Objects

A JavaScript object is an entity having state and behavior (properties and method). For example: car, pen, bike, chair, glass, keyboard, monitor etc.

JavaScript is an object-based language. Everything is an object in JavaScript.

JavaScript is template based not class based. Here, we don't create class to get the object. But, we direct create objects.

Creating Objects in JavaScript

There are 3 ways to create objects.

- **By object literal**
- **By creating instance of Object directly (using new keyword)**
- **By using an object constructor (using new keyword)**

1) JavaScript Object by object literal

The syntax of creating object using object literal is given below:

object={property1:value1,property2:value2.....propertyN:valueN}

As you can see, property and value is separated by : (colon).

Let's see the simple example of creating object in JavaScript.

```
<script>
emp={id:102,name:"Shrikant",salary:40000}
document.write(emp.id+" "+emp.name+" "+emp.salary);
</script>
```

Output of the above example

102 Shrikant Kumar 40000

2) By creating instance of Object

The syntax of creating object directly is given below:

```
var objectname=new Object();
```

Here, **new keyword** is used to create object.

Let’s see the example of creating object directly.

```
<script>
var emp=new Object();
emp.id=101;
emp.name="Shrikant";
emp.salary=50000;
document.write(emp.id+" "+emp.name+" "+emp.salary);
</script>
```

Output of the above example

101 Shrikant 50000

3) By using an Object constructor

Here, you need to create function with arguments. Each argument value can be assigned in the current object by using this keyword.

The **this keyword** refers to the current object.

The example of creating object by object constructor is given below.

```
<script>
function emp(id,name,salary){
this.id=id;
this.name=name;
this.salary=salary;
}
e=new emp(103,"Shrikant ",30000);

document.write(e.id+" "+e.name+" "+e.salary);
</script>
```

Output of the above example

103 Shrikant 30000

Defining method in JavaScript Object

We can define method in JavaScript object. But before defining method, we need to add property in the function with same name as method.

The example of defining method in object is given below.

```
<script>
function emp(id,name,salary){
this.id=id;
this.name=name;
this.salary=salary;

this.changeSalary=changeSalary;
function changeSalary(otherSalary){
this.salary=otherSalary;
}
}
e=new emp(103,"Shrikant",30000);
document.write(e.id+" "+e.name+" "+e.salary);
e.changeSalary(45000);
document.write("<br>"+e.id+" "+e.name+" "+e.salary);
</script>
```

Output of the above example

103 Shrikant 30000

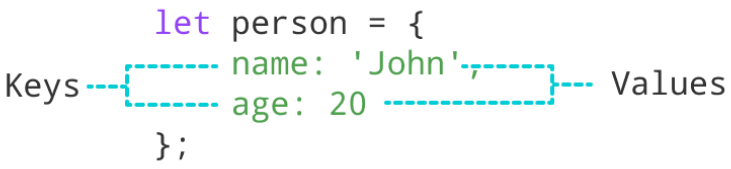
103 Shrikant 45000

JavaScript Object Properties

In JavaScript, "key: value" pairs are called **properties**. For example,

```
let person = {
name: 'John',
age: 20
};
```

Here, `name: 'John'` and `age: 20` are properties.



JavaScript object properties

Accessing Object Properties

You can access the **value** of a property by using its **key**.

1. Using dot Notation

Here's the syntax of the dot notation.

```
objectName.key
```

For example,

```
const person = {
name: 'John',
age: 20,
};
// accessing property
console.log(person.name); // John
```

2. Using bracket Notation

Here is the syntax of the bracket notation.

```
objectName["propertyName"]
```

For example,

```
const person = {
name: 'John',
age: 20,
};
```

```
// accessing property
console.log(person["name"]); // John
```

JavaScript Nested Objects

An object can also contain another object. For example,

```
// nested object
const student = {
name: 'John',
age: 20,
marks: {
science: 70,
math: 75
}
}

// accessing property of student object
console.log(student.marks); // {science: 70, math: 75}

// accessing property of marks object
console.log(student.marks.science); // 70
```

In the above example, an object `student` contains an object value in the `marks` property.

