Objects and Timing Events

OBJECTS

JavaScript objects are a collection of properties in a key-value pair. These objects can be understood with real-life objects, like similar objects have the same type of properties, but they differ from each other.

Eg., let's say a 'ball' is an object and has properties like 'shape' and 'radius'. So ***every ball will have the same properties, but different balls will have different values*** to them.

Some important points about objects are -

- Object contains ***properties separated with a comma(,).***
- Each property is represented in a ***key-value pair.***
- ***Key and value are separated using a colon(:).***
- The ***key can be a string or variable name*** that does not contain any special characters, except underscore(_).
- The ***value can contain any type of data - primitive, non-primitive,*** and even a ***function.***
- The ***objects are passed by reference to a function.***

An example of an object is -

```
var obj = {
   key1: "value1",
   key2: 12345,
   "key3": true,
   key4: function() { /* Something Here */ }
}
```

- **Creating an Object**

  The object can be created in two ways -

  - **Using curly brackets -** We can create empty object as - `var obj = {};` and an object with some initial properties as - `var obj = {key1: value1, ..., keyN: valueN}`

- **Using new operator -** We can create empty object as - `var obj = new Object();` and an object with properties as - `var obj = new Object({key1: value1, ..., keyN: valueN})`

The properties can be created at the time of creating an object and also after that. ***Both creating and accessing the properties share similar syntax.***

- **Creating and Accessing Properties**

The **properties are created in a key-value pair,** but there are some restrictions in the way some keys are created. There are two ways to create and access properties:

- **Using a dot operator -** You can use the dot operator only when the property name starts with a character. Property can be accessed like - `obj.propertyName.` Similarly, you can create property like - `obj.propertyName = value`
- **Using a square bracket -** You need to use a square bracket when the key name starts with a number. If the name contains a special character then it will be stored as a string. Property is accessed like - `obj["propertyName"].` Similarly, you create property like - `obj["propertyName"] = value`

You can also **set the function as the value** to the key. So the key then becomes the method name and ***needs parentheses to execute***. So you can execute the method like **-** `obj.methodName()` and `obj["methodName"]().`

NOTE: *If you access a property that has not been defined then 'undefined' is returned.*

- **Deleting Property**

You can ***remove the property of the object*** using the **'delete'** operator followed by the property name. You can either ***use dot operator*** or ***square bracket notation.*** The syntax is -

```
delete obj["objectName"]
```

OR

```
delete obj.objectName
```

- **How Objects Are Stored**

There are two things that are very important in objects -

- Objects are **stored in heap.**
- Objects are **reference types**.

These two are important in the regard that ***object variables point to the location*** where they are stored. This means that ***more than one variable can point to the same location***.

Until now, you are creating new objects everytime like -

```
var item1 = { name: 'banana' };
var item2 = { name: 'banana' };
```

The above two lines will create two different objects are not therefore equal -

```
item1 == item2; // Returns - false
item1 === item1; // Returns - false
```

But, if you assign one object to another like - `item2 = item1;` then the value of 'item1' gets assigned to 'item2' and therefore, they both will point to the same location -

```
item1 == item2; // Returns - true
item1 === item1; //Returns - true
```

OBJECTS

JavaScript provides a special form of a loop to traverse all the keys of an object. This loop is called for**...in**' loop.

The syntax for '***for-in***' loop is -

```
for (variable in object) {
   /* Statements */
}
```

Here the **'variable'** ***gets assigned the property name*** on each iteration and **'object'** ***is the object*** that you want to iterate. Use the ***square bracket notation with 'variable' to access the property values.***

The ***iteration may not be in a similar order as to how you see properties in the objec***t or how you have added them. Because the objects are ordered in a special manner.
The ***property names as integers are iterated first*** in ascending order. Then the other names are iterated in the order they were added.

The below code shows how you can iterate using the **'for-in'** loop -

```
for (key in obj) {
   console.log(i, ":", obj[i]);
}
```

It will print the following lines on the console -

```
key1 : value1
key2: 12345
key3: true
key4: function key4()
```

There are two more ways to access all the keys, but it will return an array of keys -

- **Object.keys(obj)**
- **Object.getOwnPropertyNames(obj)**

EXTRA: You can read about the other two ways from the links below -
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames

NESTED OBJECTS

We have already discussed that the ***value of an object's property can be anything.*** So we can have ***objects inside an object*** and they are called **nested objects.** We can have ***any number nesting inside an object,*** i.e. an object can contain another object, which can also contain another object, and so on.

Eg. a nested object is -

```
var student = {
   name: "Anjali",
   class: 5,
   roll: "016-115-19",
   address: {
       city: "New Delhi",
       pincode: 110063
   }
}
```

Here, 'address' is a nested object.

