Javascript Basics

Special Variables

Last session we covered how to declare and compare basic 5 of the 6 (plus 1) types of variables in Javascript. In this session, we'll cover Objects, Arrays, and Functions.

As with primitive variables, if you are already confortable with these topics, feel free to skip to the next session.

In Javascript, Object is a type of value that has properties and a type. Think of a property as a variable that belongs to an object. In past sessions, we interacted with the `window` object to access its `document` and `alert` properties.

Basic Objects, Arrays, and Functions are all considered to be a type of Object. This can be shown by using the `instanceof` operator (don't worry if this looks a bit foreign):

```
var myObject = {} // defines an empty Object
var myArray = [] // defines an empty Array
var myFunction = function () {} // defines a function

console.log(myObject instanceof Object)
console.log(myArray instanceof Object)
console.log(myFunction instanceof Object)

console.log(Array instanceof Object)
console.log(Function instanceof Object)
```

An Object's type defines what kind of Object it is, and each object has a heirarchy of types it derives from. All objects, be they basic Objects, Arrays, or Functions, are ultimately decendants of the "Object" type, which is why the above alerts all return true.

The classic Object-Oriented Programming, or OOP, example is to consider triangles and rectangles "objects" that derive from the shape "object". This concept more or less applies to Javascript as well.

I won't go into OOP practices too much in this session, but I did want to say a few things for those that come from an OOP background. At its heart, Javascript is not really a OOP language in the same way Java or C++ is. Technically, Javascript is a Prototypal Language, not a Classical one. This means that Javascript Objects inherit from other Objects, as opposed to Objects being derived from Classes that inherit from other Classes. Even ES6 Classes are just "syntatical sugar". This means is that you do not need to define a class to define an object, and Javascript doesn't have an official concept of "class".

But we're not really going to go into Object prototypes and inheritance this session. In my professional career, I seldomly had to worry about this, and I doubt you will for awhile, especially if you embrace Javascript as a language apart of Java or C++. I just brought it up to explain why Functions and Arrays are all considered Objects.

Objects

So anyway, let's start with basic objects. There are several ways to define a basic Object in Javascript. A couple of common ways you might come across are:

```
// Literal constructor:
var myObject1 = {}
myObject1.someProperty = 'value'

// Object constructor:
var myObject2 = new Object()
myObject2.someProperty = 'value'
```

Both variable declarations are basically doing the same thing. The latter introduces a new operator: `new`. The `new` operator generates an object that uses the object type or specified function as a prototype. In this case, we're making a new object that uses the base object as a type. I'll cover the new operator more when we get to functions.

The `new Object()` syntax might look familiar to those of you that know different programming languages, but it is not particularly common in Javascript. I'd encourage you to use the literal constructor over the Object constructor approach.

On the line under each declaration, an object property is being defined. The property can be set and accessed the same way:

```
// Literal constructor:
var myObject1 = {}
myObject1.someProperty = 'value'
console.log(myObject1.someProperty)
```

Remember: `window` is an object. You can access and set properties on any mutable object (which is most objects), including the window object.

```
window.someProperty = 'value' // Yep, this is allowed!
console.log(window.someProperty)
```

If you recall, window is a special case in the browser in that it's optional, so you could write the above code like this:

```
someProperty = 'value' // Yep, this is allowed!
alert(someProperty)
```

and it would mean the same thing.

Setting a property on the window object is considered setting a "global variable", and should be done sparingly. We'll get into that more when we cover scope.

```
var myObject1 = {}
myObject1.someProperty = 'value'
console.log(myObject1.someProperty)
```

Setting properties on an object after defining a variable as an object is one option, but typically you will want to define an object with all of its properties in one call. Here is how you do that:

```
var myObject1 = {
  someProperty: 'value'
}
console.log(myObject1.someProperty)
```

Just add the property name between the curly brackets, add a colon, and specify the object property's value.

To define multiple properties, separate each property definition with a comma, like this:

```
var myObject1 = {
  someProperty: 'value',
  anotherProperty: 24
}
console.log(myObject1.someProperty)
console.log(myObject1.anotherProperty)
```

The line breaks in the definition are optional by the way. You could define the object like this:

```
var myObject1 = { someProperty: 'value', anotherProperty: 24 }
```

It's a stylistic preference.

