JS | Data Types in JavaScript - Objects

Learning Goals

After this lesson, you will be able to:

- Explain the **key-value** relationship
- Use an `Object` in JavaScript and understand its importance
- Add, remove and modify keys and values in an object
- Access values in an `Object` with dot and bracket notations
- List the properties of an `Object`

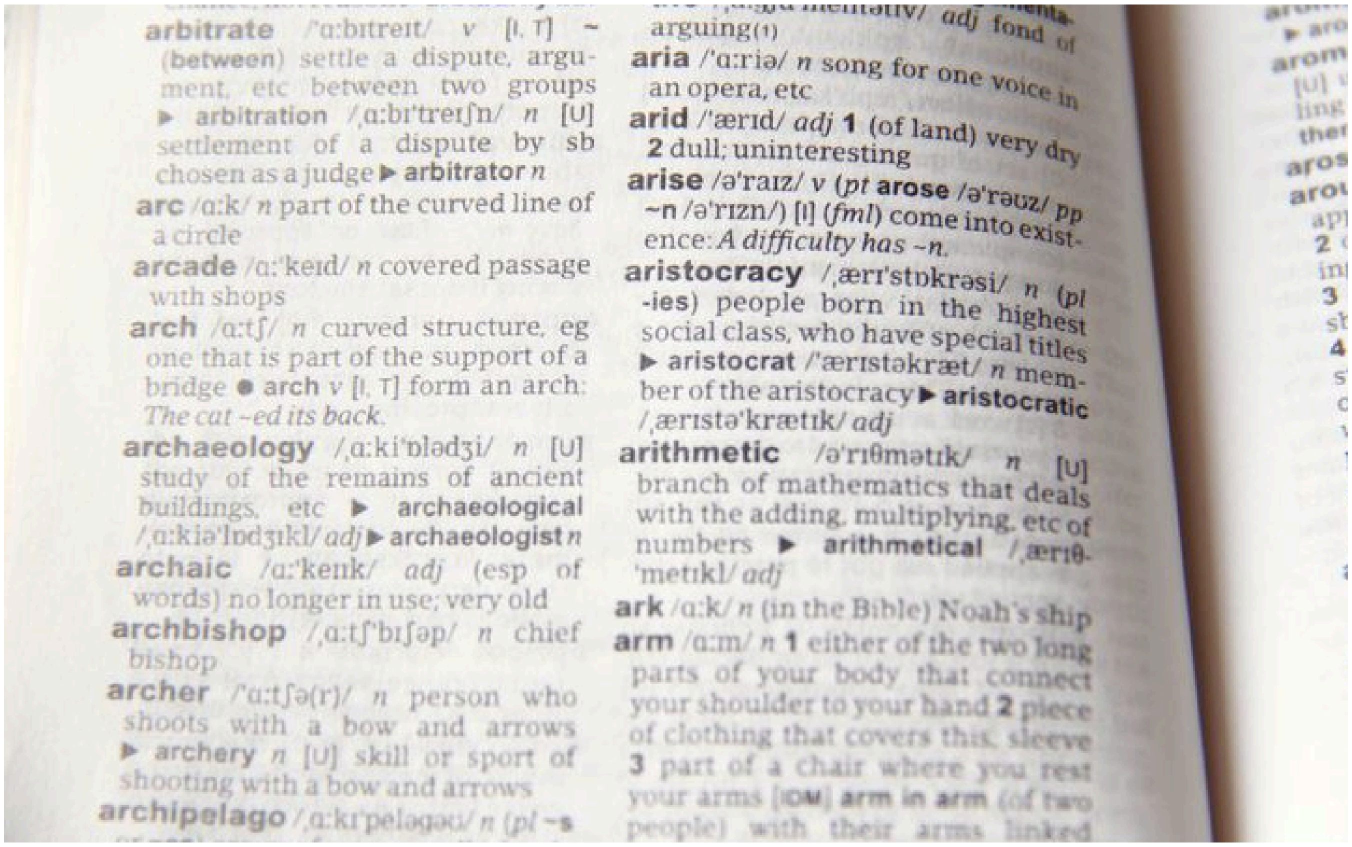
Introduction

Objects are collections of properties and each property is represented with key-value pair. The representation of an object in JavaScript is curly braces `{}`.

The **key-value pair** is a very important concept to understand how `object` data types work in JavaScript.

In programming, a **key-value pair** is a set of two linked data items. The key is a string that identifies a property of an object. It typically corresponds to the name of the property you want to access. **The keys are unique in an object; one key will always have just one value associated to it**.

