JAVASCRIPT

You Don't Know JS by Kyle Simpson

AykeriZero

Basic Programming (2018-05-14)

Interpreting

- interpretor translates code into machine language every time the program is run

Compiling

- compiler compiles the code before runtime

Javascript

- code is processed each run
- compiles the program then runs
- usually called interpreting, but not exactly true

Developer Tools Console

```
Console.log(...);
age = prompt("What is your age?");
```

Implicit Coercion

- js will change types implicitly
- e.g. "99.99" == 99.99
  returns true

Static Typing / Type Enforcement

- ensures that a variable holds a specific type

Weak Typing / Dynamic Typing

- variables can hold values of any type

Commenting

```
// this is a comment
/* this
   is a
   comment
*/
```

Functions

- can have return values

```
function functionName (parameters) {
}
```

Review

- operators
- values and types
- variables
- conditionals
- loops
- functions

```
const SPENDING_THRESHOLD = 200;
const TAX_RATE = 0.08;
const PHONE_PRICE = 99.99;
const ACCESSORY_PRICE = 9.99;

var bank_balance = 303.91;
var amount = 0;

function calculateTax(amount) {
    return amount * TAX_RATE;
}

function formatAmount(amount) {
    return "$" + amount.toFixed( 2 );
}

// keep buying phones while you still have money
while (amount < bank_balance) {
    // buy a new phone!
    amount = amount + PHONE_PRICE;

    // can we afford the accessory?
    if (amount < SPENDING_THRESHOLD) {
        amount = amount + ACCESSORY_PRICE;
    }
}

// don't forget to pay the government, too
amount = amount + calculateTax( amount );

console.log(
    "Your purchase: " + formatAmount( amount )
);
// Your purchase: $334.76

// can you actually afford this purchase?
if (amount > bank_balance) {
    console.log(
        "You can't afford this purchase. :("
    );
}
// You can't afford this purchase. :(
```

Note: The simplest way to run this JavaScript program is to type it into the developer console of your nearest browser.

Intro Javascript (2018-05-21)

Javascript Version 6

- 6th edition of ECMAScript (ES6)

Values and Types

- Javascript has typed values, not typed variables
- built-in types
  - string
  - number
  - boolean
  - null and undefined
  - object
  - symbol (new to ES6)

```
var a;
typeof a;               // "undefined"

a = "hello world";
typeof a;               // "string"

a = 42;
typeof a;               // "number"

a = true;
typeof a;               // "boolean"

a = null;
typeof a;               // "object" -- weird, bug

a = undefined;
typeof a;               // "undefined"

a = { b: "c" };
typeof a;               // "object"
```

Built-In Type Methods

native / object wrapper

- string, number, and boolean each have String, Number, and Boolean wrappers which define methods in their prototype

Coercion

- there are two types — explicit and implicit coercion

Equality

- == checks for value equality with coercion
- === checks for value equality without coercion

Objects

- properties can be accessed with dot or bracket notation.
- bracket notation is used when the property name has special characters
- bracket notation is also used when the name of another property is stored in another variable

```
var obj = {
    a: "hello world",
    b: 42,
    c: true
};

obj.a;          // "hello world"
obj.b;          // 42
obj.c;          // true

obj["a"];       // "hello world"
obj["b"];       // 42
obj["c"];       // true
```

Arrays

```
var arr = [
    "hello world",
    42,
    true
];

arr[0];             // "hello world"
arr[1];             // 42
arr[2];             // true
arr.length;         // 3

typeof arr;         // "object"
```

Functions

```
function foo() {
    return 42;
}

foo.bar = "hello world";

typeof foo;             // "function"
typeof foo();           // "number"
typeof foo.bar;         // "string"
```

Comparing Objects and Arrays

- objects and arrays are held by reference so comparing the two does not consider contents
- arrays are by default coerced into strings by joining all the values with commas inbetween

Inequality

- Javascript String values can be compared in alphabetical order

Built-in Type Methods

- String object wrapper (also called a native) pairs with primative String type
- object wrapper defines the toUpperCase() method

```
var a = "hello world";
var b = 3.14159;

a.length;
a.toUpperCase();
b.toFixed(4);
```

Function Scopes

- the var keyword declares a variable that can be in a function's scope or a global scope
- trying to access a variable out of scope creates a ReferenceError (exception?)
- if you try to set an undeclared variable, you either create a global scope variable or get an error.

