CS 4220

- Current Trends in Web Development -

Application Design and Development with Node.js

Prof. Cydney Auman

AGENDA

01 Type Coercion & Operators

02 Strings & Template Literals

03 Mutation & Arrays

04 Conditionals & Loops

05 Functions

Review

We setup our Development Environment aka VS Code and added in some Extensions like Prettier and ESLint to enforce formatting and good code practices.

We discussed variable declaration and the differences between const and let.

We discussed JavaScripts Primitive types, and JavaScripts Non-Primitive types. And we also looked at a demo with how JavaScript uses dynamic type checking.

Type Coercion

Type coercion is the automatic or **implicit conversion** of values from one data type to another - such as coercing strings to numbers. Some of the time this **coercion** is unexpected in the type casting that happens and can cause error and bugs in your code. JavaScript **coercions** will always result in one of JavaScript's Primitive Types (string, number, or boolean). There is no coercion that would result in something more complex like an object or function.

JavaScript also allows for **explicit type coerce with methods** like:

- `String()` **or** `toString()`
- `Number()` **or** `parseInt()` **or** `parseFloat()`
- `Boolean()`

Operators

The Equals Operator

`(==)` and `(!=)`

The **==** version of equality is quite liberal. Values may be considered equal even if they are different types. In JavaScript the operator will **force type coercion** of one or both into a single type before performing a comparison.

The Strict Equals Operator

`(===)` and `(!==)`

If the values are of different types the answer is always false. If they are of the same type an intuitive equality test is applied. Strings must contain identical character sets and other primitives must share the same value. Null and Undefined will **never** === another type.

Operators Continued...

Comparison Operations

Greater than (`>`)

Great than or equal (`>=`)

Less than (`<`)

Less than or equal (`<=`)

Unary Operators (typeof)

Operators that take one value. Not all operators are symbols, so can be written as words.

`typeof` - this returns the type of variable as a string.

(`++`) - incrementing

(`--`) - decrementing

Logical Operators

Javascript supports three logical operators.

- And (`&&`)
- Or (`||`)
- Not (`!`)

Strings and Template Literals

Backtick-quoted strings, usually called **template literals** or **template strings**. By using this notation we can do a few more tricks with strings. Not only strings span multiple lines, they can also embed other values.

Using the syntax **${ }** inside in a template literal will computed the value and then that value is converted to a string, and included at the position.

The example below produces: "half of 100 is 50".

```
`half of 100 is ${100 / 2}`
```

Basic String Properties & Methods

JavaScript provides many methods on strings. A comprehensive list can be found at:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

`Property`

`length` property of a String that will return the length of the String

`Methods`

`charAt(`*`index`*`)` returns the character String at the index position specified

`includes(`*`value`*`)` determines whether the String includes the specified value and returns a boolean

`split(`*`value`*`)` divides a String into list of substrings , puts these into an array and return the array

`toLowerCase()` method returns the calling String value converted to lowercase

`toUpperCase()` method returns the calling String value converted to uppercase

Variables and Block Scope

A **block of code** is considered to be between the open and closed parenthesis `{ }`. These blocks of code can be as simple as conditionals or loops or even more complex like functions. Variables declared inside these blocks of code are known as **local** variables and only available within these blocks.

Anytime a variable is declared outside blocks of code, the variable is known as a **global** variable and is accessible by the entire program.

It is important to always try to scope variables into code blocks, this encapsulates code and prevents the variable from "leaking" outside the block. In most cases, declaring variables using let in the global scope is an anti-pattern and is considered bad practice. While declaring variables using const in the global scope is a standard practice in Node.js for requiring in modules.

**As we move forward in this class and begin to write more complex code, examples we will continue to discuss local and global scope variables.*

Conditionals

Not all programs are straight roads. For example - we might want to create a branching road, where our program takes the proper path based on the situation. This is called **conditional execution**.

Conditional execution is created with the **if** keyword in JavaScript. In the simple case, we want some code to be executed if, and only if, a certain condition holds. For example, we want to print the square of a number only if the type is actually a number.

```
if (typeof n === 'number') {
   const square = n * n;
   console.log(square);
}
```

Conditionals

You often won't write just code that executes when a single condition holds true. Often times your code needs to handle the other cases. You can use the **else if** and **else** keyword to create alternative execution paths.

```
if (typeof unknown === 'string') {
   console.log('value is a string');
} else if (typeof unknown === 'number') {
   console.log('value is a number');
} else {
   console.log('value is not a number or string');
}
```

Ternary Conditional

JavaScript supports a conditional known as the **Ternary**, because it operates on three values.

A ternary will begin with a conditional which is followed by a ?. After the ? is the truth statement. After the truth statement there is a : which is followed by the false statement.

If the conditional is evaluated to **true**, then the statement directly right of the ? executes. If the conditional is evaluated to **false**, then the statement **directly right of the :** executes.

```
// conditional ? true statement : false statement
const degreesCelsius = 2;
const H2O = degreesCelsius > 0 ? 'liquid' : 'solid';
```

Mutation

A value is **mutable** if it can be changed. Therefore, **mutation** is the act of changing the properties of some value.

All of **JavaScript's Primitive** are **immutable** - they cannot change their values or properties. Even if attempt to mutate a string - we will see that we can not actually change it's value or properties.

