Overview

- Expressions
- A glimpse at default conversions.
- Declarations
- Statements including conditionals and repetition
- More objects
- Exceptions
- Automatic semi-colon insertion.

Arithmetic Operators

Recall that JavaScript only has 64-bit floating point numbers, no integers.

- Usual arithmetic operators +, -, * and /.
- Note that + converts numbers to strings if either operand is a string; other operators convert strings to numbers if necessary.
- ** is used for **exponentiation**.
- % operator is **remainder**, not modulus; no difference for positive operands. Result of % always has sign of numerator.

```
> function isOdd(n) { return n%2 === 1; } //WRONG!
undefined
> isOdd(-7)
false
> function isOdd(n) { return n%2 !== 0; }
undefined
> isOdd(-7)
true
```

Unary Arithmetic Operands

- ++ and -- are post-increment / post-decrement if used after the operand. Returns value of operand **before** increment/decrement.
- ++ and -- are pre-increment / pre-decrement if used before the operand. Returns value of operand **after** increment/decrement.
- Prefix - is used for negating a number.
- Prefix + often used to convert string to a number (better to use an explicit `Number()` conversion).

Unary Arithmetic Operands Examples

```
> x = 42
42
> x++
42
> x
43
> --x
42
> x
42
> "4" +  1
'41'
> +"4" +  1          //unary + converts to number
5
> Number("4") +  1 //more explicit
5
>
```

Bitwise Operators

Operands are treated as a 32-bit words.

- `&`: bitwise-and; `|`: bitwise-or, `^`: bitwise-xor; `~`: bitwise not.
- `<<` left-shift. Can be used for multiplying by powers of 2.
- `>>`: arithmetic right shift. Propagates sign. Can be used for dividing by powers of 2.
- `>>>`: zero-fill right shift; shifts in 0's on left.
- Can be used for implementing flags.
- Can also be sets of small non-negative integers. For example, the set $\{0, 2, 5\}$ can be represented using the number `((1<<0) | (1 << 2) | (1 << 5))` $\equiv$ `(0x1 | 0x4 | 0x20)` $\equiv$ `0x25` $\equiv 37$.

Bitwise Operators Examples

```
> 0xf & 3
3
> ~(0xf & 3).toString(16)
-4
> 3 << 4
48
> -3 << 4
-48
> 1024 >> 4
64
> -1024 >> 4
-64
> -1024 >>> 4
268435392
>
```

Logical Operators

- Logical and: `&&`. Short-circuit evaluation. `false && _` is false without evaluating _.
- Logical or: `||`. Short-circuit evaluation. `true || _` is true without evaluating _.
- Treats *truthy* values as `true` and *falsy* values as `false`.
- `&&` and `||` return truthy/falsy values (not necessarily `true` or `false`).
- Logical not: `!`; always returns `true` or `false`.

```
> 'cat' && 'dog'
'dog'
> 'cat' || 'dog'
'cat'
> '' && 'cat'
''
> !('' && 'cat')
true
```

Named Operators

- `typeof` returns primitive type or `'object'`.
- *X* `instanceof` *Y* returns `true` if *X* is an instance-of *Y*.
- *x* `in` *y* returns true if *x* is a property of object *y*.
- `delete obj[property]` deletes `property` in object `obj`.
- `delete array[index]` deletes `index` in array `array`.
- `void` used to evaluate an expression without returning a value. Often used to indicate a NOP hyper-link.

  `<a href="javascript:void(0)">NOP</a>.`

Named Operators Examples

```
> let a = []
undefined
> typeof a
'object'
> a[999] = 1
1
> 999 in a
true
> a.length
1000
> delete a[999]
true
> a.length
1000
> 999 in a
false
```

Miscellaneous Operators

- `cond ? thenExp : elseExp`. Nesting this is often frowned on.
- `,` operator; often used for declaring multiple variables.

```
for (let i = 0, j = n; i < j; i++, j--) { ... }
```

- Relational operators <, <=, >, >=. Can be used with both numbers and strings (lexicographical order).
- Equality operators ==, != and === and !==. == and != do conversions; prefer to use === and !== which do not do conversions.
- Many levels of operator precedence. Best to parenthesize except for very common cases.

