. It will inherit all props from the base.

```
function Employee(name, age, jobTitle) {
  Person.call(this, name, age);

  this.jobTitle = jobTitle;
}
```

The Employee constructor is calling the Person Constructor, and also adds a job property.

We now need to modify the prototype chain, and its constructor property, to add the parent to the prototype chain.

```
Employee.prototype = Object.create(Person.prototype);

Object.defineProperty(Employee.prototype, 'constructor', {
    value: Employee,
    enumerable: false, // doesn't appear in 'for in' loop
    writable: true });
```

Note: This is too hard. Instead, use the new class syntax, shown below.

Best Practice: use the new class syntax

The new class syntax is simpler for methods (it hides the prototype syntax).

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age  = age;
   }

  greet() {
    return (`Hello from ${this.name}.`);
  };
}
```

The new syntax is also simpler for inheritance. No need to modify the prototype chain. (The chain is properly set by the class syntax)

```
class Employee extends Person {
  constructor(name, age, jobTitle) {
    //call the constructor of the base class
    super(name, age);

    //initialize jobTitle
    this.jobTitle = jobTitle;
  }
}
```

Getters, setters and private fields

Hide a private field behind two methods. A *getter* returns the current value of the variable. A *setter* changes the value of the variable to the one it defines. We can add stuff to the getter or setter method: logging, transformations, notification, validation, etc.

```
class Employee extends Person {
  constructor(name, age, jobTitle) {
    super(name, age);
    this.#jobTitle = jobTitle;  //# is a private field
  }

  get jobTitle() {
    return this.#jobTitle;
  }

  set jobTitle(newJob) {
    this.#jobTitle = newJob;
  }
}
```

static members

Static method calls are made directly on the class. They are not callable on instances of the class. Static methods are often used to create utility functions.

```
class Utils {
  static double(num) {
    return num * 2;
  }
}

const util =  new Utils();
util.double(5); //TypeError: util.double not a function

Utils.double(5); // returns 10;
```

JavaScript built-in objects

- **Object** used to store keyed collections and more complex entities
- **Function** an object that represents a function
- **Boolean** primitive data type for boolean values (true, false)
- **Number** primitive data type for numeric values (5, 3.1415)
- **BigInt** primitive data type for int values (>) 64 bits) (5n, 42n)
- **String** primitive data type for text ("hello", "5")
- **Symbol** primitive data type used as an identifier for object properties
- **Array** creates a list-like object
- **Error** base class for errors

- **Date** represent a single moment in time
- **RegExp** used for matching text with a pattern
- **JSON** utility to convert objects to JSON, and to parse JSON to objects
- **Math** Math constants and functions (Uses Numbers, not BigInt)
- **Map** object that holds key-value pairs
- **Set** object that is a collection of values
- **Promise** represents the eventual completion (or failure) of an asynchronous operation.
- **Generator** object returned by a generator function. (Iterable)
- **Intl** namespace for the ECMAScript Internationalization API
- **WebAssembly** namespace for all WebAssembly-related functionality
- **ArrayBuffer** represents a generic, fixed-length raw binary data buffer
- **DataView** read and write multiple num types in binary ArrayBuffer
- **Proxy** used to define custom behavior for fundamental operations
- **Reflect** similar to proxy, but with static methods