CS144: JavaScript

- Started as a simple script in a Web page that is interpreted and run by the browser
  - Supported by most modern browsers
  - Allows dynamic update of a web page
  - More generally, allows running an arbitrary code inside a browser!
    - Both a blessing and a curse
- Now, JavaScript can run anywhere, phone, tablet, desktop, server, not just in a browser

History

- 1995 Netscape Navigator added a support for a simple scripting language named “LiveScript”
  - Renamed it to “JavaScript” in 1996
  - JavaScript has nothing to do with Java!
- 1997 ECMA International standardized the language submitted by Netscape
  - ECMAScript: Official name of the standard
  - Javascript: What people call it
- 1998 ECMAScript 2, 1999 ECMAScript 3
- ECMAScript 4 abandoned due to disagreement
- 2009 ECMAScript 5
- 2015 ECMAScript 6 (= ECMAScript 2015)
  - Yearly release of new standard from ECMAScript 2015
- We learn syntax based on ECMAScript 2015
  - Most books and online tutorials are based on ECMAScript 5
  - A lot of ECMAScript 5 legacy code exist today
  - Our syntax may be different from these
  - But the newer standard removes much ugliness of old JavaScript

Basic keywords and syntax

- Syntax is very close to java/c
  - `if (cond){ stmt; } else if (cond){ stmt; }`

  - `switch (a){ case 1: break; ... default: ...; }`
  - `while (i < 0){ stmt; }`
  - `for (i=0; i < 10; i++){ stmt; }`
  - `for (e of array){ stmt; } //loop over array-like elements`
    - Note "of". "in" operator checks the existence of a property
  - `try { throw 1; } catch (e){ stmt; } finally { stmt; }`

- JavaScript is *case sensitive*
  - But HTML is *NOT*. This discrepancy sometimes causes confusion.
- Variables
  - `let name=value;` // variable type is dynamic
  - A variable can be used without an explicit `let` declaration
    - becomes a global variable
    - But this is ***strongly*** discouraged
  - Constant: `const n = 42; //n cannot be reassigned or redeclared`
  - Before ECMAScript 2015, `var` was used instead of `let` with some differences (more on `var` later)
- Function declaration statement

```
function func_name(parameter1, parameter2,...)
{
    ... function body ...
    return value;
}
```

- JavaScript identifiers (like variable or function name) may have letters, numbers, `_`, and `$`
- Comparison operators
  - `==`/`!=` true if operands have the same value (after type conversion)
  - `===`/`!==` true only if operands have the same *value and type* (no automatic type conversion)
    - `3 == "3"` vs `3 === "3"`
  - When operands are objects, `==`/`===` returns true only if both operands reference the same object (more on this later)
  - Logical AND and OR operators: `&&` and `||`

Primitive Types

- JavaScript is a dynamically-typed language
  - Variables do not have a static type. Types may change over time.

```
let a = 10; // a is number type
a = "good"; // a is string type
```

- Types are either “primitive type” or “object type”
- `typeof` operator returns the current type of the variable
  - But not exactly according to the standard due to legacy code. More on this later.
- Primitive data types
  - `number`, `string`, `boolean` (and `null` and `undefined`)

`number` type

- All numbers are represented as a floating point number (double in C). No separate “integer” type
  - Bitwise operators (`&`, `|`, `^`, `>>`, `<<`) represent a number as a 32-bit integer after truncating subdecimal digits
- `NaN` and `Infinity` are valid numbers

`boolean` type

- `true` or `false`
- other “falsy” values: 0, “”, `null`, `undefined`, `NaN`

`string` type

- Single or double quotes: `'John'` or `"John"`
- `length` property returns the length of the string
- Many useful string functions exist: `charAt()`, `substring()`, `indexOf()`, ...

```
let a = "abcdef";
b = a.substring(1, 4); // b = "bcd"
```

- `number`s and `string` are automatically type converted to each other
  - “3” * “4” = 12
  - 1+“2” = “12”
- For explicit type conversion, use `Number()`, `String()`, `Boolean()`, `parseFloat()`, `parseInt()`,...

