Data Types

In this lesson, we'll cover the most important data types available in JavaScript.

WE'LL COVER THE FOLLOWING

Data Types

`JavaScript` is a dynamic language, meaning that on the contrary to a static language, you **don't** have to define the type of your variables when you define them.

```
// is this a string or a number ?
var userID;

userID = 12; // now it's a number
console.log(typeof userID); // number
userID = 'user1' // now it's a string
console.log(typeof userID); // string
```

This may seem convenient at first, but it can be a cause of problems when working on bigger projects. At the end of this course, after you have mastered the basics of `JavaScript`, I'll introduce you to `TypeScript`, which adds strong typing to `JavaScript`.

There are 7 data types in `JavaScript` : 6 **primitives** and the `Object`.

Primitives

A **primitive** is simply data that is not an `Object` and doesn't have methods.

They are:

- `string`
- `number`
- `boolean`
- `null`
- `undefined`
- `symbol` (the latest addition)

Let's have a quick look at all of them, some of which you may already know if you have prior experience in programming.

`string` is used to represent text data, whether it's a name, an address or a chapter of a book.

```
let userName = "Alberto";
console.log(userName) // Alberto
```

`number` is used to represent numerical values. In `JavaScript` there is no specific type for Integers.

```
let age = 25;
```

`boolean` is used to represent a value that is either `true` or `false`.

```
let married = false;
```

`null` represents *absence* of value, while `undefined` represent an *undefined* value

value.

`symbol` represents a value that is unique and immutable. It was added in ES2015, making it the most recent addition to this list. We will have a better look at it in the symbols lesson.

Objects

While the previous 6 **primitives** that we discussed can hold only a single value, whether it’s a null value, true, false, etc., objects are used to store the collection of properties.

Let’s first look at a simple `Object`

```
const car = {
  wheels: 4,
  color: "red",
}
```

This is a simple `Object` that I use to store properties of my car.

Each property has a key, in the case of the first line it’s `wheels`, and a value, in this case *4*.

Key is of type `string` but the value can be of **any** type. They can also be functions, and in that case, we call them `methods`.

```
const car = {
  wheels: 4,
  color: "red",
  drive: function(){
    console.log("wroom wroom")
  }
}
console.log(Object.keys(car)[0]) // wheels
console.log(typeof Object.keys(car)[0]) // string
car.drive();
// wroom wroom
```

As you can see now, we can call the function `drive` on the object `car`.

Don’t worry, we will look at functions more in the next chapter.

Create an empty object

We don’t have to declare properties when we create an Object.

Here are two ways of creating an empty `Object`:

```
const car = new Object()
const car = {}
```

The more commonly used syntax is the second one, which is called `object literal`

Now that you have a new empty `car` object to add new properties to it, you can simply do this:

```
const car = {};
car.color = 'red';
console.log(car)
// {color: "red"}
```

As you can see, we use the *dot notation* to add a new property to the `car` object.

How about **accessing properties** on the Object? It’s very simple and we have two choices:

```
const car = {
  wheels: 4,
  color: "red",
}

console.log(car.wheels);
// 4
console.log(car['color']);
// 'red'
```

We have two different ways of doing the same thing? Why?

Well, they're not completely the same. In case of *multi-word* properties we cannot use the `dot notation`.

```
const car = {
  wheels: 4,
  color: "red",
  "goes fast": true
}
console.log(car.goes fast);
// syntax error
```

```
const car = {
  wheels: 4,
  color: "red",
  "goes fast": true
}
console.log(car['goes fast'])
// true
```

When you want to use *multi-word* properties, you need to remember to wrap their names in quotation marks, and that you're able to access them only with *bracket notation*.

Another use for the `bracket notation` is to use it to access properties of an object by its key.

Let's say that our application receives an input from a user, which is then saved into a variable that will be used to access our object.

The user is looking for cars and he/she has been asked to tell us the brand that he/she likes. That brand is a key that we will use to only display back the appropriate models.

For simplicity, in the example each brand will have only one model.

```
const cars = {
  ferrari: "california",
  porsche: "911",
  porsche: "911",
  bugatti: "veyron",
}

// user input
const key = "ferrari"
console.log(cars.key);
// undefined
console.log(cars['key']);
// undefined
console.log(cars[key]);
// california
```

As you can see, we need to use `bracket notation` to access the property of the object via its key, stored in our variable.