Hoisting

- whenever var appears, the declaration is accessible throughout, even before the declaration
- bad coding practice to use hoisted variable
- hoisting is usually used for function declarations

```
//scope and hoisting

var a = 3;
foo();                          //hoisted function
console.log(a);

function foo() {
  a = 2;                        //hoisted variable
  console.log(a);
  var a;
}
```

let keyword

- let keyword limits the scope of declared variables to certain blocks

```
if(true){
    let a = 1;
}
// a is out of scope
```

Switch Statements

```
switch (a) {
    case 2:
        // some code
        break;
    case 10:
        // something else
        break;
    default:
        // fallback code
}
```

Conditional operator / ternary operator

```
var b= (a > 41) ? "hello" : "world";

/*
if (a > 41) {
   b = "hello";
else {
   b = "world";
}
```

Strict Mode

- disallows implicit global variable declaration by omitting var keyword
- can be used for a function scope or the entire file

```
"use strict";
```

Functions as Values

Anonymous Functions

```
var foo = function() {
    // some code
}
```

Named Functions

```
var foo = function bar() {
    // some code
}
```

Application Programming Interface (API)

Immediately Invoked Function Expressions (IIFE)

- often used to declare variables that won't affect the surrounding code

```
(function foo() {
    // some code
})();
```

Closure

- a way to "remember" a function's scope even once the function has finished running

```
function makeAdder(x) {
    function add(y) {
        return x+y;
    };
    return add;
};
```

Modules

- allow separation of interface and implementation

```
function User(){
    var username, password;

    function doLogin(user,pw) {
        username = user;
        password = pw;

        // do the rest of the login work
    }

    var publicAPI = {
        login: doLogin
    };

    return publicAPI;
}

// create a `User` module instance
var fred = User();

fred.login( "fred", "12Battery34!" );
```

— object with one property / method

— login() closure keeps var username, password

this Identifier

- this objed depends on the context in which the function is called

Prototypes

- if you try to reference a nonexistant property, Javascript will use the object's internal prototype to find another object to look for the prototype on
- sometimes used to emulate a class system

New Features

- new features of Javascript are not avaliable in older browsers

Polyfilling

- implementing a new code feature using a piece of code which produces the same behavior

Transpiling

- used when new syntax has been added to the language
- a tool converts newer code into its older equivalent

Scope and Closures (2018-07-14)

Compiler Theory

- traditial compiled code
  1. Tokenizing/Lexing
  2. Parsing
  3. Code Generation
- Javascript compilation is different because it must be done microseconds before execution

LHS vs RHS

LHS

- lookup a variable to see if it exists

RHS

- lookup the value of a variable

Nested Scope

- engine checks the innermost scope first, working outwards until global

ReferenceError

- RHS lookup fails
- in non-strict mode LHS lookup fails will create a global variable

TypeError

- RHS lookup succeeds but the code used on the value is wrong

Lexical Scope (2018-07-14)

Lexical vs Dynamic Scope

- two models for how scope works
- Lexical is the most common and the one JS uses

Cheating Lexical Scope

- cheating lexical scope leads to poorer performance

eval(...)

- takes a string as an argument and treats it like code at that point in the program
- in strict mode, eval has its own scope

with keyword

- takes an object and treats it as if the object is a separate lexical scope
- easy to accidentally create global variables
- not allowed in strict mode

- if eval or with are used the JS engine will not use its scope optimizations because they might be useless

Function vs Block Scope (2018-07-14)

Functions
- create their own scope bubble

Hiding Scope
- you can hide variable by putting them inside a function scope and thus thus allowing a minimal API "Principle of Least Priviledge" / "Least Authority" / "Least Exposure"
- also avoids unintended collisions

Global Namespaces
- in order to avoid collision, some libraries will create a single object as a namespace with functions / variables as properties of that object

Module Management
- no libraries are allowed to import identifiers into the global scope and instead import their identifiers into their own specific scope using a dependency manager

Function Expression
- code inside the function executes
- the function name does not pollute the enclosing scope
- can be anonymous
  - usually better practice to name function expressions
- Immediately invoked function expressions

```
(function foo(){
})();

(function () {
})();
```

Other things which block scope
- with
- try / catch

Let Keyword
- forces block scoping
- will not hoist

Const Keyword
- is block scoped

HOISTING (2017-07-17)

Declarations

- function declarations and variable declarations are hoisted
- function declarations are hoisted before variable declarations

Scope Closure (2017-07-17)

Closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope

```
funtion foo(){
    var a = 2;

    function bar() {
        console.log(a);
    }

    return bar;
}

var baz = foo();

baz();    //2
```

Module Pattern

1. outer enclosing function which creates a module instance
2. outer function must return an inner function which has closure over the private scope

Import / Export

- modules must be declared in their own separate files

bar.js
```
export hello;
```
foo.js
```
import hello from "bar";
hello();
export hi;
```
baz.js
```
module foo from "foo";
```

Dynamic Scope (2018-07-18)

- explained as a contrast to Lexical scope which is used by JS and most languages
- only cares about where the functions are called from
- similar to the "this" mechanism in JS

Polyfilling Block Scope (2017-07-18)

- use try and catch statements

Lexical This (2018-07-18)

- arrow functions do not behave like normal functions in this binding

This and Object Prototypes (2018-07-18)