JavaScript

Thilan Tran

Summer 2020

Contents

JavaScript 3

Language Overview 4
- Comparisons . . . 5
- Scope and Closures . . . 6
- this Identifier and Prototypes . . . 8
- Backwards Compatibility . . . 9

Types 11
- Arrays . . . 11
- Strings . . . 12
- Numbers . . . 13
- Special Values . . . 13
- Value vs. Reference . . . 14
- Natives . . . 15

Coercion 17
- Abstract Value Operations . . . 17
- Explicit Coercion . . . 18
- Implicit Coercion . . . 19
- Equality . . . 21

Grammar 24

Scope 26
- Lexical and Dynamic Scope . . . 27
  - Lexical Scope with Arrow Functions . . . 29
- Function Scope . . . 30
- Block Scope . . . 31

Hoisting 32

Closures **36**
Modules 37

this and Binding Rules **39**
Binding Rules 40
Precedence 43

Objects **45**
Duplicating Objects 45
Properties 46
Object Prototypes 49

Object-Oriented Design **51**
Mixins 51
Using Prototypes 53
Prototypal Inheritance 54
Using Create 56
Delegation-Oriented Design 56
Introspection 58
ES6 Classes 60

Asynchronous JavaScript **62**
Callbacks 63
Promises 64
Generators 68
Async/Await 71

Using the DOM **73**
Data Types 73

ES6 Features **75**
New Syntax 75
Organization 78
Collections 81
API Additions 82

JavaScript

- **JavaScript (JS)** is a programming language that is one of the *core technologies* on the Internet (alongside HTML and CSS):
  - JS is used for handling client-side behavior of web applications, and allows for *interactive* web pages
  - as a programming language:
    - JS is a multi-paradigm language that is imperative, functional, and event-driven:
    - JS features a C-style syntax, dynamic typing, first-class functions, and prototype-based object-orientation (rather than class-based)
    - JS uses just-in-time compilation (like Java)
  - JavaScript and Java are distinct, similar only in name and syntax
  - JS is specified by the ECMAScript (ES) specification
- note that there are key JavaScript features that are *not* JavaScript *language* features:
  - eg. JavaScript is written to run and interact with browsers, using primarly the DOM API:
    - eg. `var el = document.getElementById("foo")`
    - DOM API is not controlled by the JS specification or provided by the JS engine
    - `getElementById` is a built-in method provided by the DOM from the *browser*, which may be implemented in JS or traditionally C/C++
  - eg. input and output:
    - eg. `alert` and `console.log` are again provided by the browser, not the JS engine itself

Language Overview

- primitive builtin types:
  - `string` eg. `"hello world"`
    - single vs. double-quotes are a purely stylistic distinction
  - `number` eg. `42`
  - `boolean` eg. `true` and `false`
  - `null` and `undefined`
    - the undefined value is behaviorally no different from an uninitialized variable
  - `object` eg. in literal form `var obj = { foo: "bar" }`, or in constructed form `var obj = new Object()` and `obj.foo = "bar"`
    - literal and constructed form result in exactly the same sort of object
    - an object is a compound value with **properties** ie. named locations
    - properties accessed through dot-notation `obj.foo` or bracket-notation `obj["foo"]`
- conversion between types is done through explicit and implicit **coercion**:
  - with *explicit* coercion, the type cast is explicitly specified in the code eg. `var a = Number("42")`
  - with *implicit* coercion, the type cast occurs as a non-obvious side effect of some other operation eg. `var a = "42" * 1` coerces a string to a number implicitly
    - note that arrays are default coerced to strings by joining the values with `,` in between
      - eg. `[1,2,3] == "1,2,3"` is true
    - while objects are default coerced to the string `"[object Object]"`
- **wrapper** objects:
  - wrapper objects ie. *"natives"* pair with their corresponding primitive type to define useful builtin functions
  - eg. the string in `"hello".toUpperCase()` is *automatically* wrapped or *"boxed"* into the `String` object that supports various useful string operations
  - other wrapper objects include `Number` and `Boolean`
- arrays and functions are *specialized* object subtypes:
  - arrays are objects that hold values of any type in numerically indexed positions:
    - eg. `var arr = ["hello world", 42, true]`
    - `arr[0]` gives `"hello world"`, `arr.length` gives 3
  - functions are also an object subtype:
    - note however that `typeof func` gives `"function"` not `"object"`