`undefined` and `null` type

- `undefined`: the type of the value `undefined`
  - A variable has the value `undefined` before initialization
- `null`: the type of the value `null`
  - `null` is mainly used to represent the absence of an object
  - For legacy reasons, most systems return `object` as the type of `null` value
- `undefined` and `null` are often interchangeably used, but they are different in principle

```
undefined == null;  // true
undefined === null; // false
```

- `typeof null` is `object` for legacy issues

Object Type

- All non-primitive types in JavaScript are *object type*
- *Object*: data with a set of “properties”

```
let o = { x: 1, y: "good" };
let c = o.x + o["y"];
```

  - Note: `o["x"]` is identical to `o.x`. Objects are essentially an associative array.
- Object can be nested

```
let o = { x: 1, y: 2, z:{ x: 3, y: 4 } };
```

- Properties can be dynamically added, removed, and listed

```
let o = new Object();
o.x = 10;
o.y = 30;
delete o.x;
Object.keys(o);
```

- Object assignment is *by copying the reference*, not by copying the whole object

```
let o = { x: 10, y: 20 };
let p = o;
o.x = 30;
console.log(p.x);
```

- Object comparison is *by reference* not by value

```
let o = { x: 10 };
let p = { x: 10 };
console.log(o == p);
```

Array

- Array is a special object with integer-indexed items
- Created with `new Array()`, or `[ 1, 2, 3 ]`

```
let a = new Array(1, 2, 3);
let b = [1, 2];
console.log(a.length);
```

- `length` property returns the size of the array
  - Can be used to resize array by setting its value
- Array can be sparse and its elements types may be heterogeneous

```
let a = new Array();
a[0] = 3;
a[2] = "string";
```

```
let b = [1, "good", , [2, 3] ];
console.log(a.length)
```

  - Size of an array automatically increased whenever needed

- Array manipulation functions
  - *Mutators*: modifies input array directly
    * reverse, sort, push, pop, shift, unshift, splice
  - *Accessors*: input array stays in tact. new output array is created
    * concat, slice, filter, map

```
let a = [1, 2, 3, 4];
let b = a;
console.log(b);
a[1] = 5;
console.log(b);

let a = [1, 2, 3, 4];
let b = a.slice(1, 3);   // slice is an accessor
console.log(b);
a[1] = 5;
console.log(b);

let a = [1, 2, 3, 4];
let b = a;
console.log(b);
a = ["a", "b", "c"];
console.log(b);
```

Regular expression

- RegExp is a special object that describes a pattern to search for in a string

```
let r = /a?b*c/;
```

- Can be used in the following functions
  - String: search(), match(), replace(), split()

    - RegExp: exec(), test()

- Examples

```
/ABC/.test(str);          // true if str has substr ABC
/ABC/i.test(str);         // i ignores case
/[Aa]B[C-E]/.test(str);
'123abbbc'.search(/ab*c/); // 3 (position of 'a')
'12e34'.search(/[^\d]/);   // 2 [^x]: except x, \d: digit
```

Function

- In Javascript, functions are objects!
  - Functions can be assigned to a variable
  - Functions can be passed as a parameter

  - Functions can have properties

```
let square = function (x) { return x*x; };
    // anonymous function
    // function definition expression

square(10);

function myfunc(x, func) {
    return func(x);
}
myfunc(10, square);
myfunc(10, function (x) { return x * 2; });

myfunc.a = 20;
```

- Arrow function expression (ECMAScript 2015)
  - Shorthand notation for function definition expression
    - `(param1, ..., paramN)=> { statements }`
    - `(param1, ..., paramN)=> expression`
    - `singleParam => expression`

        - () => { statements }

```
let square = x => x*x;
console.log(square(10));
```

    - Very convenient to pass a function as a parameter in Node, Express, etc.

- Strictly speaking, a function is an object type according to the standard, but `typeof` returns "function".

Object-Oriented Programming (OOP)

- Objects can have methods

```
let o = new Object();
o.x = 1;
o.doubleX = function () { this.x *= 2; }
console.log(o.x);
```

- Inside inside an object's method, `this` points to the object
    - **Note**: Differently from `function (){...}`, arrow functions does not have its own `this`
        - `this` from the surrounding context is used
        - Do not use arrow functions for an object method or a constructor!

Class

- ECMAScript 2015 added more elegant syntax for classes and inheritance

```
class Shape {
    // Constuctor
    constructor(color) {
        this.color = color;
    }

    // Method
    printColor() {
        console.log(this.color);
    }
};

class Rectangle extends Shape {
    // Constructor
    constructor(color, width, height) {
        super(color);  // super refers to the parent class
        this.width = width;
        this.height = height;
    }
    // Getter
    get area() {
        return this.width * this.height;
    }
    // Setter
    set x(v) { this.coordX = v; console.log("this.coordX = " +
        v); }

};
let r = new Rectangle("red", 2, 3);
r.printColor();
console.log(r.area);
r.x = 1;
```

