Extra Notes - JavaScript - Objects

- Dr Nick Hayward

A brief introduction to JavaScript objects, and their general usage.

Contents

- Intro
- Basic usage
- Add values
- Object access
    - single property
    - example output
    - all keys
    - all values
    - all entries
- Get length of an object
- Combine arrays and objects
- Arrays as objects
- References

Intro Objects, as you might imagine, are particularly useful in JS.

In essence, the **object** type includes a compound value, which JS can use to set *properties*, or named locations.

So, a property in an object is an association between a defined **name (or key)** and its value.

Each of these properties holds its own value, which may be defined as any type. Hence its general flexibility in JS development, and its widespread usage.

Basic usage An example JS object may be declared as follows,

```
// declare variable - store object literal
var objectA = {
  a: 49,
  b: 59,
  c: "Philae"
};
```

So, we've just declared a variable to store an **object literal**. This is the curly braces, and the **key:value** pairs.

As JavaScript knows that each property will be a string, we do not have to add encapsulating quotation marks, these are optional syntax.

However, you will need to use such quotation marks if the property name contains multiple words and spaces.

For example,

```
var testObject = {
    "Temple Sites": {
        name: "Philae"
    }
}
```

Add values To add values to an object, we might need to start with an empty object,

```
// create empty object
var testObject = {};
```

This uses the same pattern as creating an empty array, but we use `{}` curly braces instead of `[]` square brackets.

Then, we can add single values to the new object,

```
// create empty object
var testObject = {};
// add new value with dot notation
testObject.archive = 'waldzell';
// add new value with bracket notation
testObject['access'] = 'castalia';
```

Figure 1: JS Object - add some values

Object access Object properties may be accessed using various options and methods.

single property For example, we may initially access a property's value using either **dot** or **bracket** notation,

```
//dot notation
objectA.a;
//bracket notation
objectA["a"];
```

Dot notation is, customarily, the more common option for object usage.

However, *bracket* notation is necessary for many nested objects, and may be the only option depending upon context.

example output In the following example, we may see basic object usage. This includes example access using both *dot* and *bracket* notation.

property a from objectA using dot notation = 49
property c from objectA using bracket notation = Philae

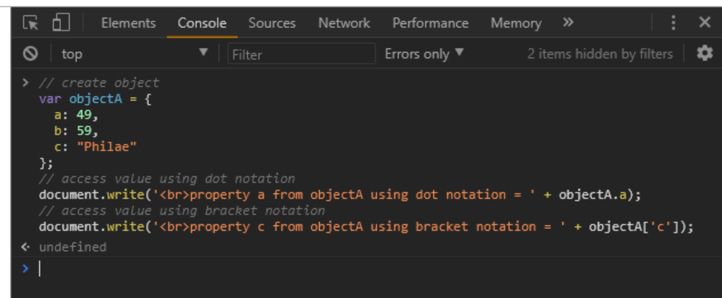

Figure 2: JS Object - example output

```
// create object
var object = {
  archive: 'waldzell',
  access: 'castalia',
  purpose: 'gaming'
};

// output with dot notation
document.write('<br>archive is ' + object.archive);

// output with bracket notation - returns undefined - no index, use property name/key
document.write('<br>access is restricted to ' + object[1]);

// output with bracket notation - correct access with key
document.write('<br>purpose is ' + object['purpose']);
```

archive is waldzell
access is restricted to undefined
purpose is gaming

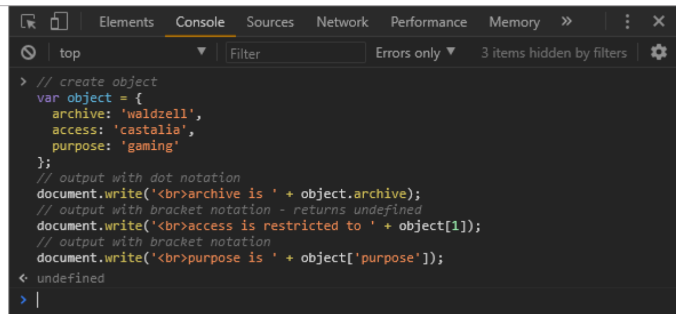

Figure 3: JS Object - example output

all keys So, we've already seen how to access single properties in an object using either dot or bracket notation. However, JS also provides a method, `keys()` to access all the **keys** within our object.

```
// create object
var testObject = {
  archive: 'waldzell',
  access: 'castalia',
  purpose: 'gaming'
};

// get all keys for passed object, testObject
Object.keys(testObject);
```

The `.keys()` method will return an array with the passed object's keys set as the values.

```
> // create object
  var testObject = {
    archive: 'waldzell',
    access: 'castalia',
    purpose: 'gaming'
  };

  // get all keys from passed object
  Object.keys(testObject);
<· ▼(3) ["archive", "access", "purpose"]
      0: "archive"
      1: "access"
      2: "purpose"
      length: 3
    ▶ __proto__: Array(0)
>
```

Figure 4: JS Object - get all keys

all values JS also provides a useful method to access all **values** in a passed object.

