Overview

- Quick summary of the main features of JavaScript so that you can start work on programming projects.
- Details will be covered in subsequent classes.
- JavaScript primitive types.
- Variables and declarations.
- Arrays.
- Objects.
- Regular expressions.
- Control statements summary.
- Functions.
- Classes.
- Dynamic language.
- JS literals vs JSON.
- Gotchas.

Numbers

All primitive numbers are floating point; no primitive integer types. Examples under nodejs (typed command `node`).

```
> 1 + 2*3**3     //exponentiation ** has high precedence
55
> -5 % 3         //remainder; has sign of first operand
-2
> 3 << 2          //left shift equiv mult by power-of-2
12
> 15 >> 2         //right shift equiv divsn by power-of-2
3
> 0xf & 0x32     //bitwise and
2
> 0xf | 0x32     //bitwise or
63
> (0xf ^ 0x32).toString(16)   //xor; hex literals
'3d'
```

Strings

- Strings are immutable.
- String literals can be enclosed within single quotes as `'hello'` or double-quotes as `"world"`. Absolutely equivalent; I prefer using `'string'` as easier to type on my keyboard.
- Above string literals can contain escape sequences. They cannot contain literal newlines but can contain escaped newlines.
- + used for string concatenation.
- *Rich set of methods.*

String Examples

```
> const str = 'hello\n\tworld'
undefined
> console.log(str)
hello
        world
undefined
> const text = "echo a \\\n\t b c"
undefined
> console.log(text)
echo a \
         b c
undefined
> str + text
'hello\n\tworldecho a \\\n\t b c'
```

Template String Literals

- Enclosed within backquotes.
- Can contain literal newlines.
- Can interpolate arbitrary expressions enclosed within `${...}`.
- Extremely powerful.

```
> `The value of PI is ${Math.PI}\n`
'The value of PI is 3.141592653589793\n'
> `Mult-line
... string`
'Mult-line\nstring'
```

Variables and Declarations

- It is possible to use a variable without declaring it. **Do not do so**; this is a bad idea and creates a "global" variable. Modern JS environments disallow this.
- Do not use legacy **var** declarations; surprising semantics.
- Modern code declares block-scoped variables using **const** or **let**. Less surprising semantics.
- No types in declarations. Variables do not have types, values have types.
- Always try to initialize variable in declaration. When possible, prefer **const** to **let**.

Variables and Declarations: Examples

```
> let a = 22;
undefined
> let a;
... Identifier 'a' has already been declared
> const b;   //doesn't make sense without initializer
... Missing initializer in const declaration
> const b = 22;
undefined
> const [c, d, e] = [88, 22, 15/3]; //destructuring
undefined
>
```

Arrays

Array literals are comma-separated expressions within square brackets; last item can optionally be followed by a comma.

```
> const arr = [1, 2, 'hello'.length, ];
undefined
> arr
[ 1, 2, 5 ]
> arr[1]
2
> arr.length
3
//arrays are objects and can have properties
> arr._name = 'arr'
'arr'
> arr
[ 1, 2, 5, _name: 'arr' ]
```

Array Methods

```
> const arr = [1, 2, 'hello'.length, ];
undefined
> arr[1]
2
> arr
[ 1, 2, 5 ]
> arr.length
3
> arr.slice(-2)
[ 2, 5 ]
> arr.indexOf(5)
2
> arr.indexOf(6)
-1
> arr.push(7, 9)
5
```

Array Methods Continued

```
> arr.shift()
1
> arr
[ 2, 5, 7, 9 ]
> arr.pop()
9
> arr
[ 2, 5, 7 ]
> arr.unshift(22); arr.push(-1)
5
> arr
[ 22, 2, 5, 7, -1 ]
> arr.sort()
[ -1, 2, 22, 5, 7 ]
> arr.sort((a, b) => a-b)
[ -1, 2, 5, 7, 22 ]
```

Objects

Objects are maps of `String` properties to arbitrary values, with OO pixie dust sprinkled in. Very powerful { }-literal notation:

```
> let obj = { a: 22, b: 33, c: 44 }
undefined
> obj.c
44
> obj['c']
44
> const c = 88
undefined
> obj = { c, 'hello world': c }
{ c: 88, 'hello world': 88 }
> obj = { c, 'hello world': c, ['hello'.length]: obj}
{ '5': { c: 88, 'hello world': 88 }, c: 88, 'hello
world': 88 }
>
```

