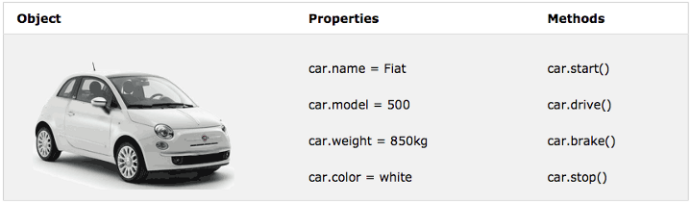
**JavaScript Objects – What are they?**

In JavaScript, almost “everything” is an object:

* **Booleans** can be objects (if defined with the **new** keyword)
* **Numbers** can be objects (if defined with the **new** keyword)
* **Strings** can be objects (if defined with the **new** keyword)
* **Dates** are always objects
* **Maths** are always objects
* **Arrays** are always objects
* **Functions** are always objects
* **Objects** are always objects

For example, in real life, a car is an **object**.

A car has **properties** like weight and color, and **methods** like start and stop:



All cars have the same **properties**, but the **property** **values** differ from car to car.

All cars have the same **methods**, but the methods are performed **at** **different** **times**.

Objects are variables too. But objects can contain many values.

**Creating and Calling objects Properties**

**Creating Objects**

There are 3 was of assigning many values (Fiat, 500, white) to a variable named car:

* **Type 1**: creating a variable and using **:** to assign the property values:

const car = {type:"Fiat", model:"500",color:"white"};

* **Type 2**: using the **[]** to refer to the property and **=** to assign the property value.

const myCar = {}

myCar["make"] = "Ford";

myCar["model"] = "Mustang";

myCar["year"] = 1969;

* **Type 3**: Using the **.** to refer to the property and **=** to assign the property value.

const myCar = {}

myCar.make = "Ford";

myCar.model = "Mustang";

myCar.year = 1969;

**Calling Properties of an Object**

There are 3 ways to access the property value from the object:

* **Type 1**: Using the **.** to refer to the property:

myCar.make

* **Type 2**: using the **[]** and the name of the property.

myCar["make"]

* **Type 3**: using the **[]** and the position of the propery in the array:

myCar[0]

**Nested Objects**

We can also access properties inside nested objects as below shown below.

const person = {

    name: 'john',

    age: 25,

    siblings: ['anna','peter'],

    greet (name) { console.log(`Hello, my name is ${name}`)},

    job: {

        title: 'developer',

        company: {

            name: 'coding addict',

            address: '123 main street'

        }

    }

}

console.log(person.job.company.name)



**The *this* keyword**

In JavaScript, the **this** **keyword** refers to an **object**. Which object depends on how this is **being** **invoked** - **not** **on** **where** **it’s** **defined** - i.e. what is on the left side of the dot **.** when it’s called:

* In an **object** / **object** **method**, **this** refers to the **object**.
* In a **constructor function**, **this** refers the new instance (i.e. new object) that was created
* **Alone**, **this** refers to the **global** **object**.
* In a **function that’s not part of an object**, **this** refers to the **global** **object**.
* In a **function inside an object**, **this** refers to that **object**
* In an **event**, **this** refers to the **element** **that** **received** **the** **event**.

**Using *this* on an Object**

**this** refers to the **john** object because it is being invoked by that object.

const john = {

    firstName: "John",

    lastName : "Doe",

    fullName : function() {

      console.log(this)

      console.log(`My full name is ${this.firstName} ${this.lastName}`)

    }

  };

  john.fullName()



**Using *this* in Constructor Function**

**this** refers the new instance (i.e. new object) that was created, which in this case was the new Person object with the first name “John”.

function Person(firstName) {

  this.firstName = firstName;

  console.log(this);

}

const john = new Person ('john')



**Using *this* alone**

Using **this** points at the **window** object by default.

console.log(this)



