# CS454 Node.js & Angular.js

Cydney Auman
Albert Cervantes
CSULA

## Introduction to Javascript - Week 1

# Setup

- Install Node.js (v4.0.0 or higher) - nodejs.org
  - after install - in your terminal run the command: `node -v`
  - the output should be v4.x.x

- Install Sublime Text - sublimetext.com/3

- Run your .js file from your terminal by executing the command: `node filename.js`
  - make sure that this command is executed in the same directory as your file is located

# What is JavaScript?

JavaScript is a cross-platform, lightweight, interpreted, prototype-based object-oriented language with first-class functions.  It is most known as the scripting language for web pages, but used in many non-browser environments as well such as Node.js.

# 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.

# Java vs JavaScript

It is important to note that JavaScript has almost nothing to do with the Java.

The similar name was inspired by marketing considerations, rather than good judgment. When JavaScript was being introduced, the Java language was being heavily marketed and was gaining popularity. Someone thought it was a good idea to try to ride along on this success.

# Java vs JavaScript

| JavaScript | Java |
| --- | --- |
| JavaScript is interpreted at runtime, it does not need to be compiled into another form to run. Node.js uses Google V8 JavaScript engine to execute code. | Java programs require Java Virtual Machine (JVM) for execution and need to be compiled into “byte code” before they can be executed by the JVM. |
| Objects are prototype-based. | Objects are class-based. |
| Dynamic<br>- variables are declared using the `var` keyword in and can accept different kinds of value | Static<br>- variables are declared with type at compile time, and can only accept values permitted for that type |
| Functions are first class. | Methods are not first class. |

# Declaration

Variables in standard JavaScript have no type attached, and any value can be stored in any variable.

Variables are declared with a `var` statement, multiple variables can be declared at once.

```
var greeting = 'hello world';
var alpha, beta, gamma;
var alpha = 5, beta = 'hello', gamma = 'world';
```

# Declaration

JavaScript is dynamically typed - this means you don't need to specify the datatype of the variable when it is declared

In Java we would say:

```
int year = 2015;
String greeting = 'hello world';
```

BUT in JavaScript:

```
var year = 2015;
var greeting = 'hello world';
```

# 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.

# 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.

# Functions

**Two Ways to Define a JavaScript Function**

```
// define a function with a name of addThree
function addThree(n) {
  return n + 3;
}

// define a function and set it to a variable
var addTwo =  function(n) {
  return n + 2;
}
```

# 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.

```
var addTwo =  function(n) {
  return n + 2;
}
```

# 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.

```
var robot = {
    name: 'Optimus Prime',
    team: 'Autobot',
    colors: ['red', 'blue', 'white']
};
```

# Objects

### Adding Properties

```
robot.homeWorld = 'Cybertron';
```

### Accessing Properties

```
console.log(robot.homeWorld); // 'Cybertron'
console.log(robot[homeWorld]); // 'Cybertron'
```

# Iterating Over Objects

**The for...in statement iterates over the enumerable properties of an object.**

```
var robot = {
    name: 'Optimus Prime',
    team: 'Autobot',
    colors: ['red', 'blue', 'white'],
    homeWorld: 'Cybertron'
};

for (var key in robot) {
  console.log(key, robot[key]);
}
```

# Objects

**Object Equality**

```
var obj = { value: 10 };
var 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.

```
var alpha = ['a', 'b', 'c', 'd'];
var 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`[26] = "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

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

### Accessing Properties

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

# Iterating Over Arrays

Arrays provide methods to iterate over and manipulate members. The following example shows how to obtain the properties of objects stored in an array.

```
var arr = ['hello', 'world', '!']

//using a for loop
for (i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}
// using the forEach method provided by Array
arr.forEach(function (word) {
  console.log(word);
});
```