Object Methods

MDN

```
> let obj = Object.assign({}, {a: 22, b: 33}, {a: 44,
c: 99})
undefined
> obj
{ a: 44, b: 33, c: 99 }
> Object.keys(obj)
[ 'a', 'b', 'c' ]
> Object.values(obj)
[ 44, 33, 99 ]
> Object.entries(obj)
[ [ 'a', 44 ], [ 'b', 33 ], [ 'c', 99 ] ]
> Object.fromEntries(Object.entries(obj).slice(1))
{ b: 33, c: 99 }
>
```

Regular Expressions

Invaluable tool for string matching, JS supports regex literals enclosed within /.../.

```
> const [var1, var2] = [3, 5]
> let str = `var1*var2 = ${var1*var2}`
> str
'var1*var2 = 15'
> str.match(/=/)        //first match for '='
[ '=', index: 10, ... ] //at index 10
> str.match(/ /)        //look for space
[ ' ', index: 9, ... ]  //at index 9
> str.match(/\s/)       //specify using special regex
[ ' ', index: 9, ... ]
> str.match(/\w/)       //word char: [a-zA-Z0-9_]
[ 'v', index: 0, ... ]
> str.match(/[A-Z]/)    //char class: match one char in class
null
```

Regular Expressions Continued

```
> str.match(/[a-z]/)    //match lowercase letters
[ 'v', index: 0, ... ]
> str.match(/[a-z]+/) //+ means one-or-more
[ 'var', index: 0, ... ]
> str.match(/\w+/)   //one-or-more word chars
[ 'var1', index: 0, ... ]
> str.match(/\d+/)   //one-or-more digits
[ '1', index: 3, ... ]
> m = str.match(/(\w+)\W+(\w+)/) //capturing paren
[ 'var1*var2','var1', 'var2', index: 0, ... ]
> [m[0], m[1], m[2]]   //m[1], m[2] captured text
[ 'var1*var2', 'var1', 'var2' ]
```

Control Statements

- C-style **if-else**, **for**, **while**, **do-while**, **switch-case**.
- For iterating through an array always use **for-of**.

```
const arr = [33, 53, 36];
> for (const a of arr) { console.log(a); }
33
53
36
> for (const [i, a] of arr.entries()) {
    console.log(a*i);
  }
0
53
72
```

Functions

- Traditional functions. Example factorial:

```
> function fact(n) {
    return (n < 1) ? 1 : n*fact(n-1);
  }
> fact(4)
24
```

- Anonymous functions stored in a variable:

```
const add = function (a, b) { return a+b; }
> add(3, 6)
9
```

- Fat arrow anonymous functions:

```
> const mult = (a, b) => a * b;
> mult(6, 7)
42
```

Variable Args

If last formal parameter is preceeded by ..., then all remaining arguments are collected into that **rest parameter** as an array.

```
> function polyEval(x, ...coeffs) {
    let pow = 1;
    let sum = 0;
    for (const coeff of coeffs) {
      sum += pow*coeff;
      pow *= x;
    }
    return sum;
  }
> polyEval(2, 1, 2, 3, 4)
49
> 1*2**0 + 2*2**1 + 3*2**2 + 4*2**3
49
```

First-Class Functions

Functions can be treated like any other value:

- Can be stored in data-structures.

```
let fns = [ add, mult, ];
> [ fns[0](2, 3), fns[1](5, 6) ]
[ 5, 30 ]
```

- Can be passed and returned from functions.

```
> const fn =
    (cond, fn1, fn2) => cond ? fn1 : fn2;
> fn(false, add, mult)(3, 6)
18
> fn(true, add, mult)(3, 6)
9
```

Classes

JS does not really have classes in the sense of other OO languages. Syntactic sugar introduced relatively recently.

```
> class Point2 {
    constructor(x, y) {
      Object.assign(this, {x, y});
    }
    dist0() { //method
      return Math.sqrt(this.x**2 + this.y**2);
    }
  }
> const p = new Point2(3, 4)
> p
Point2 { x: 3, y: 4 }
> p.dist0()
5
```