**Using *this* in a Function**

**this** points at the **window** object by default.

function showThis() {

    console.log(this)

}

showThis()



But, if the method is called **inside** **an** **object**, it will point to the object that called it.

const john = {

  name: 'john',

  showThis: function() {

    console.log(this)

  }

}

john.showThis()



Also, if that function is called as a **callback function** inside an object, it will **point back to the global object**. This is because this function is just a regular function – it’s not a method inside the object – and thus points at the global object **window**.

|  |  |
| --- | --- |
| const john = {    name: 'john',    showThis: function() {      console.log(this);      setTimeout(function() {        console.log(this)      },1000);    }  };  john.showThis() |  |

So, you can see that this refers to different objects depending on where it’s called:

* The **this** refers to the **john** object because the function that it belongs to has john object as **First Outer Scope**.
* The second **this** refers to the **global** **object** because the **john** object is only the **Second Outer Scope**, and thus it lost the link. Now it’s just a **regular** **function** that points to the **global** **object**.

**Using *this* in an Event**

Using **this** in an **Event**: **this** refers to the **HTML element that received the event**, in this case the **button** element.

<button onclick="console.log(this)">Click Me<button>

Which will console log the button, as seen below.



However, it will point to the **global** **object** **window** if inside an anonymous function. This is because it loses the link to its parent **btn2**, which is not directly above it anymore.

function showThis() {

    console.log(this)

}

const btn1 = document.querySelector('.btn1')

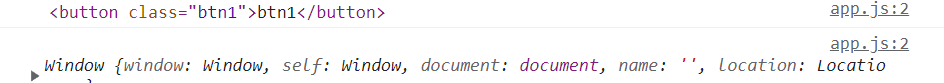
const btn2 = document.querySelector('.btn2')

btn1.addEventListener('click',showThis)

btn2.addEventListener('click',function(){

    showThis()

})



**Factory Functions**

There are a few problems with creating objects using the Object Literal Syntax.

If you want to define 2 person objects, you will have to create 2 person objects:

const person1 = {

  firstName: "John",

  lastName : "Doe",

  fullName : function() {

    return this.firstName + " " + this.lastName;

  }

};

const person2 = {

  firstName: "Peter",

  lastName : "Pan",

  fullName : function() {

    return this.firstName + " " + this.lastName;

  }

};

Now imagine **function()** has a lot of code lines, and there are more methods in the same object. This is not pratical, as you would have to copy paste, and in case there is a mistake, you would have to correct that in every object.

Just like a factory produces products, **Factory Functions** produce objects. To create a factory function that creates person object, just create a **function() createPerson {}** and copy paste the person object into it@

function() createPerson {

  const person = {

    firstName: "Peter",

    lastName : "Pan",

    fullName : function() {

      return this.firstName + " " + this.lastName

    }

  };

}

We want to return this object (person) whenever we call the **createPerson** function, so we add **return person**.

function() createPerson {

  const person = {

     firstName: "Peter",

     lastName : "Pan",

      fullName : function() {

        return this.firstName + " " + this.lastName

     }

  };

return person;

}

We don’t need the **const person** object defined as we are not going to reference it anywhere, so we can eliminate it, and just return what is inside the function itself. So, whenever we call **createPerson** **function**, it will call the person **object**.

function() createPerson {

  return {

     firstName: "Peter",

     lastName : "Pan",

     fullName : function() {

        return this.firstName + " " + this.lastName

     }

  };

}

We still have the properties hardcoded, which means we can’t easily change their values.

function(firstName, lastName) createPerson {

  return {

     firstName: firstName,

     lastName : lastname,

     fullName : function() {

       return this.firstName + " " + this.lastName

     }

  };

}

In modern JavaScript, if the **key** and the **value** are the same, we can just keep the key.

function(firstName, lastName) createPerson {

  return {

    firstName,

    lastName,

    fullName : function() {

      return this.firstName + " " + this.lastName

    }

  };