- **Creating Nested Objects**

  You create a nested object as you have created other properties, but cannot create a property of the nested object. This means `obj.propertyName.nestedProperty1 = value1` ***is invalid and gives an error.***

  Instead you create nested object as -

```
obj.propertyName = { nestedProperty1 = value1, ..., nestedPropertyN = valueN }
```

- **Accessing Nested Objects**

  The nested objects can be accessed using multiple dot operator or square brackets notation like -

```
obj.propertyName.nestedProperty1
```

  OR

```
obj["propertyName"]["nestedProperty1"]
```

ARRAY AS OBJECT

Arrays are actually objects. If you use the '**typeOf()**' method on an array, you will see that it will return "**object**". If you see an array on a console, **they are actually key-value pairs**, with the ***positive integers as the keys***.
Arrays can also store properties just like objects. Eg., `array["one"] = 1;` will store this property inside the array and can access it like - `array.one; or array["one"];`.

But if arrays are the same as objects, then what is the difference? Well, ***arrays are somewhat different than objects***. These difference are summarized in the points -

- Arrays ***have a 'length' property*** that objects do not have.
- You can access the values of the arrays like - ***array[0]; or array["0"];,*** whereas in objects you have to use **double quotes** ("") only.
- Only when you use an integer as a key, it will change the 'length' property.

- Adding a non-integer key will not have any effect on the length's property.

So we can say that ***arrays work both like objects and arrays*** (from other languages like Java).

***NOTE**: Length property will be set according to the maximum integer key of the array.*
For eg:

```
var arr = [1,2,3,4,5,6];
console.log(arr.length);
//output = 6
```

- **Using the for...in loop to Iterate**

  Since **arrays are also objects,** you can use a **'for-in'** loop to traverse it. Traversing the array using the 'for-in' loop is the same as traversing an object.

  There is something interesting about arrays you need to know. Let's say that you have an array like -

```
var arr = [10, 20, 30];
```

  and you add another property like -

```
arr["four"] = 40;
```

  then if you print an array on the console like - `console.log(arr);`, it will show the array as - Array(3) [10, 20, 30].

  But, it also ***contains the property*** "`four: 40`", ***but it does not show in the array***. But if you use a 'for-in' loop to traverse it, you can traverse all the properties.

```
for (var i in arr) {
   console.log(i, ":", arr[i]);
}
```

  you will see something like this on the console -

  0: 10
  1: 20
  2: 30
  there: 123

TIMING EVENTS

The **timing events** allow the ***execution of a piece of code at a specific time interval.*** These events/methods are directly available in the DOM Window object, i.e. they are there in the browser. You'll learn about DOM in the next lecture.
Therefore, these are **global methods** and can be **called using a 'window' object or without it.**
Below we have used the timing events that window provides us-

- **setTimeout()**

  The **'setTimeout()'** method is used to ***execute a piece of code after a certain amount of time.*** The piece of code is usually written inside a function.
  The ***function can be passed as a parameter or you can use an anonymous function directly as a parameter.***

  The syntax is -

```
var timeoutID = scope.setTimeout(function, delay, param1,
   param2, ...)
```

  The '**setTimeout()**' method ***returns a positive integer*** which r***epresents the ID of the timer*** created. The use of this ID will be explained later.

  The ***parameters passed*** (specified after the delay time) are ***optional*** and are accessible to the 'function'.

  The **'delay'** is ***written in milliseconds***, so '1000' represents '1' second.

  It is ***optional to use a variable to store the ID***, but it depends upon use cases which will be defined later.

- **setInterval()**

  The **'setInterval()'** method is used to ***execute a piece of code repeatedly with a fixed time interval between each call.***

  The syntax is -

```
var intervalID = scope.setInterval(function, delay, param1,
   param2, ...)
```

The meaning and use of each of the parameters are the same as that of the 'setTimeout()' method.

- **clearTimeout()**

  The **'clearTimeout()'** method is used to **cancel a timeout** established using the 'setTimeout()' method.
  The syntax is - `scope.clearTimeout(timeoutID)`
  The ***'timeoutID' is the ID that 'setTimeout()' method returns***. Passing an invalid ID to this method will not do anything.

  NOTE*: When you don't need to use the 'clearTimeout()' method, then there is no need to store the ID returned by the 'setTimeout()' method.*

- **clearInterval()**

  The **'clearInterval()'** method is used to **cancel the repeating timed action established using 'setInterval()' method.**

  The syntax is **-** `scope.clearInterval(intervalID)`

  The ***'intervalID' is the ID that 'setInterval()' method returns*****.** Passing an invalid ID to this method will not do anything.

  NOTE*: The 'setTimeout()' and 'setInterval()' method share the same pool for storing IDs, which means that you can use 'clearTimeout()' and 'clearInterval()' methods interchangeably. However, you should avoid doing so.*