The Object Type

Up to this point you've interacted with a lot of different data types in JavaScript. Now it's time to introduce one of the most diverse and widely used data types of all: `Objects`.

An object is a data structure that stores other data, similar to how an array stores elements. An object differs in that each `value`stored in an object is associated with a `key`. `Keys`are almost always strings while `values`can be any data type: numbers, strings, functions, arrays, other objects, anything at all!

When you finish this reading, you should be able to:

1. Create objects using correct syntax with a variety of values.
2. Identify that an object is an unordered collection of values.
3. Key into an object to receive a single value using both Bracket and Dot notation
4. Use Bracket notation to set a variable as a key in a Object.
5. Implement a check to see if a key already exists within an Object.
6. Understand how object precedence fits in with dot notation for objects.

The object of my affections

To reiterate, an object is a data structure that stores other data. In other programming languages similar data structures to the Object type are referred to as 'dictionaries', 'maps', or 'associative arrays'. Objects are different from the previous data structures we've talked about (i.e. arrays) in two important ways:

1. Instead of accessing values within an object through an index with numbers, objects are indexed using `keys`.
    - This allows us to access values quickly and efficiently. We'll be talking a more more about this point later on in the course.
2. Order is **not**guaranteed within an Object. When you iterate through the values in an object, they may not be in the same order as when they were entered.

Objects are defined by using curly braces: `{}`. See below for an example:

```
> let car = {};
undefined

// here is our new empty object!
> car
{}
```

Fun Fact: Objects are known by the affectionate industry jargon: Plain Old JavaScript Objects (or POJO for short). Expect to see that short-hand often!

Setting keys and values

When learning about objects it can be helpful to think about real life objects. For instance think about a car. A real life car can have a color, a number of wheels, a number of seats, a weight, etc. So a real life car has a number of different properties that you wouldn't list in any particular order, though all those properties define the characteristics of that car.

Thinking of a car - let's create a `car`object to represent that collection of properties. We can create new `key`-`value`pairs using bracket notation `[]` and assignment `=`. Notice that the key inside the brackets is represented with a string:

```
// here "color" is the key!
> car["color"] = "Blue";
"Blue"
```

```
> car["seats"] = 2;
2

// accessing our object at the key of color
> car["color"]
"Blue"

> car["seats"]
2

> car
{color: "Blue", seats: 2}
```

When we enter `car["color"]`, we are using `"color"`as our `key`. You can think of `keys`and `values`in an object just like a lock and key in real life. The`"color"`key "unlocks" the corresponding value to give us our `car`'s color,`"Blue"`!

Keys without values

What happens if we try to access a key that we have not yet assigned within an object?

```
> car
{color: "Blue", seats: 2}

> car["weight"]
undefined
```

If we try to access a key that is not inside an object we get `undefined`. This falls right into place with our understanding of where `undefined`shows up in JavaScript. It's the common default value of a lot of things.

The `undefined`type is the default for unassigned variables, functions without a `return`, out-of-array elements, and non-existent object values.

Using this knowledge, we can check if a key exists in an object:

```
> car
{color: "Blue", seats: 2}

> car["color"]
"Blue"

> car["color"] === undefined;
false

> car["weight"] === undefined;
true
```

While this is a common pattern, in modern JS the preferred method to check if an object exists in a key is to use the `in`operator:

```
> car
{color: "Blue", seats: 2}

> "color" in car;
true

> "model" in car;
false
```

Using variables as keys

So we've talked about assigning string keys within Objects. Additionally, we know how to create variables that have strings as values. Sooo... you might be thinking: what happens if we assign a variable with a string value as a `key`within an `Object`? Glad you asked! Let's look at an example below for setting keys within `Objects`using `variables`.

Let's keep playing with the `car` we made previously:

```
> car
{color: "Blue", seats: 2}

> let newVariable = "color";
undefined

> newVariable
"color"

> car[newVariable]
"Blue"

> car["color"]
"Blue"
```

Aha! Of course we can use a variable as our key! A variable *always*evaluates to the value we assigned it. So `car[newVariable]`and `car["color"]`are equivalent! Why is this useful? We know that variables can change; so now the keys we use for objects can change!