Arrays and Objects in JavaScript are considered mutable.   In fact, both are mutable even when declared using const. The keywords `const` and `let` only control whether a variable can be re-assigned and have nothing to do with mutation of properties.

Arrays

JavaScript provides a data type called an **array** specifically for storing sequences of values. An **array** is written as a list of values between square brackets, separated by commas. When declaring an **array** there is **no** need to use the `new` keyword.

```
const alpha = ['a', 'b', 'c', 'd'];
```

Elements in the array can be accessed by index. The first index of an array is zero. So the first element is retrieved with `alpha[0]` which return the value 'a'.

Javascript arrays also have a length property. This tells us how many elements it has inside the array. `alpha.length` will return 4.

**The contents of an array in JavaScript can be almost anything and in fact JavaScript arrays can have mixed data types.*

Basic Array Methods

JavaScript provides many methods on arrays. A comprehensive list can be found at:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#

`push(`*`value`***`)`**
adds one or more elements to the end of an array.

`pop()`
removes the last element from an array and returns that element.

`unshift(`*`value`***`)`**
adds one or more elements to the beginning of an array.

`shift()`
removes the first element from an array and returns that element.

Basic Array Methods

JavaScript provides many methods on arrays. A comprehensive list can be found at:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#

`reverse()`
reverses an array in place. The first array element becomes the last, and the last array element becomes the first

`concat(`*`array`*`)`
merges two or more arrays - this method does not change the existing arrays, but instead returns a new array

`join(`*`value`*`)`
returns a String concatenating the elements in the array by the value specified

`includes(`*`value`*`)`
determines whether the array includes the specified value and returns a boolean

`sort()`
sorts the elements of an array in place and returns the sorted array - the default sort order is ascending.

While Loop

The word **`while`** is followed by an expression in parentheses and then a statement of code to be executed. The loop keeps entering that statement as long as the expression produces a value of `true`.

```
let index = 0;
const alpha = ['a', 'b', 'c', 'd'];
while (index < alpha.length) {
    console.log(alpha[index]);
    index++;
}
```

For Loop

There are three parts to a `for` loop. The first part initializes the loop. The second part is the expression that checks whether the loop must continue. The final part updates the state of the loop after every iteration. In many cases, this is shorter and clearer than a while loop.

```
const alpha = ['a', 'b', 'c', 'd'];
for (let index = 0; index < alpha.length; index++) {
   console.log(alpha[index]);
}
```

Functions

Functions are an important part of JavaScript programming. It provides a way to structure our code and reduce repetition.

This form below is known as declaration notation or function declaration.

Functions can be declared with the keyword **`function`** followed by the function name. Functions have an optional set of arguments and a body, which contains the statements that are to be executed when the function is called. Finally, functions can have an optional return statement.

```
function add(a, b) {
   const sum = a + b;
   return sum;
}
add(2, 3);
```

Function Expression

A **function expression** is similar to the syntax as the function declaration. One difference between a function expression and a function declaration is the function name. Instead, the function is assigned to a variable and it is important to know that this variable is not the function's name. Since the name of the function is omitted and creates what is known as an **anonymous function**.

In general, **function expressions** are considered a best practice. This is because the behavior is intuitive. Meaning it follows the logical code flow - the function is defined and then called. Additionally, they keep the global scope light and maintains clean syntax.

```
const add = function (a, b) {
    const sum = a + b;
    return sum;
};
add(2, 3);
```

Arrow Functions

ES6 introduced the the **arrow function** which is also considered a expression. And for the most part are a compact alternative to a traditional function expressions seen in the slide above. There are some difference such as we cannot use the keyword `this` and thus not suitable for methods such as `call` and `apply` which rely on the `this` keyword.

To create an arrow function we simply remove the keyword function and place arrow between the argument and opening bracket. Arrow functions can be compacted even more when the function is simple.

```
const add = (a, b) => {
    const sum = a + b;
    return sum;
};
add(2, 3);
```

Default Values

In JavaScript, function parameters default to undefined when no value is passed. It's often useful to set a different default value. Default function parameters allow named parameters to be initialized with **default values** only if no value or undefined is passed.

```
const add = (a, b = 10) => {
   return a + b;
};
add(7, 2);  // 9
add(7);     // 17
```

Resources

ES6 Features

- https://github.com/lukehoban/es6features

Modern JavaScript Tutorial

- https://javascript.info/type-conversions
- https://javascript.info/array
- https://javascript.info/while-for
- https://javascript.info/function-basics

Mozilla Introduction to JavaScript

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections#array_object
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions

```
console.log(‘Week 02’);
console.log(‘Code Examples’);
```

Review and Prep

Review

- Review Slides
- Review and Run Code Examples

Quiz Due Sunday 02/05 at 11:59pm

- Open on Canvas at Thursday 02/02 at Noon PST
- Closes at 02/05 at 11:59pm PST
- Timed 12 minutes from start of Quiz. (1 attempt)
- Quiz will include all topics from 1/23 and 1/30 Lectures
- 10 Questions based on slides/code.
- YES! you can have slides, book, code opened

Preparation for Next Class

- Read Eloquent Javascript - Chapters 3 & 4