JavaScript: The Language

STEPHEN SCHAUB

Topics

- Language Overview
- Variables and Types
- Operators and Expressions
- Statements

What is JavaScript

- "The World's Most Misunderstood Programming Language"
  - - Douglas Crockford
- Contains many good ideas and some horrible ones
- Used by a wide range of programmers
  - From computer scientists to cut-n-pasters

Language Introduction

- Invented by Brendan Eich at Netscape
- Standardized as ECMAScript
- Popular implementations
  - V8 (Google Chrome, Node.js, Edge)
  - Chakra (IE)
  - Nashorn (Java)
- Latest version: ECMAScript 2018 (Edition 9)
  - Browser support varies

JavaScript Outside the Browser

- Use JavaScript to
  - Write Windows shell scripts (since Windows 98)
  - Script popular apps (Adobe Creative Suite, OpenOffice)
  - Write server-side apps (Node.js)

Birth of JavaScript

- Netscape hired Eich to design LiveScript
- Eich wanted to write a Scheme interpreter
- Netscape wanted a language for the masses
- Eich had 10 days
- JavaScript was born
- Eich was CTO at Mozilla Corporation until 2014

Language Overview

- Dynamic scripting language featuring
  - C syntax
  - Smalltalk / LISP semantics
- Object oriented
  - Prototypes and (more recently) Classes
- Functional features

Standard Library

- Extremely small
  - Math methods
  - String, Array, Date, RegExp objects
- No I/O mechanisms
  - Completely dependent on API provided by hosting environment for UI concerns
  - console.log( ) is available in browser and server environments for debugging
- Today, the language only

Syntax Basics

- Case sensitive
- Freeform syntax
- C++ style comments
- Semicolon statement terminator optional
  - Best practice: Use semicolons

Data Types

Data Types

- String
  - "Something in quotes" (single or double)
- Number
  - Floating-point values
  - No integer type
- Boolean
  - true, false
- Null
  - null
- Object / Array
- Function

Value Types vs. Reference Types

- Like Java/C#:
  - Numbers and Booleans are value types
  - Other types are reference types
- Example:
  - var arr = new Array();
    var arr2 = arr; // arr2 has a reference to arr's array
    arr2[0] = 25; // alters the single array referenced by arr / arr2

Numbers

- JavaScript represents all numbers as floating point
- Special value NaN results from illegal numeric operations
  - Use isNaN(value) to test for this value

Strings

- String literals use either single or double quotes
  - No separate char type
- C-style Escape sequences

Escape Sequence	Character	Meaning
\ddd	0ddd	octal character
\xdd	0xdd	hexadecimal character
\\	\	backslash
\'	'	single quote
\“	”	double quote
\b	BS	backspace
\f	FF	form feed
\n	NL or LF	new line (or line feed)
\r	CR	carriage return
\t	HT	horizontal tab
\	<new line>	continuation

String Operations

- Concatenate strings:
  - str1 + str2
- Determine length of string:
  - str.length
- Access character at index:
  - str[index]
- Extract substring:
  - str.substring(start, len)
- Compare strings:
  - str1 < str2

Numbers

- JavaScript represents numbers internally as floating point values
- Convert string to number
  - parseInt(str)
  - parseFloat(str)

Boolean

- Literal values: true, false
- Other values are interpreted as boolean by if / while
  - Interpreted as false:
    - 0, -0, null, "", false, undefined, NaN
  - Other values are true:
    - Any nonempty string (including "0" and "0.0")
    - Nonzero number
    - Any array or object

Variables

Variables and Types

- Like Python, variables don't have types…
- **Values do.**
- Variables can hold values of different types over their lifetime

```
myvar = 5;          // it's a double now
myvar = "5";        // it's a string now
myvar = null;       // it's a null now
```

Defining Variables

Three ways to define variables:

- let statement (preferred)
  - let z; // define
  - let z = 5; // define and initialize
- var statement
  - var x;
  - var x = 10;
- assignment statement
  - y = 0; // creates y if it does not exist

undefined vs. null

- JavaScript includes two related values:
  - undefined - the default value for uninitialized variables
  - null - used to indicate the explicit absence of a value
- The following are different:
  - let myvar; // myvar's value is **undefined**
  - let myvar = null; // myvar's value is **null**

