JavaScript Fundamentals

Lecture 4
Chapters 1-5 in JavaScript textbook

Topics

- We’re taking a break from the browser, and will be writing some simple JavaScript programs

- They will execute in the Node.js environment
  - If you have yet to install Node.js on your system - do so now!!!

- We’ll cover JavaScript variables, statements, conditionals, loops.

History of JavaScript

JavaScript was created by Netscape and was released in 1995.

JavaScript was developed to be a language that let the programmer manipulate HTML loaded into the browser.

It looks a lot like Java and C++, but its actually vastly different “under the hood”.

History of JavaScript

- JavaScript is not Java - they included the "Java" part because Java was very exciting ...
- While it used to be considered a "scripting" language, that's not the case anymore...
  - Modern JavaScript is becoming viable as a general purpose programming language
- There actually is no such thing as JavaScript!
  - JavaScript is a trademark of Oracle (Sun)
  - JScript was Microsoft's implementation (dead)
  - EMCAScript is the ***actual*** standard
    - European Computer Manufacturer's Association

How JavaScript runs

- C and C++ are languages that let you:
  - Create variable, arrays, and objects
  - Perform calculations
  - Call functions
- In C and C++, you can interact with the Operating System through standard API's to perform I/O.
- The OS would be considered the *execution host*.

- Java is similar, but the host is the JVM

How JavaScript runs

- Unlike C, C++, and Java - JavaScript doesn't actually define I/O!
- The *host* is a program, traditionally the web browser.
- The browser acts as an *interpreter* of JavaScript.
  - For obvious (?) reasons, JavaScript never initially had access to the file system, a command line, etc.
  - It interacted with the user through host functions like "alert" and "prompt"

We'll get back to the browser shortly… and deal with browser execution of JavaScript in detail

Chrome, V8, and Node.js

Once upon a time, JavaScript was only interpreted - and "run" by the browser. It was **slow**.

With the release of Chrome (2008), Google released an open source JavaScript "engine" called V8

- V8 compiles JavaScript into *native* machine code and is heavily optimized.
- The result is that JavaScript (once downloaded and compiled by V8) runs exceptionally fast… like C++.

V8 and Node.js

- Chrome ships with V8, and when it downloads JavaScript it executes it with V8.
  - Its the same old JavaScript though…

- Node.js is an open source platform developed in 2009 which wraps V8 in a *server* environment.
  - Provides access to OS using a POSIX-like API
    - File input and output
    - Networking with Sockets
    - etc.

V8 and Node.js

- When you install Node.js, you are also installing Google's V8 engine.
- Node is a host environment for JavaScript, and provides objects and functions for you to call.
- However for the most part, its just JavaScript - without a browser.

Node is a command line system. You'll execute JavaScript by typing "node file.js" at the command line - where file.js contains your JavaScript

JavaScript and C++

- JavaScript is case sensitive
  - Remember, HTML is not…
- Whitespace in JavaScript is just like C++
- Comments are just like C++ too.
- … and JavaScript variable / function / class naming rules are the same as C++ too!
  - Although $ are also allowed as characters in identifiers*

You will pick JavaScript up quickly… but there are some very serious differences between it and C++!

* while not technically allowed by the C++ standard, the $ is actually supported in Visual Studio and g++

Simple comparison

```cpp
#include <iostream>
using namespace std;

int main() {
     int x = 4;
     int y = 5;

     cout << (x + y) << endl;
}
```

```
sfrees@inspiron ~/Desktop $ g++ -o test test.cpp
sfrees@inspiron ~/Desktop $ ./test
9
```

- You compile a C++ program into an executable, then run *directly*.
- C++ has strong typing
- C++ requires a main function as its entry point
- C++ requires semi-colons

```javascript
var x = 4;
var y = 5;
console.log(4+5);
```

```
sfrees@inspiron ~/Desktop $ node test.js
9
```

- **node** (or another host) compiles and executes JavaScript
- JavaScript is dynamically typed
- All uncontained code is executed
- Semi-colons are technically optional (although this is rarely a “win”)

Reserved Words

See page 24-25 in text book for list of reserved words. We'll learn many as we go...

Interestingly, JavaScript reserves many keywords that are not actually used - they are reserved for possible future use, or to avoid confusion...

As we'll see - JavaScript is "distributed" as source code - so code written now needs to "compile" on someones browser in 2021...

Datatypes...

Sometimes people say "JavaScript doesn't have data types".... that's ridiculous.

```
var x = 5;  // var is a keyword
```

- However - when you *declare a variable*, you don't *specify* its datatype explicitly.
- The datatype of a variable depends on what you put into it!

Datatypes

Two kinds of types:

1. **Primitive**: numbers, strings, booleans, null, undefined.
2. **Objects**: anything with properties (name-value pairs), which includes **arrays** and **functions**.

Unlike other languages, the datatypes names are not really keywords - since you never explicitly use them!

