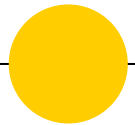
CS4220 Node.js & Angular.js

Cydney Auman
Albert Cervantes
CSULA



Introduction to Javascript

History of Javascript

First released in 1995 as part of Netscape Navigator. Netscape referred to their language as LiveScript. It was subsequently renamed to JavaScript (due to the popularity of Java). However, JavaScript is in no way related to Java.

In 1996 Netscape submitted JavaScript to ECMA International for standardization. This resulted in a new language standard, called ECMAScript (ES).

All major JavaScript implementations have been implementations of the ECMAScript standard, but the term JavaScript name stuck.

In the everyday world ECMAScript is usually used to refer to the standard while JavaScript is used when talking about the language in practice.

Everyone knows JavaScript as the language of the browser. An interesting Netscape introduced an implementation of the language for server-side scripting with Netscape Enterprise Server.

ES5 and ES6

- ES5 was released in 2009. That is still what a lot of developers are using today and what people know as modern JavaScript.
- ES6 was released in 2015 and is now becoming the new standard in development.
- A transpiler is a tool that reads code written in one language and then produce the equivalent code in another language. In our case ES6 to ES5
- Compatibility Table - https://kangax.github.io/compat-table/es6/

Babel Transpiler Example

You put JavaScript in

JavaScript Try Copy

```
[1,2,3].map(n => n + 1);
```

And get JavaScript out

JavaScript Try Copy

```
[1,2,3].map(function(n) {
  return n + 1;
});
```

Goal of ES6

- Providing a better language for writing:
  - complex applications
  - libraries shared by across applications
  - generators. (allow you to define an iterative algorithm by writing a single function which can maintain its own state)

Installation and Setup

- Install Node.js (v7.4.0 or higher) - nodejs.org

- After install
  - in your terminal run the command: `node -v`
  - the output should be `v7.4.x`

- Install your prefered IDE
  - Sublime Text - sublimetext.com/3

Core vs Client Side vs Server Side

Core JavaScript contains a core set of objects, such as Array, Date, and Math, and a core set of language elements such as operators, control structures, and statements. Core JavaScript can be extended for a variety of purposes by supplementing it with additional objects.

Client-side JavaScript extends the core language by supplying objects to control a browser and its Document Object Model (DOM). For example, client-side extensions allow an application to place elements on an HTML form and respond to user events such as mouse clicks, form input, and page navigation.

Server-side JavaScript extends the core language by supplying objects relevant to running JavaScript on a server. For example, server-side extensions allow an application to communicate with a variety of databases, provide continuity of information to and from the application, or perform file manipulations on a server.

Declaration

Variables in standard JavaScript have no type attached, and any value can be stored in any variable. In ES5 variables were all declared using the keyword **var**.

ES6 introduced **const** and **let.**

Using **const** makes your variables a constant value. Variables defined using the keyword **const** will never be changeable.

Using **let**, is more similar to **var** in the sense that you can change the value assigned. However, where **let** and **var** differ is in relation to how they scope themselves.

Javascript Primitive Types

- **Number** - JavaScript does not define different types of numbers, like integers, short, long, floating-point etc. *They are always 64-bit Floating point.*
- **String** - In JavaScript strings can be created using single or double quotes.
- **Boolean** - true and false literals.
- **Undefined** - The value of "undefined" is assigned to all uninitialized variables, and is also returned when checking for object properties that do not exist.
- **Null** - Unlike undefined, null is often set to indicate that something has been declared *BUT* has been defined to be empty.

Automatic Type Conversion

When an operator is applied to the “wrong” type of value, JavaScript will quietly convert that value to the type it wants, using a set of rules that often aren’t what you want or expect. This is called ***type coercion.***

JavaScript goes out of its way to accept almost any program you give it, even programs that do odd things.

Comparison Operators

The Equals Operator (==) (!=)

The == version of equality is quite liberal. Values may be considered equal even if they are different types, since the operator will force coercion of one or both operators into a single type (usually a number) before performing a comparison.