Undeclared vs. Undefined

- Using undeclared variables in an expression causes a runtime error

```
x = y + 1;  // causes crash if y is undeclared
```

- Using a variable with value **undefined** is legal

```
let x;
let y = x; // stores value undefined in y
```

Variable Scope

- **var** supports only two scope levels
  - Creates a global when used **outside** a function
  - Creates a local when used **inside** a function
- **let** supports block scoping
- Assignment to an undeclared variable always creates a global
  - x = 0; // if x undeclared, creates global

Locals vs. Globals

- Create a global variable #1 (preferred):

```
let x;
function foo() {
  console.log (x); // legal - "undefined"
}
```

- Create a global variable #2:

```
x = 5;
function foo() {
  console.log (x);      // 5
}
```

Locals vs. Globals

- Create a global variable #3 (ugh):

```
function foo() {
  x = 5;
}
foo();
console.log(x); // 5
```

- Create a local variable:

```
x = 5;
function foo() {
  let x = 10;
  console.log(x); // 10
}
foo();
console.log(x); // 5
```

Avoid using var

- Local variables defined with **var** are always implicitly "hoisted" to the top of a function
  - But the initialization occurs on the line where it is written
- Leads to confusing behavior

```
var scope = "global";

function f( ) {
  alert(scope); // Displays "undefined", not "global"
  var scope = "local"; // Variable initialized here
  alert(scope); // Displays "local"
}
f( );
```

Variable Definition Recommendations

- Staying out of trouble with variables:
  - Prefer defining variables with **let**
- Better yet: Use "Strict" mode

Strict Mode

- Put at top of script:
  - "use strict"; // include the quotes
- Requires all variables be defined with **var** or **let**
- Turns undesirable behavior into errors, and reduces the number of unwanted surprises
- For details:
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

Operators and Expressions

Operators

- C-style operators for
  - Assignment
  - Math
  - Comparison
  - Logic
  - Bit

Addition vs. Concatenation

- JavaScript uses + for both addition and concatenation
  - An unfortunate design choice
- var x = y + z;
  - If either y or z is a string (or object), concatenation occurs
  - If both are numeric, addition occurs
- To prevent problems, use parseInt() / parseFloat() when uncertain about the value
  - var x = parseInt(y) + parseInt(z);

Comparing Values

- JavaScript provides the usual C-style comparison operators:

== equal	!= not equal
< less than	<= less than or equal to
> greater than	>= greater than or equal to

- Things work as expected when the two values being compared are the same type
- The plot thickens when different types are involved

JavaScript Coercion

- Consider:
  - val1 = prompt("Enter a number:"); // user enters nothing
    if (val1 == 0) {
      /* surprise - they are equal!! */
    }
- The rules for JavaScript type coercion in comparison operations are arcane and hard to remember
  - A common source of tricky bugs

Equals vs. Identical

- Safer approach: Use
  - === identical
    - true only if both values are same type and value
  - !== not identical
    - the logical negation of identical
- Example:
  - if (val1 === 0) { /* true only if val1 is the number 0 */ }
- Douglas Crockford:
  - Prefer === and !==
  - Think of == and != as the "evil twins" of === and !==

Logical Operators

- C-style:
  - ! Not
  - && And
  - || Or
- && and || are short-circuiting
  - yield operand values
- Example:
  - var max = max_width || preferences.max_width || 500;
  - Idiomatic usage: selects first value that is defined and not null

Statements

Statements

- JavaScript provides C++/Java-style control statements:
  - if / else
  - while
  - do while
  - for
  - switch
  - try / catch / finally

for loops

- Two forms:
  - for (*initialize* ; *test* ; *increment*) { *body* }
    - for (let i = 0; i < 5; ++i) { … }
  - for (*variable* in *object*) { *body* }
    - for (let prop in obj) { … }

try / catch / finally

- Syntax:
  - ```
    try {
      // guarded statements
    }
    catch (e) {
      // exception handler
    }
    finally {
      // cleanup code
    }
    ```
- Only one catch block allowed
- finally block guaranteed to execute
- Exception parameter receives object thrown by throw statement
    ```

throw

- Raises an exception
- Works like Java/C#
  - throw new Error("Can't do that!");