We may use the `Object.values()` method to return an array of indexed values from the object.

```
// create object
var testObject = {
  archive: 'waldzell',
  access: 'castalia',
  purpose: 'gaming'
};

// get all values for passed object, testObject
Object.values(testObject);
```

So, the `value()` method will return an array of values for `testObject`. Each value will be pushed to an array with the expected auto-increment for the index value.

In effect, it's similar to manually getting each value, and then calling the `push()` method for each value in an empty array.

```
> var testObject = {
    archive: 'waldzell',
    access: 'castalia',
    purpose: 'gaming'
  };

  // get all values from passed object
  Object.values(testObject);
<· ▾(3) ["waldzell", "castalia", "gaming"] i
     0: "waldzell"
     1: "castalia"
     2: "gaming"
     length: 3
   ▸ __proto__: Array(0)
>
```

Figure 5: JS Object - get all values

all entries JS provides another useful method to access all **entries** in a passed object.

The method `Object.entries()` will return each key and value for each property in the passed object.

```
// create object
var testObject = {
  archive: 'waldzell',
  access: 'castalia',
  purpose: 'gaming'
};

// get all entries for passed object, testObject
Object.entries(testObject);
```

When we execute the `entries()` method, it will return a multidimensional array of keys and values for each property in the passed `testObject` .

Each inner array has the key and value for the corresponding property in the passed object.

```
> var testObject = {
    archive: 'waldzell',
    access: 'castalia',
    purpose: 'gaming'
  };

  // get all entries from passed object
  Object.entries(testObject);
<· ▼(3) [Array(2), Array(2), Array(2)] i
    ▶0: (2) ["archive", "waldzell"]
    ▶1: (2) ["access", "castalia"]
    ▶2: (2) ["purpose", "gaming"]
     length: 3
    ▶__proto__: Array(0)
>
```

Figure 6: JS Object - get all entries

The `Object.entries()` method will return a value regardless of data type.

This might include primitive data types such as *strings*, *numbers*, *booleans*, and data structures such as *arrays*, *sets*, plain *objects*, and so on.

```
var testObject = {
  archive: 'waldzell',
  access: 'castalia',
  purpose: 'gaming',
  games: {
    primary: 'glass bead',
    secondary: 'arithmetica',
    tertiary: 'ultima'
  }
};

// get all entries from passed object
Object.entries(testObject);
```

So, the `entries()` method will still return a multidimensional array of keys and values for `testObject` regardless of data type for each property value.

```
> var testObject = {
    archive: 'waldzell',
    access: 'castalia',
    purpose: 'gaming',
    games: {
      primary: 'glass bead',
      secondary: 'arithmetica',
      tertiary: 'ultima'
    }
  };

  // get all entries from passed object
  Object.entries(testObject);
<· ▼(4) [Array(2), Array(2), Array(2), Array(2)] i
    ▶0: (2) ["archive", "waldzell"]
    ▶1: (2) ["access", "castalia"]
    ▶2: (2) ["purpose", "gaming"]
    ▼3: Array(2)
       0: "games"
      ▶1: {primary: "glass bead", secondary: "arithmetica", tertiary: "ultima"}
       length: 2
      ▶__proto__: Array(0)
     length: 4
    ▶__proto__: Array(0)
>
```

Figure 7: JS Object - get all entries

Get length of an object As an object does not include its own `length` property, in contrast to a standard array, we need to use a different option to find its size.

A quick and easy option is to combine the `keys()` method with an array's `length` property. In effect, we get the keys for all properties in the object, using the `keys()` method, which returns an array of the object's keys. We may then use the `length` property of this array.