Assignment Expressions

- Basic assignment operator is =.
- Can be chained `a = b = 2;`; right-associative, i.e. is equivalent to `a = (b = 2);`. Returns RHS expression.
- Can be combined with a binary operator $\oplus=$ for $\oplus$ one of the arithmetic operators +, -, *, /, % or the bitwise operators <<, >>, >>>, &, | and ^. $a \oplus= \text{exp}$ is like $a = a \oplus \text{exp}$ except that a is evaluated only once.
- Has low precedence; can be used with other right-associative operators like => which has higher precedence:

```
> g = f = a => b => a*b //function which returns function
[Function]
> f1 = f(2); g1 = g(3); //f1 = doubler; g1 = tripler.
[Function]
> f1(5)
10
> g1(5)
15
```

Automatic Conversions within Expressions

- Very complex conversion rules; best to avoid in new code, but need to handle legacy code.
- Operators where conversions occur include + (both prefix and infix), - (both prefix and infix), other arith/relational ops.
- + is used for both strings (concatenation) and numbers (addition). If either operand is a string then we are doing concatenation.

Some Simple Conversions

```
> 1 + '2'
'12'
> '2' * 3
6
> false * 6
0
> null + 5
5
> undefined * 4
NaN
> true * '5'
5
>
```

More Conversion Examples

```
> 1 + 2 + "3" + 4   //left-assoc +: ((1 + 2) + "3") + 4
'334'
> a = '1'
'1'
> a = a + 3 + 6   //concat "3" + "6"
'136'
> a += 3 + 6      //numeric add 3 + 6
'1369'
>
```

A Glimpse at Conversion Rules

Arithmetic and concatenation expressions are evaluated using primitive operands. Specifically, if we are looking for a primitive operand as a `Number`:

1. If operand is primitive, then nothing needs to be done.
2. If operand is an object `obj` and `obj.valueOf()` returns a primitive object, then return that primitive object.
3. If operand is an object `obj` and `obj.toString()` returns a primitive object, then return that primitive object.
4. Otherwise throw a `typeerror`.

If we are looking for a primitive operand as a `String`, then interchange steps 2 and 3.

Object Conversion Examples

```
> x = { toString: function() { return "5"; },
...       valueOf: function() { return 2; } }
{ [Number: 2] toString: [Function: toString],
  valueOf: [Function: valueOf] }
> x + 3
5
> x + '3' //+ calls valueOf() first for both operands
'23'
> String(x)
'5'
>
```

Equality

- js has both == and === operators along with corresponding `!=` and `!==` operators.
- **Loose equality** operator == tries to convert its operands to the same type before comparison.
- **Strict equality** operator === does not do type conversion; simply returns `false` if types are different.
- Almost always use === and `!==`; do **not** use == or `!=`.
- Do a google search on `js wtf`.

Equality Examples

```
> '1' == 1
true
> '1' === 1
false
> undefined == null
true
> undefined === null
false
> '' == 0
true
> 0 == '0'
true
> '' == '0'
false                    //breaks transitivity
>
```

Object Equality Examples

For both == and !==, objects are equal only if they have the same reference.

```
> {} == {}
false
> {} === {}
false
> x = {}
{}
> y = x
{}
> x == y
true
> x === y
true
>
```

Declarations

- My order of descending preference: `const`, `let`, `var`.
- Convention is to use all uppercase names for manifest constants.
- Many JS programs have multiple declarations using a single specifier:

```
let var1 = value1,
    var2 = value2,
    ...
    varN = valueN;
```

  I consider that error prone and would prefer that each declaration stand alone using its own `let` specifier.
- If you assign to an undeclared variable, then that variable will be created as a property of the global object. Force error by always specifying `use strict`.

Statements

- Any expression is a statement; so usually an assignment or function call is used as a statement. However, JavaScript will also accept `1+2;` as a statement.
- Conditional statements: `if-then` statement `if (`*cond*`)` *statement*`;` `if-then-else` statement `if (`*cond*`)` *statement* `else` *statement*`;`
- Switch statement `switch (`*expr*`) { case` *value1*`: ... }`. Uses strict === equality for matching *expr* with *value1*, ... **Must** end case with `break`, otherwise control falls-through to next `case`. No default case can also cause entire statement to be skipped. Notorious for introducing bugs.