First Class Classes

Classes are merely syntactic sugar for functions; hence they too are first-class.

```
> const obj = {
    point: Point2,
    name: class {
      constructor(first, last) {
        this.first = first; this.last = last;
      }
      fullName() {
        return `${this.first} ${this.last}`;
      }
    }
  }
```

First Class Classes Continued

```
> const p1 = new obj.point(12, 5)
> p1
Point2 { x: 12, y: 5 }
> p1.dist0()
13
> const n = new obj.name('bill', 'smith');
> n
name { first: 'bill', last: 'smith' }
> n.fullName()
'bill smith'
```

Dynamic Language

```
> p.label = 'first'; p1.label = 'second';
'second'
> p
Point2 { x: 3, y: 4, label: 'first' }
> p1
Point2 { x: 12, y: 5, label: 'second' }
> delete p1.label
true
> p1
Point2 { x: 12, y: 5 }
> Point2.descr = '2D Point'
'2D Point'
> Point2
[class Point2] { descr: '2D Point' }
>
```

JavaScript Literals

JavaScript has very rich notation for *literal data*:
A primitive JS literal value is one of:

- An integer (including `0xAf` and octal `072` literals), floating point literal like `1.23e-2` or `1E4`.
- A string literal delimited by single or double-quotes or by backquotes for template literals.
- Anonymous functions using either the **function** keyword, or fat-arrow.
- Anonymous classes.

JavaScript Literals Continued

A compound JS literal value is one of:

- Arrays of comma-separated literal data enclosed within `[ ]`.
- Object literals consisting of comma-separated `key: value` pairs enclosed within `{ }`.
  - Keys can be specified as strings or simple identifiers.
  - Dynamic key values can be specified within `[ ]`.
  - If a key-value is simply the name of a variable `var`, then it is equivalent to the key-value `var: var`.
  - If a key-value looks like `id(...) { ... }`, then it is equivalent to `id: function (...) { ... }`.

Both object and array literals permit an optional trailing `,` after the last item.

Complex JavaScript Literal Example

```
const [ n1, n2 ] = [ 42, 33 ];
> const [ str1, str2 ] = [ 'hello', 'world' ];
> { n1: n1,
    n2,
    [`${str1}_${str2}`]: (a, b) => a+b,
    mult(x, y) { return x*y; },
    arr: [ n1, n2, ],
  }
{
  n1: 42,
  n2: 33,
  hello_world: [Function: hello_world],
  mult: [Function: mult],
  arr: [ 42, 33 ]
}
>
```

JavaScript Object Notation: Very simple specification. A JSON literal can be one of:

- A primitive literal can be a number, string within double quotes, or **true**, **false** or **null**.
- An array literal consists of JSON literal values separated by comma's. An optional trailing comma is not allowed.
- An object literal consists of `key: value` pairs separated by comma's. An optional trailing comma is not allowed. The `key` **must** be a string within double-quotes.

JSON Example

```
{
  "john": {
     "name": "John Cassidy",
     "age": 42,
     "kids": [ "jane", "bill" ]
   },
  "mary": {
     "name": "Mary Jones",
     "age": 42,
     "kids": [ "lucy", "sue" ]
   }
}
```

JSON Evaluation

- Can be easily parsed by standard JS library using JSON.parse().
- A JS object can easily be converted to a JSON string using JSON.stringify(). Usually, JS objects without a JSON representation are silently ignored or represented as a **null** when within an array.
- Excellent data format for exchanging structured data between heterogeneous systems.
- Restricted format.
- Comments are not allowed. Makes it a bad choice as a configuration format (unfortunately, many JS tools use it as a configuration format).

Gotchas

- **return** with return-expression on the next line will result in **undefined** being returned. If inconvenient to begin return-expression on same line as **return** keyword, use:

```
return (
  longExpr
);
```

- Never use == and != for checking equality. Surprising type converions. Always use === and !==; no conversions.

```
> null == undefined
true
> null === undefined
false
> '' == 0
true
> '' === 0
false
```

Gotcha's from C Legacy

- Need **break** after **case** to avoid fall-thru.

```
switch (type) {
  case 'number':
    x = 1;
    //need a break statement here
  case 'string';
    x = 42;
    break;
}
```

- Integer literals starting with leading 0 are treated as octal:

```
> 010
8
```