The Strict Equals Operator (===) (!==)

This one's easy. If the operands are of different types the answer is always false. If they are of the same type an intuitive equality test is applied: object identifiers must reference the same object, strings must contain identical character sets, other primitives must share the same value. NaN, null and undefined will never === another type.

Pitfalls of Comparison

- Just because the value of a type is falsey ***does not*** mean that values of two different types are equal using the double equals. *(Ex, null and undefined)*

Functions

JavaScript Functions are First-Class Objects

- They can be assigned to variables, array entries, and properties of other objects.
- They can be passed as arguments to functions.
- They can be returned as values from functions.
- They can possess properties that can be dynamically created and assigned.

Functions

JavaScript Functions are composed of four parts:

- The function keyword.
- An optional name that, if specified, must be a valid JavaScript identifier.
- A comma-separated list of parameter names enclosed in parentheses.
- The body of the function, as a series of JavaScript statements enclosed in braces.

```
function addTwo(n) {
  return n + 3
}
```

Anonymous Functions

The function below is an **anonymous function** (a function without a name).
Functions stored in variables, do not need names. They are always invoked/called using the variable name.

```
const addThree = function(n){
  return n + 3
}
```

Arrow Functions

Arrow functions are functions defined with a new ES6 syntax that uses an “arrow” (=>).

An arrow function expression has a shorter syntax than a function expression and does not bind its own *this*, *arguments*, *super*, or *new.target*. Arrow functions are always anonymous.

```
const addThree = function(n) {
    return n + 3
}

// equivalent to:
const addThree = (n) => {
    return n + 3
}


// equivalent to:
const addThree = (n) => n + 3
```

Objects

Values of the type object are arbitrary collections of properties, and we can add or remove these properties as we please. One way to create an object is by using a curly brace notation.

```
const transformer = {
    name: 'Optimus Prime',
    team: 'Autobots',
    colors: ['red', 'blue', 'white']
}
```

Objects

Adding Properties

```
transformer.homeWorld = 'Cybertron'
transformer['vehicle'] = 'truck'
```

Accessing Properties

```
console.log(transformer.homeWorld)  // 'Cybertron'
console.log(transformer['vehicle']) // 'truck'
```

Objects ES5 vs ES6

ES5

```
var name = 'Cydney'
var
     key = 'occupation',
     val = 'software engineer'
var person = { name: name }
person[key] = val
```

ES6

```
const name = 'Cydney'
const
     key = occupation,
     val = software engineer

const person = { name, [key]: val }
```

Object Equality

```
const obj = { value: 10 }
const obj2 = { value: 10 }

console.log(obj == obj2) // false
```

JavaScript's == operator, when comparing objects, will return true ***only*** if both objects are precisely the same value. Comparing different objects will return false, even if they have identical contents. There is no "deep" comparison operation built into JavaScript.

Arrays

In JavaScript, objects and arrays are handled almost identically, because arrays are a special kind of object. Arrays have a length property but objects do not.

```
const alpha = ['a', 'b', 'c', 'd']
const beta = ['a', 1, {value: 10}, new Date()]
```

* If you assign a value to an element of an array whose index is greater than its length (for example, `alpha`[25] = "z"), the length property is automatically increased to the new length.

```
// [ 'a', 'b', 'c', 'd', , , , , , , , , , , , , , , , , , , , , , , 'z' ]
```

* If you make the length property smaller, any element whose index is outside the length of the array is deleted.

```
alpha.length = 1  //['a']
```

Arrays

Adding Properties

```
const alpha = ['a', 'b', 'c', 'd']
alpha.push('e');
// ['a', 'b', 'c', 'd', 'e']
```

Accessing Properties

```
console.log(alpha[2]) //'c'
```

Iterating Over Arrays and Strings

Loops offer a quick and simple way to perform some action or actions repeatedly. Both arrays and strings can be iterated over using a For loop.

```
const arr = ['hello', 'world', '!']
for (let i = 0; i < arr.length; i++) {
     console.log(arr[i])
}

const str = 'hello world !'
for (let i = 0; i < str.length; i++) {
     console.log(str[i])
}
```