You may have noticed the new property value wasn't a string. Object properties can be defined to anything a variable can, even other objects:

```
var myObject1 = {
  someProperty: 'value',
  anotherProperty: 24,
  innerObject: {
    innerProperty: false
  }
}
console.log(myObject1.someProperty)
console.log(myObject1.anotherProperty)
console.log(myObject1.innerObject.innerProperty)
```

The inner object is referenced the same way: we're just adding an extra dot operator.

We're alerting the "innerProperty" on the "innerObject" on the "myObject1".

One other note: properties on Objects are much more lenient on permitted names than raw variables. You can have properties that start with numbers, or if you use quotes around the property name, have properties with spaces in the name. To reference properties like this, you'll need to reference it with square brackets. For example:

```
var myObject1 = {
  someProperty: 'value',
  '20anotherProperty': 24, // starts with a number, need to add quotes
  innerObject: {
    'inner property': false // has a space, adding quotes
  }
}
console.log(myObject1.someProperty)
console.log(myObject1['20anotherProperty'])
console.log(myObject1.innerObject['inner property'])
```

But please try to stick to traditional naming conventions.

Arrays

In Javascript, an Array is a type of object that is used to store an ordered list of properties. Properties in an array are referred to as "elements". This is the most common way to define a Javascript Array:

```
var myArray = ['a', 2, 'd']
```

To access an element of the array, use square brackets and an index, or the integeral point in an array starting at 0, of the element:

```
var myArray = ['a', 2, 'd']
console.log(myArray[0]) // alerts 'a'
console.log(myArray[1]) // alerts 2
console.log(myArray[2]) // alerts 'd'
```

Array elements can be of any type, and the length of the array does not need to be specified when it is first defined. The array and all elements in it are mutable.

By the way, Object properties can be referenced in a similar manner:

```
var myObject = { prop: 'val' }
console.log(myObject['prop']) // alerts 'val'
```

But the reverse is not true: You cannot access elements in array with the dot operator. You need to use square brackets. This actually uniquely an array issue, but occurs because Javascript does not allow accessing property names via the dot operator if the property starts with a number, as we mentioned earlier.

There's also an Array constructor object similar to the "new Object()" syntax we described earlier:

```
myArray = new Array('a', 2, 'd') // identical to ['a', 2, 'd']
myArray = new Array(5) // Creates an array with 5 undefined elements
```

You'll rarely use this syntax in your professional career, but the constructor has its uses, such as casting array-like node lists into Javascript arrays. That's out of scope for this session though.

Arrays have a handful of properties and a good number of functions for searching, mapping, and filtering, and we'll get into those more later on. Let's start with the length property:

```
console.log(myArray.length)
```

The length property on an array returns the number of elements in the array, regardless of whether they're defined or not. Do not ever modify this property.

The length property is often used for for loops, which we'll cover later on, and for getting the last element of the array:

```
console.log(myArray[myArray.length - 1])
```

Why -1? Because the length is 3, but because the indexes start at 0, the last index in the array is 2. The length is always one greater than the last index of the array.

There's also the `sort` function, which by default reorders the array based off the string representation of each element's UTF-16 code unit value:

```
var stringArray = ['apple', 'zebra', 'blueberry', 'aardvark']
stringArray.sort()
console.log(stringArray)
```

The default sort logic can be overridden, and most of the time you'll want to override that logic. We'll get into how to do that when we go over functions.

Functions

A Function is a type of object that executes an inner script. It allows passing in arguments and returns a result (or undefined if no return is specified).

Common Javascript functions are almost always created with the `function` operator. Here's an example of a global function:

```
function showSum (a, b) {
  console.log(a + b)
}
```

The word after the function operator, "showSum", is the name of the function. It's good convention to always have a function name start with or be a verb, like apply, fetch, or query. A good rule of thumb is to mentally split the words in the function name and see if it makes sense. For example, `showSum`'s functionality is to "Show the Sum". If I named it something like `numberCombine`, it would read "Numbers the Combine", which is a bit confusing.