JavaScript Methods and this Keyword

In JavaScript, objects can also contain functions. For example,

```
// object containing method
const person = {
name: 'John',
greet: function() { console.log('hello'); }
};
```

In the above example, a `person` object has two keys (`name` and `greet`), which have a string value and a function value, respectively.

Hence basically, the JavaScript **method** is an object property that has a function value.

JavaScript Built-In Methods

In JavaScript, there are many built-in methods. For example,

```
let number = '23.32';
let result = parseInt(number);

console.log(result); // 23
```

Here, the `parseInt()` method of Number object is used to convert numeric string value to an integer value.

Adding a Method to a JavaScript Object

You can also add a method in an object. For example,

```
// creating an object
let student = { };

// adding a property
student.name = 'John';
```

```
// adding a method
student.greet = function() {
console.log('hello');
}

// accessing a method
student.greet(); // hello
```

In the above example, an empty `student` object is created. Then, the `name` property is added. Similarly, the `greet` method is also added. In this way, you can add a method as well as property to an object.

JavaScript Constructor Function

In JavaScript, a constructor function is used to create objects. For example,

```
// constructor function
function Person () {
```

```
this.name = 'John',
this.age = 23
}

// create an object
const person = new Person();
```

In the above example, `function Person()` is an object constructor function.

To create an object from a constructor function, we use the `new` keyword.

Note: It is considered a good practice to capitalize the first letter of your constructor function.

Create Multiple Objects with Constructor Function

constructor function. For example,

```
// constructor function
function Person () {
this.name = 'John',
this.age = 23,

this.greet = function () {
console.log('hello');
}
}

// create objects
const person1 = new Person();
const person2 = new Person();

// access properties
console.log(person1.name);  // John
console.log(person2.name);  // John
```

In the above program, two objects are created using the same constructor function.

JavaScript this Keyword

In JavaScript, when `this` keyword is used in a constructor function, `this` refers to the object when the object is created. For example,

```
// constructor function
function Person () {
this.name = 'John',
}

// create object
const person1 = new Person();

// access properties
console.log(person1.name);  // John
```

Hence, when an object accesses the properties, it can directly access the property as `person1.name`.

JavaScript Constructor Function Parameters

You can also create a constructor function with parameters.

For example,

```
// constructor function
function Person (person_name, person_age, person_gender)
{

// assigning  parameter values to the calling object
this.name = person_name,
this.age = person_age,
this.gender = person_gender,

this.greet = function () {
return ('Hi' + ' ' + this.name);
}
}

// creating objects
const person1 = new Person('John', 23, 'male');
```

```
const person2 = new Person('Sam', 25, 'female');

// accessing properties
console.log(person1.name); // "John"
console.log(person2.name); // "Sam"
```

In the above example, we have passed arguments to the constructor function during the creation of the object.

```
const person1 = new Person('John', 23, 'male');
const person2 = new Person('Sam', 25, 'male');
```

This allows each object to have different properties. As shown above,

console.log(person1.name); gives John

console.log(person2.name); gives Sam

Create Objects: Constructor Function Vs Object Literal

Object Literal is generally used to create a single object. The constructor function is useful if you want to create multiple objects. For example,

```
// using object literal
let person = {
name: 'Sam'
}
// using constructor function
function Person () {
this.name = 'Sam'
}

let person1 = new Person();
let person2 = new Person();
```

Each object created from the constructor function is unique. You can have the same properties as the constructor function or add a new property to one particular object. For example,

```
// using constructor function
function Person () {
this.name = 'Sam'
```

```
}

let person1 = new Person();
let person2 = new Person();

// adding new property to person1
person1.age = 20;
```

Now this `age` property is unique to `person1` object and is not available to `person2` object.

However, if an object is created with an object literal, and if a variable is defined with that object value, any changes in variable value will change the original object. For example,

```
// using object lateral
let person = {
name: 'Sam'
}

console.log(person.name); // Sam

let student = person;
```

```
// changes the property of an object
student.name = 'John';

// changes the origins object property
console.log(person.name); // John
```

When an object is created with an object literal, any object variable derived from that object will act as a clone of the original object. Hence, any change you make in one object will also reflect in the other object.