        * as objects, functions can also have properties
        * as first-class values, functions are *values* that can be assigned to variables:
            * JS has *anonymous* function expressions and *named* function expressions
            * eg. `var foo = function() {}` vs. `var foo = function bar() {}`
        * an **immediately invoked function expression (IIFE)** is another way to execute a function expression immediately:
            * eg. `var x = (function foo() { console.log(42); return 1; })()` immediately prints 42 and assigns 1 to x
            * the first outer `()` prevents the expression from being treated as a normal function declaration
            * the next `()` immeidately executes the function
            * often used to declare variables that do not affect the surrounding code
            * the declared function is not accessible outside of the IIFE
- identifiers in JS are `[a-zA-Z$_][a-zA-Z$_0-9]*`
    - nontraditional character sets such as Unicode are also supported
    - execpting *reserved* words such as `for, in, if, null, true, false`

Comparisons

- *"truthy"* and *"falsy"* values are automatically coerced to their corresponding boolean values by JS:
    - the complete list of JS falsy values are:
        * `""` , `0, -0, NaN` , `null, undefined` , and `false`
    - anything that is not falsy is truthy:
        * note that *empty* arrays and objects coerce to true, as well as functions
- there are four equality operators in JS:
    - `==, ===, !=, !==`
    - double equals checks for value equality with coercion *allowed*
        * eg. `"42" == 42` is true
        * note that `null` is a special case that is equal to `null` or `undefined` only
            * eg. `null == ""` is false
    - while triple equals or *"strict equality"* checks for value equality *without* allowing coercion
        * ie. checking both value and type equality
        * eg. `"42" === 42` is false
    - for non-primitive values like *objects*:

    - `==` *and* `===` check if the *references* match, rather than the underlying values
    - eg. `[1,2,3] == [1,2,3]` is false
- there are four *relational* comparison operators in JS:
  - `<, >, <=, >=`
  - usually used with numbers, as well as strings
  - there are no *strict* comparison operators
  - like equality, coercion rules apply:
    - eg. `41 < "42"` is true
    - eg. `"42" < "43"` is true lexicographically
    - eg. `42 < "foo"` and `42 > "foo"` are *both* false since `"foo"` is coerced to `NaN` , which is neither greater nor less than any other value
      - note that `NaN` does not equal anything, even itself
    - eg. `42 == "foo"` is false

Example comparison coercions:

```
true + false // 1 + 0    -> 1
[1] > null   // "1" > 0 -> 1 > 0 -> true

"foo" + + "bar" // "foo" + (+"bar") -> "foo" + NaN -> "fooNaN"
[] + null + 1   // "" + null + 1     -> "null" + 1   -> "null1"

{} + [] + {} + [1] // +[] + {} + [1] -> 0 + "[object Object]" + [1] ->
                   // "0[object Object]1"
! + [] + [] + ![]  // (!+[]) + [] + (![]) -> !0 + [] + false ->
                   // true + "" + false    -> "truefalse"
```

Scope and Closures

- the `var` keyword declares a variable belonging to the current function scope, or the global scope if at the top level:
  - JS also uses **nested scoping**, where when a variable is declared, it is also available in any lower ie. inner scopes
    - inner scopes ie. nested functions
  - without `var` , the variable is *implicitly* auto-global declared
    - can use **strict mode** with the `"use strict";` declaration, which throws errors such as disallowing auto-global variables
  - in ES6, **block scoping** can be achieved instead of function scoping using the `let` declaration keyword
    - allows for a finer granularity of variable scoping

- in JavaScript, whenever `var` appears *inside* a scope, that declaration is *automatically* taken to belong to the *entire scope*:
  - this behavior is called **hoisting** ie. a variable declaration is conceptually *moved* to the top of its enclosing scope
  - variable hoisting is usually avoided, but function hoisting is a more commonly used practice