Scope

- Global vs local scope
  - A variable declared with `let` inside a block is valid only within the block: *block-scope local variable*
  - A variable declared outside of any block has *global scope*.
  - A variable that is assigned to a value without an explicit `let` declaration has *global scope*.
    * A variable created this way becomes a property of the *global object* (in case of browser, `window`)
    * It is ***strongly recommended not to create global variables this way***.

```
let a = "a"; // global vs local?
b = "b"; // global vs local?

function f()
{
    c = "c"; // global vs local?
    let d = "d"; // global vs local?
}
```

- `let` vs `var`
  - `let` was introduced only in ECMAScript 2015.
  - Before `let`, `var` was used with the following difference
    * function scope (not block scope)
    * hoisting (vs no hoisting)
      - declaration is “moved” to the top of its scope

```
var a = 10;  // global vs local?
function f() {
    b = 10;  // global vs local?
    console.log(b);
    var b;
}
f();
console.log(b);
```

  - Use of `let` produces much cleaner code! So use it
    * Unfortunately, many existing codes and examples still use `var`
- Functions can be nested
  - *lexical scope* (not dynamic scope) is used to determine the scope of local variables

```
function f() {
    let a = 1;
    let b = 2;

    function g() {
        console.log(a);  // a =
        console.log(b);  // b =
        b = 3;
    }

    if (a > 0) {
        let b = 4;
        g();
        console.log(b); // b =
    }
    console.log(b);     // b =
}
f(); // what will be printed?
```

Keyword `this`

- The meaning of `this` is a source of great confusion and bug in JavaScript
- Inside browswer, `window` object becomes the *global object*
  - Any variable assigned without declaration becomes a property of the global object
- Interpretation of `this`
  - At the top-most block (outside of any function call), `this` = global object
  - Inside a method call on an object (including constructor), `this` = the object
  - When called as an event handler inside a browser, `this` = DOM element to which the event handler was set
  - Inside all other function calls, `this` = the global object
- But arrow functions (`() => {}`) does not provide their own `this` binding
  - It retains the `this` value of the enclosing lexical context

```
x = 10;

function_printx = function() { console.log(this.x); }
arrow_printx = () => console.log(this.x)

o = { x: 20 };
o.printx_f = function_printx;
```

```
o.printx_a = arrow_printx;

// What will be printed?
console.log(this.x);
function_printx();
arrow_printx();
o.printx_f();
o.printx_a();
```

- **Note**
  - Do not use arrow functions to define a class method/constructor
  - Except inside class definition, use `this` only if it is absolutely necessary

JavaScript Object Notation (JSON)

- The standard syntax to represent literal objects in JavaScript (with some restrictions)
  - e.g., `[{ "x": 3, "y": "Good"}, { "x": 4, "y": "Bad"}]`
  - Q: What does the this notation mean in JavaScript?
  - Compared to JavaScript, the main differences are
    - Object property names *require* double quotes
    - Strings need *double quotes*, not single quotes
    - JSON values cannot be functions or undefined
- JSON-related functions:
  - `JSON.stringify(obj)`: JavaScript object -> JSON string
  - `JSON.parse(str)`: JSON string -> JavaScript object
- Example

```
let x = '[{ "x": 3, "y": "Good" }, { "x": 4, "y": "Bad" }]';
let o = JSON.parse(x);
let n = o[0].x + o[1].x;
console.log(n);
```

- JSON has become one of the two most popular data-exchange format on the Web
  - Based on JavaScript

- Easy to understand

Modules

ECMAScript 2015 Module

- ECMAScript 2015 added support for modules
  - One module <-> One JavaScript file
  - Everything in a module stays local unless declared `export`
  - `export` entities can be `import`ed and used by another JavaScript code
- Multiple named export example

```
//------ lib.js ------
export function square(x) {
    return x * x;
}
export function dist(x, y) {
    return Math.sqrt(square(x) + square(y));
}

//------ main.js ------
import { square, dist } from './lib';
square(11);
dist(4, 3);

//----- main2.js ------
import * as lib from './lib';
lib.square(11);
lib.dist(4, 3);
```

- Single default export

```
//------ myFunc.js ------
export default function () { ... }

//------ main1.js ------
import myFunc from 'myFunc';
```

```
myFunc();
```

References

- Javascript: The Definitive Guide by David Flanagan
  - Strongly recommended if you plan to code in JavaScript extensively
- ECMAScript standard: ECMA 262 https://www.ecma-international.org/ecma-262/
  - The ultimate reference on what is really correct
  - But very boring to read and learn from
  - Browser support is a few generations behind
- Summary of new features in ECMAScript 2015: http://es6-features.org/
- JSON standard: ECMA 404 http://www.json.org/