Repetition Using while

- `do` *statement* `while ( condition ).` *statement* is executed at least once.
- `while ( condition )` *statement*

Prefer `do`-‘while‘ if possible because it documents the fact that the loop is to be executed at least once.

Repetition using for

`for ( init ; condition ; updates ) statement`

init Initializations; usually assignments or declarations. Can be omitted by simply having `;`.

condition Evaluated before each loop interation; *statement* executed if *condition* is truthy. Can be omitted by simply having `;`.

updates Expression to be evaluated after each loop iteration before evaluation of *condition*.

Often used for indexing through a range of integers:

```
const n = ...
const step = ...
for (let i = 0; i < n; i++) { ... }
for (let i = 0; i < n; i += step) { ... }
```

Property Attributes

```
> a = { x: 22 }
{ x: 22 }
> Object.getOwnPropertyDescriptors(a)
{ x:
   { value: 22,
     writable: true,
     enumerable: true,          //loop for...in or forEach
     configurable: true } }     //change descr; delete
> Object.defineProperty(a, 'y', {})
{ x: 22 }
```

Property Attributes Continued

```
> Object.getOwnPropertyDescriptors(a)
{ x:
   { value: 22,
     writable: true,
     enumerable: true,
     configurable: true },
  y:
   { value: undefined,
     writable: false,
     enumerable: false,
     configurable: false } }
```

Property Attributes Continued

```
> delete(a['x'])
true
> Object.getOwnPropertyDescriptors(a)
{ y:
   { value: undefined,
     writable: false,
     enumerable: false,
     configurable: false } }
> delete(a['y'])
false
> Object.getOwnPropertyDescriptors(a)
{ y:
   { value: undefined,
     writable: false,
     enumerable: false,
     configurable: false } }
>
```

Property Getter

```
> obj = { get len() { return this.value.length; } }
{ len: [Getter] }
> obj.value = [1, 2]
[ 1, 2 ]
> obj.len
2
> obj.value = [1, 2, 3]
[ 1, 2, 3 ]
> obj.len
3
```

Property Setter

Use property `x` as proxy for property `_x` while counting # of changes to property `x`.

```
> obj = { nChanges: 0,
... get x() { return this._x; },
... set x(v) {
..... if (v !== this._x) this.nChanges++;
..... this._x = v;
..... }
... }
```

Property Setter Continued

```
> obj.x
undefined
> obj.x = 22
22
> obj.nChanges
1
> obj.x = 42
42
> obj.nChanges
2
> obj.x = 42
42
> obj.nChanges
2
>
```

Enumerating Object Properties using for-in

```
for (let v in object) { ... }
```

- Sets v to successive **enumerable** properties in object **including inherited properties**.
- No guarantee on ordering of properties; specifically, no guarantee that it will go over array indexes in order. Better to use plain `for` or `for-of`.
- Will loop over enumerable properties defined within the object as well as those inherited through the prototype chain.
- If we want to iterate only over local properties, use `getOwnPropertyNames()` or `hasOwnProperty()` to filter.

Enumerating Example

```
> a = { x: 1 }
{ x: 1 }
> b = Object.create(a) //a is b's prototype
{}
> b.y = 2
2
> for (let k in b) { console.log(k); }
y
x
undefined
> for (let k in b) {
... if (b.hasOwnProperty(k)) console.log(k);
... }
y
undefined
```

Enumerating Example Continued

```
> names = Object.getOwnPropertyNames(b)
[ 'y' ]
> for (let k in names) { console.log(k); }
0
undefined
for (let k in names) { console.log(names[k]); }
y
undefined
>
```

Another Enumerating Example

```
> x = {a : 1, b: 2 }
{ a: 1, b: 2 }
> Object.defineProperty(x, 'c',
                        { value: 3}) //not enumerable
{ a: 1, b: 2 }
> x.c
3
> for (let k in x) { console.log(k); }
a
b
undefined
> x.c
3
>
```

Iterating using for-of

Values contained in `Iterable` objects can be iterated over using `for-of` loops.

```
for (let var of iterable) { ... }
```

Builtin iterables include `String`, `Array`, ES6 `Map`, `arguments`, but **not Object**.

```
> for (const x of 'abc') { console.log(x); }
a
b
c
undefined
>
```

An Aside for JavaScript Symbols

- Many languages have internal *symbols* which are guaranteed to be unique. For example, Lisp has *atoms*, Java, Python have intern'd strings, Ruby has symbols.
- JavaScript has a `Symbol` primitive.
- No literal representation.
- Created using the `Symbol()` function; **not a constructor**.
- Can provide a description string for debugging.
- Usually used to provide unique property names for Objects.