Following the function name are the function arguments. These are a list of variable names, separated by commas, and surrounded by parentheses.

The funciton body is the lines of code that exists between the curly brackets. The variables specified in the argument list (the variables between the parentheses) are available here. The code in the function body is not executed unless the function is called.

To call a function, specify the function name with parentheses, just like when we called `window.alert` or `console.log`:

```
function showSum (a, b) {
  console.log(a + b)
}
showSum(1, 3)
```

When the function is called with the arguments "1" and "3", the function body replaces "a" with 1 and "b" with 3, and then outputs the result of combining them via the addition operator.

The `showSum` function is outputting the result of two variables combined with the addition operator into the console. It's not actually "returning" anything:

```
function showSum (a, b) {
  console.log(a + b)
}
var result = showSum(1, 3)
console.log('result is: ' + result)
```

To have a function return something, we use the `return` statement. I'll change the behavior and name of the function to return the sum of the two values:

```
function calculateSum (a, b) {
  return a + b
}
var result = calculateSum(1, 3)
console.log('result is: ' + result)
```

So now this function isn't just displaying the sum, it's calculating the sum and *returning* the result, which we later display.

Functions can return anything. They can return numbers, strings, objects, and even other functions. A function doesnt always need to return data of the same type either, but you should avoid having functions that return different types of data as it can lead to confusing issues.

The return statement specifies the end of the function. Once it's hit, the function does not execute anything else. So this additional console log in the function body will not be called:

```
function calculateSum (a, b) {
  return a + b
  console.log('Not hit') // return statement hit, so this won't be called
}
var result = calculateSum(1, 3)
console.log('result is: ' + result)
```

An alternative way to define functions is to specify the name as a variable prior to the function operator, like this:

```
var calculateSum = function (a, b) {
  return a + b
}
var result = calculateSum(1, 3)
console.log('result is: ' + result)
```

There's a very slight difference with this approach. When the Javascript engine in the Browser parses the script, it does an initial pass to find all functions defined with the function operator first. It later executes the code. That means you can call functions before they're defined in the code if you define the function a particular way.

So on a fresh page load, this will work:

```
var result = calculateSum(1, 3) // Works because calculateSum is registered before
the script executes
console.log('result is: ' + result)

function calculateSum (a, b) {
  return a + b
}
```

But this will not:

```
var result = calculateSum(1, 3) // Error because calculateSum not defined yet
console.log('result is: ' + result)

var calculateSum = function (a, b) {
  return a + b
}
```

You can define functions as properties on objects as well, and call them the same way you would call functions on the built-in window or console objects:

```
var calculateSum = function (a, b) {
  return a + b
}
var myObject = {
  calculateSum: calculateSum
}
var result = myObject.calculateSum(1, 3)
console.log('result is: ' + result)
```

Or more concisely:

```
var myObject = {
  calculateSum: function (a, b) {
    return a + b
  }
}
var result = myObject.calculateSum(1, 3)
console.log('result is: ' + result)
```

ES6 introduces even more concise ways to define functions and properties on objects, but we'll discuss that later on.

And just to show that a function is in fact a type of object, here's the calculateSum function above defined using the Function object constructor:

```
var calculateSum = new Function('a', 'b', 'return a + b')

var result = calculateSum(1, 3)
console.log('result is: ' + result)
```

You should NEVER define a function like this! It's slower (because the interpreter needs to evaluate a string as opposed to raw code), introduces potential security issues (because no failsafes are handled in the string body), cannot take advantage of debugger tools, and is a lot harder to read. But fundamentally, this is indeed how functions are constructed in Javascript.

And that's the basics for the Object type. In summary, the Object type is the foundation of Javascript Objects, Arrays, and Functions. Objects are data types that have properties with values attached. Arrays are a subset of Objects that are used for storing an ordered list of properties, called elements. Functions are a subset of Objects that are used to execute code inside of its function body.

Now that we know about functions in particular, we can move on to the next topic: scope.