JavaScript Objects and Arrays

Data Types

In programming there is a concept of a data type. A data type defines the values that a variable can hold, and the operations that can be performed on it. JavaScript is very flexible in terms of data types, people call it a "weakly" or "loosely" typed language. What this means is that even though a variable may start as one type, it can be changed to another type at a later time. In "strongly" typed languages once a variable is set to a specific type it cannot be changed. "Strongly" typed languages provide more type safety, meaning you can assume that a variable will always be a certain type in a "strongly" typed language. However, "weakly" typed languages allow for a lot of flexibility.

Data Sets

Consider you have a collection of numbers: 2, 4, 6, 8, and 10. You need a way to represent this collection in memory (where a computer stores information). You could potentially store the numbers in a string `"2 4 6 8 10"`, but that would make it really difficult to access the numbers and perform operations with them. If you wanted to sum the numbers in the collection, you would end up having to split the string with a space and then extract each number and add them.

JavaScript provides a data type that makes it very easy to interact with collections known as an `Array`. Arrays are written as a list of values between square brackets. The representation of the collection above would look like this in JavaScript:

```
let list = [2, 4, 6, 7, 10];
```

It is important to note that `list` is just the name we chose to store the array into, you could use any value for that variable. Now that we have the list, JavaScript provides a mechanism for accessing the items in the list. You use square brackets to access a value in an array by its `index`. The term `index` is used to represent the numbered position of a value in an array. Consider the following code:

```
let list = [2, 4, 6, 7, 10];

// Logs 6 to the console
console.log(list[2]);
```

The code above will define our array of numbers, store it in a variable called "list" and then log the value at index "2" to the console. The comment mentions that it would log "6", and you may be thinking that the value at position "2" is 4. The reason "6" is logged is because arrays in JavaScript and many (but not all) other languages is "zero-indexed". This means that the first item in an array can be found at index 0. So the following code will log "2" to the console:

```
let list = [2, 4, 6, 7, 10];

// Logs 2 to the console
console.log(list[0]);
```

You can think of the `index` as the amount of items to skip when starting at the first item in an array. It is also possible to put some logic in between the square brackets when accessing array values. Consider the following code:

```
let list = [2, 4, 6, 7, 10];
let index = 2;
let a = 5;
let b = -3;

// Logs 6 to the console
console.log(list[2]);

// Logs 6 to the console
console.log(list[1 + 1]);

// Logs 6 to the console
console.log(list[index]);

// Logs 6 to the console
console.log(list[a + b]);
```

All of the `console.log` statements above will result in the same thing being logged to the console.

Properties

Up to this point you have probably been using a couple properties without actually knowing what they are and how they exist. For example, if you have a string and want to log it's length you could write code like the following:

```
let greeting = 'Hello World';

// Logs 11 to the console
console.log(greeting.length);
```

The `.length` on the end of `greeting` is calling to access the `length` property that exists on all strings. Almost every single JavaScript data type has properties. The exceptions to that rules are `null` and `undefined`. Trying to access properties off of those values will result in errors.

In JavaScript you can access properties of an object in two different ways: with a dot and with square brackets. The following code will accomplish the same thing:

```
let greeting = 'Hello World';
let property = 'length';

// Logs 11 to the console
console.log(greeting.length);

// Logs 11 to the console
console.log(greeting['length']);

// Logs 11 to the console
console.log(greeting[property]);
```

It is highly recommended that you use dot notation when you know the property you are looking for. The bracket notation is a little more verbose, and should be reserved for when you need to access a dynamic property using a variable like in the `greeting[property]` example above.

Methods

Strings and Arrays both contain a `length` property. In addition to that they both contain properties that are functions. These are called `methods`.

```
let greeting = 'Hello World';

// Logs "function" to the console
console.log(typeof greeting.toUpperCase);

// Logs "HELLO WORLD" to the console
console.log(greeting.toUpperCase());
```

All strings contain the same methods, and you can read more about them on MDN. Now take a closer look at the code above. `greeting.toUpperCase()` calls the `toUpperCase` method on all Strings, and without any arguments it is able to return `HELLO WORLD`. Properties that contain functions are called `methods` for this purpose, they have access to the value to which they belong. This will be explained in depth further on in this course. For now, know that when someone says "call the [y] method on x" they are in fact saying that "x" has a property on it called "y" and it is a function.