For example,

```
// create object
var testObject = {
  archive: 'waldzell',
  access: 'castalia',
  purpose: 'gaming'
};

// get all keys for passed object, testObject
var objectKeys = Object.keys(testObject);
// get length of object using return array for keys
var objectLen = objectKeys.length;
```

```
> // create object
  var testObject = {
    archive: 'waldzell',
    access: 'castalia',
    purpose: 'gaming'
  };

  // get all keys for passed object
  var objectKeys = Object.keys(testObject);
  // get length of object using return array for keys
  var objectLen = objectKeys.length;
  // test output of objectKeys
  objectKeys;
<· ▼(3) ["archive", "access", "purpose"]
      0: "archive"
      1: "access"
      2: "purpose"
      length: 3
    ▶ __proto__: Array(0)
> // test output of objectLen
  objectLen;
<· 3
>
```

Figure 8: JS Object - get object length - v.1

```
> // create object
  var testObject = {
    archive: 'waldzell',
    access: 'castalia',
    purpose: 'gaming'
  };

  // get length of object using return array for keys
  var objectLen = Object.keys(testObject).length;
  // test output of objectLen
  objectLen;
<· 3
> |
```

Figure 9: JS Object - get object length - v.2

The difference between these solutions is the return values we store in variables.

In v.1, we explicitly store the `keys` array in a variable, which may be used elsewhere in the app's logic.

However, in v.2 we only return the length of the passed object. We do not have access to the `keys` array unless we explicitly call that method again.

Arrays as objects In JS, an array is a custom object that contains values, again of any type, in numerically indexed positions.

So, we can store a number, a string, and the array will start at index position `0`. It will then increment by **1** for each new value.

These arrays can also have properties, for example the automatically updated **length** property.

```
var arrayA = [
  49,
  59,
  "Philae"
];
arrayA.length; //returns 3
```

Each value can be retrieved from its applicable index position,

```
  arrayA[2]; //returns the string "Philae"
```

Due to the nature of arrays, as special objects, we could use them as a catch-all solution for storing our values. We could even add our own named properties, thereby mimicking the functionality of an object.

However, this is often considered poor usage, or misuse in many respects, of the functionality of objects and arrays in JavaScript.

Therefore, we can use objects for named properties, and arrays for values with numerically indexed positions.

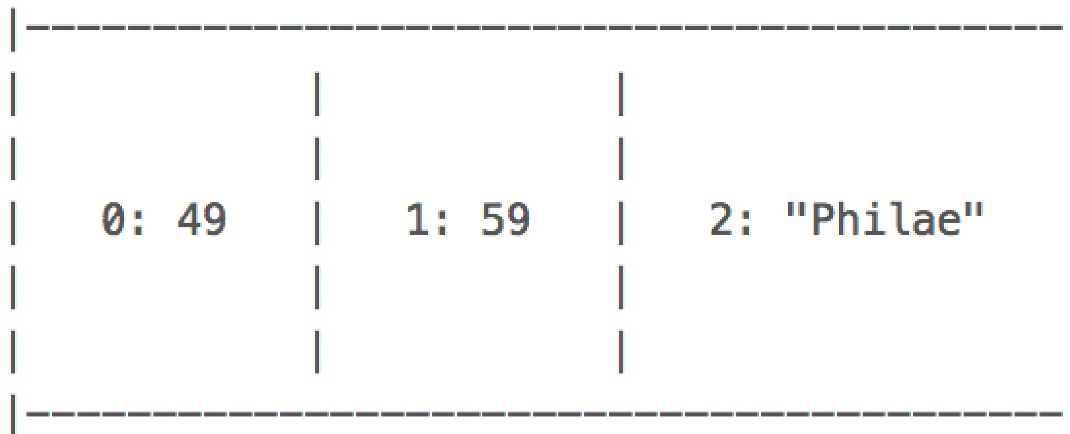

Figure 10: JS Array

Combine arrays and objects Objects and arrays may also be combined in JavaScript - an object in an array, and so on.

For example,

```
// create array with objects
var archives = [
    { name: 'waldzell', access: 'castalia', purpose: 'gaming' },
    { name: 'bodleian', access: 'oxford', purpose: 'research'}
];
```

Then, we can access one of the inner objects,

```
// get first archive object
var firstArchive = archives[0];
```

or perhaps get a single value with the name of the first archive.

```
// get name from first object - bracket notation
var archiveName = firstArchive["name"];
// get name from second object - dot notation for object
var archiveName2 = archives[1].name;
```

first archive is waldzell
second archive is bodleian

```
// create combined array and inner objects
var archives = [
    {name: 'waldzell', access: 'castalia', purpose: 'gaming' },
    { name: 'bodleian', access: 'oxford', purpose: 'research'}
];

// get first archive object
var firstArchive = archives[0];

// get name from first object - bracket notation
var archiveName = firstArchive["name"];
// get name from second object - dot notation for object
var archiveName2 = archives[1].name;

// output archive names to document
document.write('<br>first archive is ' + archiveName);
document.write('<br>second archive is ' + archiveName2);
undefined
```

Figure 11: JS - array and object combined

References

- MDN - Working with Objects
- W3Schools - Objects and Properties