When you access the property, the object will return the associated value of the indicated key.



mentative/ adj fond of arguing(1)

arbitrate /ˈɑːbɪtreɪt/ v [I, T] ~ (between) settle a dispute, argument, etc between two groups ▶ **arbitration** /ˌɑːbɪˈtreɪʃn/ n [U] settlement of a dispute by sb chosen as a judge ▶ **arbitrator** n

arc /ɑːk/ n part of the curved line of a circle

arcade /ɑːˈkeɪd/ n covered passage with shops

arch /ɑːtʃ/ n curved structure, eg one that is part of the support of a bridge ● **arch** v [I, T] form an arch: *The cat ~ed its back.*

archaeology /ˌɑːkiˈɒlədʒi/ n [U] study of the remains of ancient buildings, etc ▶ **archaeological** /ˌɑːkiəˈlɒdʒɪkl/ adj ▶ **archaeologist** n

archaic /ɑːˈkeɪɪk/ adj (esp of words) no longer in use; very old

archbishop /ˌɑːtʃˈbɪʃəp/ n chief bishop

archer /ˈɑːtʃə(r)/ n person who shoots with a bow and arrows ▶ **archery** n [U] skill or sport of shooting with a bow and arrows

archipelago /ˌɑːkɪˈpeləgəʊ/ n (pl ~s

aria /ˈɑːriə/ n song for one voice in an opera, etc

arid /ˈærɪd/ adj **1** (of land) very dry **2** dull; uninteresting

arise /əˈraɪz/ v (pt **arose** /əˈrəʊz/ pp ~n /əˈrɪzn/) [I] (fml) come into existence: *A difficulty has ~n.*

aristocracy /ˌærɪˈstɒkrəsi/ n (pl -ies) people born in the highest social class, who have special titles ▶ **aristocrat** /ˈærɪstəkræt/ n member of the aristocracy ▶ **aristocratic** /ˌærɪstəˈkrætɪk/ adj

arithmetic /əˈrɪθmətɪk/ n [U] branch of mathematics that deals with the adding, multiplying, etc of numbers ▶ **arithmetical** /ˌærɪθˈmetɪkl/ adj

ark /ɑːk/ n (in the Bible) Noah's ship

arm /ɑːm/ n **1** either of the two long parts of your body that connect your shoulder to your hand **2** piece of clothing that covers this; sleeve **3** part of a chair where you rest your arms [IDM] **arm in arm** (of two people) with their arms linked

The most common example to explain this concept is a Dictionary. The key is the word you are looking for, while the value is the description of that word:

```
{website: "Connected group of pages on the World Wide Web regarded as a single entity,
usually maintained by one person or organization and devoted to a single
topic or several closely related topics."}
```

✔ Each `word` is the key, and the `definition` is the value associated with this key, which is unique in the Dictionary.

Why should we use objects?

Objects are useful to **group values that belong together into a single unit**. Objects store relationships between variables and properties using *key* and *value* associations.

💡 The property's value in an object can be any type we need: strings, numbers, arrays, functions… or even other objects!

This kind of composition is very useful, as we can store variables that hold related information into one `object`.

By using this notation, our code will be much more clear and easy to understand. It will help us to have a more semantic code, and the maintenance of the code will be much easier.

Object definition

Creating objects is super easy, all we need to do is use the curly braces `{` and `}`, and add keys and values to it. This way of creating a new object is known as **object literal syntax**:

```
let someObject = {
  key1: value,
  key2: value,
  key3: value
}
```

In a couple of lessons, you will know what constructors are, but we want you to keep in mind that you can create objects also using **object constructor syntax**:

```
let someObject = new Object();
```

For now we will use **object literal** approach.

We have an object with a history of the Olympic games records:

```
{
  athletics100Men: "Justin Gatlin"
}
```

In this case, we just have one *property*, `athletics100Men`, which will give us the *value* for the Athletics 100 meter men's Olympic record.

- The **key** is `athletics100Men`
- The associated **value** is `Justin Gatlin`

If for some reason, you need to use property names that consist of more than one word, we highly recommend using `camelCase` nomenclature. If you decide to go for multi-word properties with no *camelCase* approach, then you need to put the property in the quotes “”.

```
let olympicRecords = {
  "athletics long jump men": "Mike Powel"
}
```

We will keep using *camelCase* approach as it is the most used.

Objects are literals (like `23` or `false`), so they can be stored in variables. Let’s store our object in the `olympicRecords` variable:

```
let olympicRecords = {
  athletics100Men: "Justin Gatlin"
}
```

If we wanted to store Mike Powel and his long jump record of 8.95m, we could do it by adding another key:

```
let olympicRecords = {
  athletics100Men: "Justin Gatlin",
  athleticsLongJumpMen: "Mike Powel"
}
```

Notice how we separate the `athletics100Men` and the `athleticsLongJumpMen` properties with a **comma** after the value, otherwise it will give you an error!