Numbers

- There is no difference between integers, floating-point values, and no such thing as "short" or "long"
- All numbers are represented as 64-bit IEEE 754 standard floating point values
  - Absolute value max +/- 1.8×10^{308}
  - Absolute value min +/- 5×10^{324}

- Generally, the syntax of numeric literals follows what you've seen in C++

Arithmetic

The standard numeric operators apply (+-*/%)

There is a Math object built into the language

- Math.pow(2, 4) will give you 16
- Math.random() will give you a random number between 0 and 1
- Math.PI will give you a high-precision constant for PI

See page 33 in the JavaScript text

Strange numbers

- JavaScript includes **Infinity** and **-Infinity** in its numeric set.
- When comparing Infinity with *anything*, Infinity is always larger…
  - `console.log(5/0)` prints "Infinity"
  - `console.log(Infinity == Infinity)` prints "true"
- NaN stands for "Not a Number".
  - `console.log (0/0)` is NaN
  - `console.log(NaN == NaN) //` take a guess

Text

- A string is an immutable, ordered sequence of characters
  - strings cannot be changed (like numbers)
  - strings are in Unicode by default
- Strings can be delimited by double or single quotes (although they need to match up)
- Escape sequences are similar to C++ (\t\n\\)

Strings

- Strings can be concatenated with + operator
- Most things can be cast to strings when used with the + operator.

```
console.log(5 + "hello");
```
prints “5hello”

```
console.log("6" + 5);
```
prints “65”

JavaScript has excellent support for regular expressions and pattern matching.
We will revisit this as needed.

Boolean values

- Boolean values work just like C++, and `true`/`false` are both keywords.

- Any type can be converted to a boolean
  - The following all convert to `false`

    `undefined` and `null`

    `0` and `-0`

    `NaN`

    `""` and `''` (empty strings)

Everything else is "truthy"

null and undefined

- `null` represents the *absence of a value*
  - any **variable** without a **value** is null.
  - typedef(null) returns "object"
  - null is actually considered a primitive "type"

- `undefined` means the "thing" doesn't exist!
  - Functions can return "undefined"

```
console.log(undefined == null)
console.log(undefined === null)
```

=== ?

There are two equality operators

== means “equal, with an effort to cast”

(“5” == 5 is true)

=== means equal, in value and type

(“5” === 5 is false)

Check out http://www.w3schools.com/js/js_comparisons.asp

Global Object

The Global object: Anything that is not part of an object is technically part of the **global** object

- Code declared outside functions
- `Math`, `JSON` objects
- `isNaN`, `parseInt`
- `Date()`, `String()`, `Object()`, `Array()`

Hosts can attach other objects to global

- Methods like `alert` and `prompt`
- Objects like `console`

Wrapper Objects

Like Java, primitives also have full-scale object types that behave similarly

```
var s = “hello”; // string
var str = new String(s);
console.log(str.substring(1, 3));
```

Primitives are automatically wrapped in their counterparts as needed

```
console.log(s.substring(1, 3));
```

Type conversions

- We've already seen numbers automatically turned into strings… `console.log("5" + 6);`
- A + operator tries to convert to strings if any operand is a string. If there are no string, it will try to convert all to numbers (booleans)
- A * operator will try converting operands to numbers though - since * only makes sense on numbers

`console.log("5" * "4")` prints 20!

There is a full table on all the rules of conversion on page 46 in the JavaScript text book.

Of course when in doubt - test!

Parsing numbers

Type conversion of strings to numbers is limited

For full parsing capabilities, use parseInt(string) and parseFloat(string)

These functions can parse hex strings, strings with trailing spaces, and even sentences (with characters) as long as they start with numbers!

```
console.log( 3 + parseInt("3 blind mice")); prints 6!
```

Variable declaration

Variables should be declared before use

```
var my_new_variable;
```

The initial value is always undefined if not specified.

Variables change types at will.

```
var x = 5;
x = “hello”;
```

This is why its called Dynamically Typed

Undeclared variables

When trying to write to an undeclared variable, the variable is simply declared implicitly

If you attempt to read an undeclared variable, you'll get an **exception**.

Redeclaration of variables (in same scope) have no effect.

Variable Scope

Scope *is similar to* C++, but be careful with leaving out the var keyword!

```
var scope;
function checkScope() {
    scope = 5;    // this is changing the global variable,
                  // not creating a local!
    var scope = 6;  // local variable
}
```

However there are some BIG differences!

Block vs. Function Scope

In C++ variables are scoped within { and }

- It doesn't matter what the { and } are attached to… could be for loop, if statement, switch, or function…
- This is called **block** scoping

JavaScript uses **function** scoping.

Function Scoping

```
function test() {
    var i = 0;
    if ( i == 1 ) {
        var j = 5;
    }
    console.log (j);
}
```

Function scoping has significant implications, and make JavaScript a *functional* language.

We'll learn more about this when we start discussing *closures*.

Note - no error is thrown, j is actually defined (but not initialized). Console output is "undefined"

Operators and Expressions

Sections 4.1-4.11 largely cover operators and expression that do not differ from what you already know (C++)

- They also discuss syntax and operators for arrays and object, which we'll cover later.

An exception:

- === vs. ==, !== vs. !=
- The === and !== performs no type conversion

```
console.log("3" == 3);
console.log("3" === 3);
```

Statements (Chapter 5)

Chapter 5 in the text similarly introduces concepts you are already familiar with.

Please make sure you've read Chapter 5.

- if statements, else if, else
- switch statements
- for loops, while loops, do/while loops
- note the for/in statement for objects

Next Lecture

We will cover Chapter 6 - Objects next class

That will be followed by Chapters 7 and 8 - arrays and functions

Please keep up with the reading!