Arrays in JavaScript have some very useful methods for manipulating data. The following code shows you just a couple of methods on arrays:

```
let list = [2, 4, 6, 7, 10];

console.log(list);
// -> [2, 4, 6, 7, 10]

list.push(3);

console.log(list);
// -> [2, 4, 6, 7, 10, 3];

console.log(list.pop());
// -> 3

console.log(list);
// -> [2, 4, 6, 7, 10]

list.unshift(3);

console.log(list);
// -> [3, 2, 4, 6, 7, 10]

console.log(list.shift());
// -> 3

console.log(list);
// -> [2, 4, 6, 7, 10]

console.log(list.concat([3, 4]));
// -> [2, 4, 6, 7, 10, 3, 4]

console.log(list);
// -> [2, 4, 6, 7, 10]

list = list.concat([3, 4]);

console.log(list);
// -> [2, 4, 6, 7, 10, 3, 4]
```

- The `push` method adds values to the end of an array
- The `pop` method removes the last value in an array and returns it
- The `unshift` method adds values to the beginning of an array

- The `shift` method removes the first value in an array and returns it
- The `concat` method combines two arrays together and returns a new array (not changing either array)

The names of the methods may be confusing/seem not intuitive. However, they are based on programming operations for stacks and queues. A stack is a common programming data structure that allows you to push and pop values in a Last In First Out (LIFO) pattern. A queue is a structure that uses a First In First Out (FIFO) pattern.

For more information on Array methods see the documentation on MDN.

Objects

Now you have a decent amount of experience with Strings and Arrays in JavaScript. However, what if you need to represent a set of arbitrary data. Say for example you have a blog post. The posts consists of a title, category, body, and a list of tags. It would be awfully difficult to represent a post using any of the data types you currently know. This is where `objects` come in. Objects are arbitary collections of properties. Representing the post as an object is fairly simple:

```
let post = {
    title: 'Welcome To My Blog',
    category: 'Announcements',
    body: 'Hello and welcome to my blog. I will be posting about my adventures as a
developer',
    tags: ['JS', 'CSS', 'HTML']
};

console.log(post.title);
// -> 'Welcome To My Blog'

console.log(post.tags[1]);
// -> 'CSS'
```

The above code organizes the post into a logical collection of properties. Note that it is possible to nest objects as properties of other objects, etc. In the above vode `tags` is an Array as a property on `post`. The notation used above is called "object literal" notation. You will see further in this course that there is more than one way to create an object in JavaScript. Object literal notation starts with braces:

```
let myObject = null;
```

Inside the braces there is a collection of properties separated by commas. Each property has a name followed by a colon and a value.

```
let myObject = {
    propA: 'value a',
    propB: 'value b'
    'prop c': 'value c'
};
```

Notice above `'prop c'` is a property on `myObject`, but it is not a valid binding name/number so it must be surrounded by quotes. It is common to write object properties on new lines, and indent them for better readability.

Up to this point you have understood braces `null` to mean the start of a block of statements. Now we are introducing a new functionality in that they describe objects. While it may seem a little weird right now, in practice it's fairly easy to deal with both functionalities.

You don't always have to define all the properties of an object up front, sometimes it can be useful to assign them later. You do this with an `=`, similar to how you declare and assign a variable:

```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};

console.log(person.email);
// -> undefined

person.email = 'john.doe@gmail.com';
console.log(person.email);
// -> 'john.doe@gmail.com'
```

Sometimes it can be useful to get all the properties that exist on an object, you can do this with `Object.keys`:

```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};

console.log(Object.keys(person));
// -> ['firstName', 'lastName']
```

There is also an `Object.assign` function that copies all properties from one object into another:

```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};

console.log(Object.assign(person, {
    firstName: 'Jane',
    email: 'jane.doe@gmail.com'
}));
// -> { firstName: 'Jane', lastName: 'Doe', email: 'jane.doe@gmail.com' }
```

This isn't the only way to change properties on an object, you can update an object's properties with the `=` operator:

```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};

person.firstName = 'Jane';

console.log(person);
// -> { firstName: 'Jane', lastName: 'Doe' }
```

Now that you know what Objects are, it is not a stretch to understand that Arrays are just a special kind of object used for storing sequences of things. In fact, Strings are also special types of objects used to store sequences of characters. So in a way Strings are like Arrays.

Object Mutability

You have observed up to this point that values in an object can be modified. Prior to now you have been dealing mostly with objects that cannot be modified, such as numbers and strings. Those are called "immutable" objects. It is not possible to change 1 to 2. Yes you can do `let x = 1; x = 2`, but that doesn't change 1 to 2, it simply changes `x` to

2. Objects are different. You can change the properties on an object, causing it to have different content at different times. Consider the following code:

```
let a = {
    value: 10
};

let b = a;
let c = {
    value: 10
};

console.log(a == b);
// -> true
console.log(a === b);
// -> true
console.log(a == c);
// -> false
console.log(a === c);
// -> false

a.value = 15;

console.log(a.value);
// -> 15
console.log(b.value);
// -> 15
console.log(c.value);
// -> 10

a = {
    value: 10
};

console.log(a.value);
// -> 10
console.log(b.value);
// -> 15
console.log(c.value);
// -> 10
```

In the above example `a` and `b` are the same object, so changing `a` results in a change to `b` as well. However setting `a` to a new object disconnects it from the object that is assigned to `b`.