Let's take a look at what happens when we change the variable above:

```
> car
{color: "Blue", seats: 2}

> newVariable
"color"

> newVariable = "weight";
undefined

> car[newVariable]
undefined
```

```
undefined

// car doesn't change because we didn't *assign* the new variable key in our obje
> car
{color: "Blue", seats: 2}
```

We can now use our newly assigned variable to set a *new key*in our object:

```
> car
{color: "Blue", seats: 2}

> newVariable
"weight"

// assigning a key value pair using a variable!
> car[newVariable] = 1000;
1000

> car
{color: "Blue", seats: 2, weight: 1000}
```

Using different notations

So far we've shown how to access and set values in objects using `object[key]`- also known as *Bracket Notation*. However, this is only one of two ways to access values within an object. The second way we can access values within an object is called *Dot Notation*. We can use `.`to assign and access our key-value pairs. The easiest to notice difference is when we use *dot notation*, we don't need to use string quotes as the key:

```
> let dog = {};
undefined
```

```
> dog.bark = "Bowowowo";
"Bowowowowo"

> dog.bark
"Bowowowo"

> dog
{ bark: "Bowowowowo" }
```

Bracket notation vs Dot notation

Now that we know two ways to access values of an object, you are probably asking yourself: which one should you use? Here is a quick list of pros for each.

Dot notation:

- easier to read
- easier to write because we don't have to deal with using quotation marks
- **cannot**use variables as keys
- keys can't contain numbers as their first character (`object.1key`doesn't work!)

Bracket notation Pros:

- you can use variables (assigned to string values) as keys!
- It is okay to use variables and Strings that start with numbers as keys (use `object['1key']`does work, while `object.1key`does not)

There are tradeoffs and advantages for either notation, so practice using both! You will learn quickly that there are **a ton** of different ways to write the same thing in JavaScript. Having both of these options available to you will allow you to use different tools to solve different problems.

One of the most fun parts of being a programmer is the ability to come up with different solutions to the same problem. So you should have both types of notation in your tool-belt to be a versatile programmer!

Let's look at the difference:

```
let myDog = {};
myDog.name = "Fido";

// let's use a variable as our key and some bracket notation:
let myKey = "name";
console.log(myDog); // prints `{name: "Fido"}`
console.log(myDog[myKey]); // prints `Fido`

// what if we try to use the variable in dot notation:
// the below is interpreted as myDog['myKey']
console.log(myDog.myKey); // prints: undefined
```

When we use dot notation to write `myDog.myKey`, `myKey`will **not be interpreted by JavaScript as a variable**. The text we write after the `.` will be used as the **literal**key. Remember that if we try to use a key that does not exist in an object, we get back the default value of `undefined`.

```
// continued from above

console.log(myDog.myKey); // prints `undefined`
myDog.myKey = "???";
console.log(myDog); // prints `{name: "Fido", myKey: "???"}`
console.log(myDog.myKey); // prints `???`
// mind === "blown"
```

Putting it all together

We can also create an entire object in a single statement:

```
let myDog = {
  name: "Fido",
  type: "Doge",
  age: 2,
  favoriteToys: ["bone", "ball"]
};

console.log(myDog.age); // prints 2
console.log(myDog["favoriteToys"]); // prints ["bone", "ball"]
```

Operator precedence revisited

Just like with math and logical operators, the concepts of operator precedencealso pertain to objects. Associativity determines the order of operation, along with precedence. There are two types of associativity: right-associativity and left-associativity.

Right-associativityis when code is evaluated right-to-left. Let's take a closer look at what is happening in the line of code below:

```
a = b = 1;
```

1. Variable `b`is assigned as `1`.
2. Variable `a`is assigned as `b = 1`.
3. `b = 1`returns the value `1`, so variable `a`is now assigned as `1`.

The assignment of variables takes lowest precedence, which is why we evaluate the return value of `b = 1`before completing the assignment of variable `a`.

The example below is **left-associativity**is when code is evaluated left-to-right. It evaluates the `document.getElementById`method before accessing `value`.

```
let id = "header";
let element = document.getElementById(id).value;
```

1. We resolve the `document`variable to be the *document object*.
2. We use dot notation to retrieve the `getElementById`function. (The function is a property of the *document object*).
3. We attempt to call it, but before the call can proceed we must first evaluate the function's arguments.
4. We resolve the `id`variable to be the string `"header"`.
5. The `getELementById`function returns an *HTMLElement object*and then uses dot notation to access `value`.
6. Finally we do assignment which is the LOWEST precedence (that's why assignment happens last). Its associativity is right to left, so we take the value on the right and assign it to the left.

Now let's dive into the example below. Resolving the variables to their values happens before the operators.

```
add(number1, number2) + number3;
```

1. `number3`is resolved to its value.
2. The function is invoked, but its variables need to be resolved.
3. `number1`and `number2`are resolved to their values.
4. The function is invoked so `number1`, `number2`, and `number3`are finally added together!

What you learned

In this reading we covered:

- Objects are un-ordered data structures consisting of key and value pairs.
- Object `keys`are strings, but their `values`can be anything (arrays, numbers, strings, functions, etc.)

- Setting key and value pairs using both Bracket and Dot notation
    - Using Bracket notation to set variables as `keys`in Objects
- The default value when accessing a key **not**in an object is `undefined`
    - How to check if a key is already within an object using the `object[key] === undefined`pattern