}

When a function is inside of an object (**Method**), its notation can be written as a normal function but without the keyword **function**:

function createPerson(firstName, lastName)  {

  return {

     firstName,

     lastName,

     fullName() {

       return this.firstName + " " + this.lastName

     }

  };

}

We can now call the **Factory Function** **createPerson** to create a **person** **object** (**person1**) and assign values to its keys:

const person1= createPerson (“Peter”, “Pan”)

If we now type **console.log(person1)**, we get:

{firstName: 'Peter', lastName: 'Pan', fullName: ƒ}

And if we type **console.log(person1.fullName)**, we get:

Peter Pan

**Constructor Functions**

Another pattern to create objects is called **Constructor** **Functions**.

When creating **Constructor** **Functions** we should use **Pascal** **Notation**:

* **Pascal** **Notation**: OneTwoThree
* Camel Notation: oneTwoThree

The equivalent of a **Constructor** **Function** for the Factory Function presented before would be:

function Person (firstName, lastName) {

  return {

    this.firstName = firstName;

    this.lastName = lastName;

    this.fullName = function() {

      return this.firstName + " " + this.lastName;

    }

  };

}

In **Constructor** **Functions**, we use **this** to initialize an object.

**this** is a reference to the object that is executing the piece of code. Until we call an object, the object still doesn’t exist.

In order to create a **Person** object:

const person1 = new Person(“Peter”,”Pan”);

The **new** operator does 3 actions:

1. creates an **empty** **object** **person1 = {}**

2. It will make **this** to **point** to that object.

3. It will return the object from the Constructor Function (**person1**).

**Prototype Extensions**

The **prototype** property allows to add new **properties** and **methods** to an existing **object** **constructor** **function**.

For example, the following code would log **English**:

function Person(first, last, age, eyecolor) {

  this.firstName = first;

  this.lastName = last;

  this.age = age;

  this.eyeColor = eyecolor;

}

Person.prototype.nationality = "English";

const myFather = new Person("John", "Doe", 50, "blue");

console.log(myFather.nationality)

The following code shows how to use prototype extensions to add a method to an object constructor function. It will log **John Doe**.

function Person(first, last, age, eyecolor) {

  this.firstName = first;

  this.lastName = last;

  this.age = age;

  this.eyeColor = eyecolor;

}

Person.prototype.name = function() {

  return this.firstName + " " + this.lastName;

};

console.log("My father is " + myFather.name())

**Classes in JavaScript**

**Classes** were introduced in  ES6 to provide a **cleaner** way to follow object-oriented programming patterns.

Before classes, we used **constructor** **functions** to do OOP in JavaScript. Have a look at the example below:

function Pen(name, color, price) {

  this.name = name;

  this.color = color;

  this.price = price;

}

const pen1 = new Pen("Marker", "Blue", "3");

console.log(pen1);

The above code shows a **Pen** constructor function that has name, color, and price properties. We are using the **new** keyword with the **Pen** constructor to create an object **pen1**.

Now let's say we want to add a new function to the **Pen** constructor. To do this we need to add the function into the prototype property of **Pen**. Have a look at the **showPrice** function below:

function Pen(name, color, price) {

  this.name = name;

  this.color = color;

  this.price = price;

}

const pen1 = new Pen("Marker", "Blue", "3");

Pen.prototype.showPrice = function(){

  console.log(`Price of {this.name} is {this.price}`);

}

pen1.showPrice();

We can re-create the above example with the help of the **class** keyword. Have a look at the below code:

class Pen {

  constructor(name, color, price){

      this.name = name;

      this.color = color;

      this.price = price;

  }

  showPrice(){

      console.log(`Price of {this.name} is {this.price}`);

  }

}

const pen1 = new Pen("Marker", "Blue", "3");

pen1.showPrice();

We have achieved the same results but with much cleaner syntax. The addition of a new member function like **showPrice** is much easier as compared to adding a function directly into the constructor's prototype.