Accessing the values

If we try to access `olympicRecords`, it’s going to return the whole object, the container of all properties and values.

```
console.log(olympicRecords);
//=> Object {athletics100Men: "Justin Gatlin", athleticsLongJumpMen: "Mike Powel"}
```

Wouldn't it be useful to access the values inside the object?

We have two different ways to access the values inside the object: the `dot notation` or the `brackets notation`.

```
olympicRecords.athletics100Men          // => "Usain Bolt"
olympicRecords["athleticsLongJumpMen"] // => "Mike Powel"
```

There is no difference between them, but using the **dot notation** is three characters shorter and much more used so we will keep using this way as preferred one 😌

Add properties to the Object

Let's add some new properties (key-value pairs) to our object.

We have two different ways to add new properties to the object: while we are declaring the object, and after we declare it.

Let's add the most decorated Olympian of all time, the Olympic Record in Swimming 200 meters men category Michael Phelps.

We can add a new property to the object when we declare it. Easy:

```
let olympicRecords = {
  athletics100Men: "Justin Gatlin",
  athleticsLongJumpMen: "Mike Powel",
  swimming200Men: "Michael Phelps"
}
```

As you can see, we can add many keys as we want inside an object, all we need to do is separate all the properties with a `comma`.

Adding properties with dot notation

We can also add properties to an object **after we define it**. The same way we declare a variable, we need to give it a name and a value. We do that by accessing a **new key** and assigning it a value:

```
let olympicRecords = {
  athletics100Men: "Justin Gatlin",
  athleticsLongJumpMen: "Mike Powel",
}

olympicRecords.swimming200Men = "Michael Phelps";
```

As you see, you can add a new key by referencing it directly with a dot. Nothing else! Awesome, huh? ;)

Adding properties with bracket notation

There is also another way to add new properties to the object. We can treat is as if it was an array. Let's add the Olympic Champion and nine-time world champion Katie Ledecky to the object:

```
let olympicRecords = {
  athletics100Men: "Justin Gatlin",
  athleticsLongJumpMen: "Mike Powel",
  swimming200Men: "Michael Phelps"
}

olympicRecords["swimming400Women"] = "Katie Ledecky";
```

So you can use the brackets to add new pairs of key-values to the object. You have to indicate the key between the brackets and assign the value.

Remember that if you use bracket notation, we need to **wrap the key with quotes** (`"`), unless it's a variable containing a string!

Does property exist in object - `in` operator

We can use the `in`operator to verify if a certain property exists in an object. It returns a `boolean` depending if the property exists or not.

Syntax

```
prop in objectName
```

Examples

```
let myCar = {
    make: 'Honda',
    model: 'Accord',
    year: 1998
};

'make' in myCar  // returns true
'model' in myCar // returns true
```

Update values

Hold on… Justin Gatlin?? What happened to the greatest sprinter of all time Usain Bolt? **It's outdated!!** Don't worry, we will update it.

We have two different ways to update values in an object. Do you know which ones? Exactly! The same ways we just learned:

```
olympicRecords.athletics100Men = "Usain Bolt";

// or

olympicRecords["athletics100Men"] = "Usain Bolt";
```

When you indicate the key between brackets, you have to put it between quotes.

Removing properties

Let's suppose added a new key, the Double Ollie world record:

```
olympicRecords.doubleOllie = "Chris Chann"
```

But after careful consideration, we realize that it's a fake. How can we remove this key?

In JavaScript we have the `delete` operator to remove keys from an object. You just have to specify which key you want to remove:

```
delete olympicRecords.doubleOllie;

// or

delete olympicRecords["doubleOllie"];
```

Now, if we take a look at the `olympicRecords` object, we will have just the real Olympic records!!

```
let olympicRecords = {
  athletics100Men: "Usain Bolt",
  athleticsLongJumpMen: "Mike Powel",
  swimming200Men: "Michael Phelps",
  swimming400Women: "Katie Ledecky"
}
```

List properties

To finish the lesson, we will introduce you to two Object methods that will help you to list all the properties and values of the object.

It is useful when you have a huge object and you are not sure which properties and values it has.