Symbol API Examples

```
> Symbol()
Symbol()
> new Symbol()
TypeError: Symbol is not a constructor
...
> Symbol('hello')
Symbol(hello)
> Symbol('hello') == Symbol('hello')
false
> Symbol('hello').toString()
'Symbol(hello)'
>
```

Symbol API Examples Continued

```
> s = Symbol.for('hello') //create in global symbol registry
Symbol(hello)
> t = Symbol.for('hello') //return previous value if present
Symbol(hello)
> s === t
true
> Symbol.keyFor(s)
'hello'
> Symbol.iterator   //built-in symbol
Symbol(Symbol.iterator)
>
```

Defining Iterable Objects

- An object is *iterable* (using `for-of`) if it has a `Symbol.iterator` property which holds a no-argument function which returns a *iterator* object.
- An object is a iterator if it has a `next()` function which returns a object containing at least one of the following properties:

  `done` If true, then the iterator is done. If `value` is defined, then it is the return value of the iterator.

  `value` The next value in the iterator sequence.

A Sequence Iterable

In iterable-seq.js:

```
#!/usr/bin/env nodejs

'use strict';

/** Produce seq from inclusive bounds lo to hi */
function seq(lo=0, hi=Number.POSITIVE_INFINITY) {
  let i = Math.floor(lo);
  return {
    [Symbol.iterator]: () => ({
      next: () =>
        (i <= hi) ? { value: i++ } : { done: true }
    })
  };
}
```

A Sequence Iterable Continued

```
const LO1 = 2, HI1 = 5;
console.log(`seq(${LO1}, ${HI1})...`);
for (const i of seq(LO1, HI1)) { console.log(i); }

const LO2 = -2.2, N = 4;
console.log(`first ${N} of seq(${LO2})...`);
let n = 0;
for (const i of seq(LO2)) {
  console.log(i);
  n++;
  if (n >= N) break;
}
```

Sequence Iterable: Log

```
$ ./iterable-seq.js
seq(2, 5)...
2
3
4
5
first 4 of seq(-2.2)...
-3
-2
-1
0
$
```

A Glimpse at Generators

Generators defined using `function*` and `yield`.

```
> function* seq(lo=0, hi=Number.POSITIVE_INFINITY) {
    for (let i = Math.floor(lo); i <= hi; i++) yield(i);
  }
undefined
> for (s of seq(1, 3)) console.log(s);
1
2
3
undefined
>
```

Exceptions

```
> a = new Array(-1)
RangeError: Invalid array length
    at repl:1:5
    ...
> try { let a = new Array(-1); }
  catch (ex) {  } //BAD CODE!
undefined
> try { let a = new Array(-1); }
  catch (ex) { console.log(ex.message); }
Invalid array length
undefined
```

Exceptions Continued

Allows a `finally` which can be used to clean up resources. Usual pattern:

```
f = openFile(); //create resource
try {
   //process file
}
finally {
   f.close();   //clean-up resource
}
```

Exceptions Continued

- Any object can be `throw`'n as an exception.
- Runtime exceptions use `Error` as prototype.
- Runtime exceptions include `EvalError`, `RangeError`, `SyntaxError`, `TypeError`, etc.

```
> try { throw { msg: 'thrown' } }
  catch (ex) { console.log(ex.msg); }
thrown
undefined
>
```

Semicolon Insertion

Automatic Semicolon Insertion (ASI):

- Insert semicolon at newline if that fixes syntax error.
- Always insert semicolon after `return`, `break`, `continue` when followed by a newline.
- Always insert semicolon if next line starts with ++ or --.

```
> function f() {
    return 5
      + 3
  }
undefined
> f()
8
```

Semicolon Insertion Continued

Can cause problems:

```
> function f() { //silently returns undefined
    return
     {          //start of unreachable code block
       a:       //label!
       false    //expression statement
     }
  }
undefined
> f()
undefined
```