**Syntax of a Class**

**Class** methods are created with the same syntax as object methods. Use the keyword **class** to create a class.

Always add a **constructor()** method. The **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.

class ClassName {

  constructor() { ... }

  method\_1() { ... }

  method\_2() { ... }

  method\_3() { ... }

}

**Callback Functions**

A **Callback** **Function** is passed as an argument to another function.

These functions that are able to take other functions as argument are called **Higher Order Functions**.

Suppose you want to create a calculator function and, depending on the operation you want, it would call a different function (add, subtract, multiply or divide).

First, you would create the 4 functions that return each operation:

function add(n1,n2) {

  return n1+n2;

}

function subtract(n1,n2) {

  return n1-n2;

}

function divide(n1,n2) {

  return n1/n2;

}

function multiply(n1,n2) {

  return n1\*n2;

}

Then you need the **Higher Order Function**, which will do the calculation taking into consideration the operator:

function calculator (n1, n2, operation) {

  return operation(n1,n2);

}

**operation** is the **Callback** **Function**, i.e. it is going to call one of the other functions.

For example, if we want to do a multiplication by calling the higher order function, we do:

calculator (2, 3, multiply)

**Call Method**

With the **call()** method, you can write a method that can be used on different objects.

In the example below, two objects have been created: **john** and **susan**.

const john = {

    name: 'john',

    age: 24,

    greet () {console.log(`Hello, I'm ${this.name} and I'm ${this.age} years old`)}

}

const susan = {

    name: 'susan',

    age: 24

}

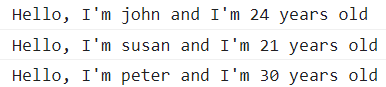
We can access the function inside the john object using the call method, and thus, using that function on another object.

All we need to do is add the **john** object to the left when calling it, and pass the object we want to reference as a parameter. We can also create new objects when calling the function, as shown below with the peter object.

john.greet();

john.greet.call(susan);

john.greet.call({ name: 'peter', age: 30});



The function can also be outside the object. In that case, we don’t need the john object when calling the function.

const john = {

    name: 'john',

    age: 24,

}

const susan = {

    name: 'susan',

    age: 21

}

function greet () {

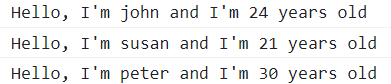
    console.log(`Hello, I'm ${this.name} and I'm ${this.age} years old`)

}

greet.call(john);

greet.call(susan);

greet.call({ name: 'peter', age: 30});



**Bind Method**

The **bind()** method is very similar to the **call()** method, except that it doesn’t run instantly.

const john = {

    name: 'john',

    age: 24,

}

function greet() {

    console.log(`Hello, I'm ${this.name} and I'm ${this.age} years old`)

}

const johnGreet = greet.bind(john);

johnGreet();



**Using bind() in Event Listeners**

**bind()** can be a very useful method when you want to use a function in a event listener, but you want it to point at the object instead of event listener target.

In the example below, we want to increase the counter by calling the function **increment()** each time the button is clicked. But you can see from the console log that **this** is pointing at the button, and we don’t want to increment the button, we want to increment the **count** property.

const counter = {

    count: 0,

    increment() {

        console.log(this);

        this.count++;

        console.log(this.count);

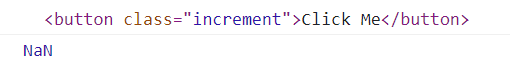
    }

}

const btn = document.querySelector('.increment')

btn.addEventListener('click',counter.increment)

Result:



To correct this, we can use the **bind()** to change the target of this.

const counter = {

    count: 0,

    increment() {

        console.log(this);

        this.count++;

        console.log(this.count);

    }

}

const btn = document.querySelector('.increment')

btn.addEventListener('click',counter.increment.bind(counter))