Be careful, no strings are around `key`, as it’s a variable name and not a string.

Copying objects

In contrast to primitives, objects are copied by reference, meaning that if we write:

```
let car = {
  color: 'red'
}
let secondCar = car;
```

Our `secondCar` will simply store a reference, an “address”, to the `car` and not the `Object` itself.

It’s easier to understand if you look at this:

```
let car = {
  color: 'red'
}
let secondCar = car;

car.wheels = 4
console.log(car);
// {color: 'red', wheels: 4}
console.log(secondCar);
// {color: 'red', wheels: 4}
```

As you can see, the `secondCar` simply stored a reference to `car`, therefore when we modified `car`, `secondCar` also changed.

If we compare the two objects, we can see something interesting:

```
console.log(car == secondCar);
// true
console.log(car === secondCar);
// true
```

Whether we use **equality** (`==`) or **strict equality** (`===`),we get `true` meaning that the two objects are the same.

Only a comparison between the same `Object` will return `true`.

Look at this comparison between empty objects and objects with the same properties.

```
const emptyObj1 = {};
const emptyObj2 = {};

console.log(emptyObj1 == emptyObj2);
// false
console.log(emptyObj1 === emptyObj2);
// false

const obj1 = {a: 1};
const obj2 = {a: 1};

console.log(obj1 == obj2);
// false
console.log(obj1 === obj2);
// false
```

As you can see, only a comparison between the **same object** returns `true`.

A quick way of making a clone of an `Object` in JavaScript is to use `Object.assign`.

```
const car = {
  color:'red'

}

const secondCar = Object.assign({}, car)
car.wheels = 4;
console.log(car);
// {color: 'red', wheels: 4}
console.log(secondCar);
// {color: 'red'}
```

Updating `car` did not affect `secondCar`. `Object.assign` takes a target object as the first argument, and a source as the second one. In our example, we used an empty `Object` as our target and our `car` as the source.

If you are ready for a more in-depth look at copying Objects in JavaScript, I suggest this article on Scotch.io.

Arrays

As we have seen, objects store data in a **key value** pair. Now we'll have a look at what an `Array` is.

An `Array` is an `Object` that stores values in order. In the example above we used an `Object` to store our car because it had specific properties that we wanted to be able to access easily via a key. If we had just wanted to store a list of items, then there's s no need to create an `Object`. Instead, we can use an `Array`.

For example:

```
const fruitBasket = ['apple','banana','orange']
```

We access values of an array via their **index**. **Remember that arrays start at position 0**.

```
const fruitBasket = ['apple','banana','orange']
console.log(fruitBasket[0])
// apple
```

```
console.log(fruitBasket[1])
// banana


console.log(fruitBasket[2])
// orange
```

There are many methods that we can call on an `Array`. Let’s have a look at some of the most useful.

```
const fruitBasket = ['apple','banana','orange']
// get the length of the Array
console.log(fruitBasket.length);
// 3

// add a new value at the end of the array
fruitBasket.push('pear')
console.log(fruitBasket);
// ["apple", "banana", "orange", "pear"]

// add a new value at the beginning of the array
fruitBasket.unshift('melon')
console.log(fruitBasket);
// ["melon", "apple", "banana", "orange", "pear"]

// remove a value from the end of the array
fruitBasket.pop()
console.log(fruitBasket);
// ["melon", "apple", "banana", "orange"]

// remove a value from the beginning of the array
fruitBasket.shift()
console.log(fruitBasket);
// ["apple", "banana", "orange"]
```

We can easily add and remove elements from the beginning or the end of an `Array` with these methods.

You can find a longer list of methods on MDN.

Determining types using `typeof`

We can use `typeof` to determine the value of our variables. For example:

```
const str = "hello"
console.log(typeof(str));
// string

const num = 12;
console.log(typeof(num));
// number

const arr = [1,2,3];
console.log(typeof(arr))
// object

const obj = {prop: 'value'};
console.log(typeof(obj));
// object
```

Remember that ‘Array’ is not a type, Arrays are Objects!

Everything seems pretty straight forward so far, but what if we try something like this:

```
typeof(null)
```

We know that `null` is a **primitive**, so should we expect to see `null` as the result?

```
console.log(typeof(null));
// object
```

Long story short, it’s a bug from the first implementation of `JavaScript`. If you want to know more about it, this article offers a great explanation.

In the next lesson, we’ll learn about functions in JavaScript.