`keys`

Object.keys()

In one side, we have the Object.keys() method. It receives, as a parameter, the object you want to inspect. In our case it would look something like this:

```
Object.keys(olympicRecords);
// => ["athletics100Men", "athleticsLongJumpMen", "swimming200Men", "swimming400Women"]
```

The function returns an array with all the properties keys of the object. Once you have the array, you can iterate over the elements and do whatever you please.

for … in loop

This is a special case of *for* loop which allows us to walk through the properties of any object in JavaScript:

```
//    placeholder, can be any word
//       |
for(let key in olympicRecords){
  console.log(key);
}
```

```
// console:
// athletics100Men
// athleticsLongJumpMen
// swimming200Men
// swimming400Women
```

`values`

Object.values()

In the other side, we have the Object.values() method. You also need to pass the object you want to inspect as a parameter. It will look like this:

```
Object.values(olympicRecords);
// => ["Usain Bolt", "Mike Powel", "Michael Phelps", "Katie Ledecky"]
```

The function returns an array with all the values of the object. Once you have the array, you can iterate over the elements and do whatever you please.

Can we use `const` to declare object?

The answer is - **absolutely yes**. Although it might seem that variables declared with `const` can't be changed ever, and that is true, we have to understand a bit deeper meaning of this *"can't be changed"*.

In the case of declaring an object using the `const` keyword, this means that new properties and values can be added BUT the value of the object itself **is fixed to the same reference (address) in the memory** and the object (or any variable declared with *const*) **can't be reassigned**.

Let's see what this means:

```
const student = {
  firstName: "Ana"
}
student.age = 25;
console.log(student); // <== { firstName: 'Ana', age: 25 }
```

So we see that we were able to add a new property to this object. When will the error happen then? The answer is - when we try to reassign the object:

```
const student = {
  firstName: "Ana"
}
student = {
  firstName: "Ale"
}
console.log(student); // <== error: "student" is read-only
```

📝 **Time to practice**

Okay, it's time for practice! We will work with objects from simple to complicated. Let's solve it, one iteration at a time.

We have received a request from the Public Library to send them an example of what would be a good way to organize their user registry:

1. They want to track user's information (user id and full name), and which books each user has.
2. For each book, they want to have some information about the book: title, author, category and ISBN.

Let's split the problem into small parts. It's the best way to confront a big problem. We will work over on repl.it to do this exercise.

Iteration 1

First of all, let's create a `user` object. You should start with something like this:

```
const user = { name: "Nick", id: 7 };
```

Let's create the user with your own information (your name, and your favorite number as an id). So we will have to add an id and the name to the object, and set them up with the right values.

Iteration 2

In the same way, let's create some `book` objects. Let's create two different books (your favorite books) with the following data: Title, Author, ISBN and Category.

```
const book1 = { title: "The Catcher in the Rye", author: "J.D Salinger", isbn:
"0316769487", category: "Classic Literature" };
const book2 = { title: "To Kill a Mockingibrd", author: "Harper Lee", isbn: "0446310786",
category: "Classic Literature" };
```

If you want to figure out the ISBN of your favorite books, visit the ISBN search page.

Iteration 3

The next step is to relate the books with the user. As we can deduce, a user can have several books at the same time. Which data type do we know that allows us to specify several data in the same field?

Exactly, an **array**. Let's add an array inside the user that represents the books. The array must contain the books that we've created.

Use the dot notation to add the new key in the user object.

Iteration 4

Now, we have to create a library object and add the only user that we have right now. Again, we will have several users in the Library, so we could use an array to store all of them.

```
const library = [];
```

So, we have to do three different things here:

- Add the array of books to the user object
- Create the library array.
- Add the user object into the library array.

Iteration 5

Let's pick up a new book from the library. That means we will have to add a new book inside the `user.books` array. But now, the user is inside the `library` object. So you have to access the `library`, then the `user` and, finally, the `books` array to add the new book.

```
const book3 = {};
```

Bonus iteration

Let's iterate! To finish this exercise, let's iterate over the users and the books. We have to get a list of the users and the books that have each user. For example, Ironhack books are the following:

```
Ironhack books:

- JavaScript - The Good Parts, Douglas Crockford
- JavaScript - The Definitive Guide, David Flanagan
- High Performance JavaScript, Nicholas C. Zakas
```

In this case we just have one user. Try to add another user with books, and list the books of both users.

Summary

In this unit, we learned a lot of new things.

- We learned some programming fundamentals like key-value pairs and the JSON format.
- We know what a JavaScript `Object` is, and why it's a good practice to use them.
- We know how we can create objects and interact with their properties and values.
- Finally, we've learned an `Object` method to list all the properties of an object, something that will be very useful in the future, during the bootcamp :)

Extra Resources

- `Object` - MDN Documentation.
- Access to Properties - MDN Documentation.
- `delete` operator - Delete a property from an `Object`.
- `Object.keys()` method - Lists all the properties of an `Object`