Illustrating hoisting:

```
var a = 2;
foo(); // foo declaration is *hoisted*

function foo() {
  a = 3; // a declaration is *hoisted*
  console.log(a); // 3
  var a;
}

console.log(a); // 2
```

- **closures** are a way to *remember* and continue accessing the variables in a function's scope even once the function has finished running:
  - an essential part of **currying** in functional programming languages
  - closures are also commonly used in the **module** pattern
    - allows for defining private implementation details, with a public API

Closure example:

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

var plusOne = makeAdder(1);  // returns ref to inner add that has bound x to 1
var plusTen = makeAdder(10); // returns ref to inner add that has bound x to 10

plusOne(41); // gives 42
plusTen(41); // gives 51
```

Module example:

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

```
  function doLogin(user, pw) {
    username = user;
    password = pw;
    ...
  }
  var publicApi = { login: doLogin };
  return publicApi;
}

var bob = User(); // not new User, User is a function
bob.login("bob", "1234"); // binds variables from the *instantiation* of User
                          // even though User function itself has returned
```

this Identifier and Prototypes

- the `this` keyword in a function points to an object:
  - which object it points to depends on *how* the function was called
    - *dynamically* bound
  - `this` *does not* refer to the function itself
  - not *exactly* an object-oriented mechanism

`this` example:

```
function foo() {
  console.log(this.bar);
}

var bar = "global";
var obj1 = {
  bar: "obj1",
  foo : foo
};
var obj2 = {
  bar: "obj2"
};

foo();          // "global", this set to global object in non-strict mode
obj1.foo();     // "obj1", this set to obj1
foo.call(obj2); // "obj2", this set to obj2
new foo();      // undefined, this set to brand new empty object
```

- the **prototype** mechanism in JavaScript allows JS to use an object's internal

prototype reference to find another object to look for a missing property:
  - ie. a *fallback* when an accessed property is missing
  - the internal prototype reference linkage occurs when the object is created
  - could be used to emulate a fake class mechanism with inheritance, but more naturally is used for the delegation design pattern

Prototype example:

```
var foo = { a: 42 };
var bar = Object.create(foo); // creates bar and links it to foo
bar.b = "hello";

bar.b; // "hello"
bar.a; // 42, delegated to foo
```

Backwards Compatibility

- JavaScript as a language has been constantly evolving:
  - ECMAScript specifications change, currently on ES6
  - older browsers do not fully support ES6 JS
  - two methods to achieve backwards compatibility with older versions, polyfilling and transpiling
- a **polyfill** takes the definition of a newer feature and produces a piece of code that is equivalent behavior-wise, but is still able to run on older JS environments:
  - note that some features are not fully polyfillable
  - different polyfill libraries available for ES6, eg. `ES6-Shim`

Example polyfill for `Number.isNaN` for ES6:

```
if (!Number.isNaN) {
  Number.isNaN = function isNaN(x) {
    return x !== x; // NaN not equal to itself
  };
}
```

- **transpiling** converts newer code into older code equivalents:
  - there is no way to polyfill new syntax added in new ES versions
    - source code with new syntax must instead be *transpiled* into an old syntax form
  - transpiler is inserted into the build process, like the code linter or minifier

  - eg. Babel and Traceur transpile ES6+ into ES5

Transpiling ES6 default parameter values:

```
// in ES6:
function foo(a = 2) { console.log(a); }

// transpiled:
function foo() {
  var a = arguments[0] !== (void 0) ? arguments[0] : 2;
  console.log(a);
}
```

Types

- there are seven builtin JavaScript types:
  - `null, undefined, boolean, number, string, object, symbol`
  - the `typeof` operator inspects the type of the given value:
    - eg. `typeof undefined === "undefined"` , `typeof 42 === "number"`
    - note that `typeof` gives `"function"` for functions, even though functions are a subtype of object
    - note the special case `typeof null === "object"`
      - to check for `null` , note that it is the only falsy value that `typeof` returns `"object"` for
- variables that have the value `undefined` have no value *currently*:
  - note that *undefined* is distinct from *undeclared*
  - an undefined variable has been declared, but *at the moment* has no value in it
  - interestingly, `typeof` on an undeclared and undefined variable gives `"undefined"` for both
    - however, `typeof` *does* fail for temporal deadzone references
  - thus to perform a global variable check:
    - `if (typeof DEBUG !== "undefined")` works, while `if (DEBUG)` throws an error if undeclared
    - alternatively, `if (window.DEBUG)` , since object property check does not throw an error

Arrays

- JavaScript arrays are containers for any type of value:
  - no need to presize arrays, arrays start at length 0 and can resize as values are added
  - arrays are numerically indexed, but as objects, they can still have string keys and properties added:
    - these properties do not count towards the `length` property of the array
    - unless the string value can be coerced to a number, in which case it will be treated as a numeric index
  - array gotchas:
    - *sparse* arrays have empty or missing slots
    - setting the length of an array *without* setting explicit values *implies* that the slots exist:

        - ie. implicitly creates empty slots, can also be done with `new Array(len)`
        - has issues with serialization in browsers, as well as certain array operations failing
        - eg. `map` will fail because there are no slots to iterate over, while `join` works because it only loops up to the `length`
    - using `delete` on an array value will remove that slot from the array, but the `length` property is *not* updated
  - to create an array from array-like objects (such as DOM queries, etc.):
    - borrow `slice` on the value eg. `Array.prototype.slice.call(arrLikeObj)`

Illustrating array nuances:

```
var a = [];
a.length; // gives 0

a[0] = 1;
a["2"] = [3];
a["foo"] = 2;

a[1];     // gives undefined
a[2];     // gives [3]
a.foo;    // gives 2
a.length; // gives 3
```

Strings

- JavaScript strings are very similar to arrays of characters:
  - both are array-like, have a `length` property, and have `indexOf` and `concat` methods
  - *however*, strings are *immutable*:
    - individual characters can be accessed but not set using array indexing or `charAt`
    - thus string methods create and return *new* strings, while array methods perform changes in-place
  - nonmutation array methods can be borrowed on strings:
    - eg. `Array.prototype.join.call` or `Array.prototype.map.call`
    - however, borrowing mutator methods such as `Array.prototype.reverse.call` will *fail* since strings are immutable
  - hack for reversing strings:
    - `str.split("").reverse().join("")`

- string methods (from the wrapper `String.prototype`):
  - none of these methods modify the string value in place
  - `indexOf, charAt`
  - `substr, substring, slice, trim`
  - `toUpperCase, toLowerCase`

Numbers

- JavaScript has just one numeric type that includes both integer and decimal numbers:
  - in JS, there are no true integers, as in other languages
  - all numbers stored in IEEE floating point
  - supports exponential form, as well as `0x 0b 0o` forms for hex, binary, and octal, respectively
  - the automatic boxing of primitive numbers into the `Number` wrapper gives access to methods:
    - `toFixed` specifies how many decimal places to represent the value
    - `toPrecision` specifies how many significant digits to represent the value
  - `Number.isInteger` tests if a value is an integer, and `Number.isSafeInteger` tests if a value is a *safe* integer
  - note that `.` will be interpreted as a numeric character before a property accessor:
    - `42.toFixed(3)` is a syntax error, while `42..toFixed(3)` is not
- the infamous side effect of floating point representation is rounding error:
  - `0.1 + 0.2 === 0.3` is false
  - small decimal values should be compared with respect to a *tolerance* value for rounding error:
    - this tolerance value is `Number.EPSILON` or 2^{-52} for JavaScript specifically
- number ranges:
  - `Number.MIN_VALUE` and `Number.MAX_VALUE` for floating point values
  - `Number.MIN_SAFE_INTEGER` and `Number.MAX_SAFE_INTEGER` for integers
- note that some numeric operations such as bitwise operators are *only* defined for 32-bit numbers:
  - to force a number value `a` to a 32-bit signed integer value, use `a | 0` as a bitwise no-op

Special Values

- nonvalue values `null` and `undefined` :
  - different ways to distinguish between `null` and `undefined` :
    - `null` is an empty value, while `undefined` is a missing value
    - `null` had a value and doesn't anymore, while `undefined` hasn't had a value yet
  - note that `undefined` is a *valid* identifier, while `null` is not
  - the `void` operator *voids* out any value so that the result of the expression is always `undefined` :
    - eg. `void 0` , `void true` , `undefined` are all identical
    - can be used to ensure an expression has no result value, even if it has side effects
- `NaN` or *"not a number"* represents invalid or *failed* numbers:
  - note that `typeof NaN === "number"`
  - `NaN` is never equal to itself
  - can check for `NaN` with `Number.isNaN`
- infinities:
  - `1 / 0 === Infinity` ie. `Number.POSITIVE_INFINITY`
  - `-1 / 0 === -Infinity` ie. `Number.NEGATIVE_INFINITY`
- zeros:
  - JavaScript has positive *and* negative zeros
  - eg. `0 / -3 === -0` and `0 * -3 === -0`
  - note that stringifying a negative zero value always gives `"0"`
    - but the reverse operations result in `-0` , eg. `+"-0" === -0`
  - in addition, note that `0 === -0`
  - can also check for `NaN` and `-0` using `Object.is(a, b)`

Value vs. Reference

- in JavaScript, there are no pointers, so references work differently from other languages:
  - the *type* of a value alone controls whether that value is assigned by value-copy or reference-copy:
    - primitives always assign by value-copy, including `null, undefined, symbol`
    - compound values like `object, array, function` and wrappers always assign by reference-copy
      - thus changes are reflected in the shared value when using either reference

Illustrating reference-copy nuances:

```
function foo(x) {
  x.push(4);
```

```
  x = [4, 5, 6]; // this assignment does *not* affect
                 // where the initial reference points
  x.push(7);
}

function bar(x) {
  x.push(4);
  x.length = 0;        // empty array in-place
  x.push(4, 5, 6, 7); // mutate array
}

var a = [1, 2, 3];
foo(a);
a; // gives [1, 2, 3, 4] and not [4, 5, 6, 7]

var b = [1, 2, 3];
bar(b);
b; // gives [4, 5, 6, 7] and not [1, 2, 3, 4]
```

- in order to pass a compound value by value-copy:
  - must manually make a copy of it, so the passed reference no longer points to the original
  - eg. `foo(a.slice())`
- in order to pass a primitive value in a way that its value updates can be seen, like a reference:
  - must wrap the value in another compound value that *can* be passed by reference-copy
  - note that we cannot simply use the primitive's wrapper class:
    * the underlying scalar primitive in a wrapper is *immutable*

Natives

- **natives** are builtins that when construct called, create an object wrapper around the primitive value:
  - eg. `String Number Boolean Array Object Function Symbol`
    * as well as `RegExp Date Error`
  - because primitive values don't have properties or methods, natives are needed to wrap the value
    * JS will *automatically* box primitive values to fullfil property accesses
    * eg. `new String("abc")` creates a wrapper object for the primitive

string

- note that boxing a boolean false creates a *truthy* value, since objects are truthy
- unboxing can be done with the `valueOf` method, or can happen implicitly when the native becomes coerced
- while primitives become wrapped, arrays, objects, functions, and RegEx values are the same, whether created literally or with the constructor form:
  * ie. there is no unwrapped value

- although most of the native prototypes are plain objects:
  - `Function.prototype` is an empty function
  - `RegExp.prototype` specifies empty RegEx
  - `Array.prototype` is an empty array
- the `[[Class]]` property is a classification for values that are `typeof` object:
  - property can be accessed by borrowing `Object.prototype.toString` on the value
  - eg. `Object.prototype.toString.call([1, 2, 3])` gives `"[object Array]"`
  - primitive vaues are boxed, eg. `Object.prototype.toString.call(42)` gives `"[object Number]"`
  - note that the `[[Class]]` value for `null` and `undefined` are `"[object Null]"` and `"[object Undefined]"` , even though no such native wrappers exist

Coercion

- converting a value between types is called **type casting** when done explicitly, and **coercion** when done implicitly:
  - alternatively, type casting occurs at compile time, and type coercion occurs at runtime
  - note that JavaScript coercions *always* result in one of the scalar primitive values `string, number, boolean`

Abstract Value Operations

- abstract value operations specify the *internal* conversion rules used by JavaScript:
  - eg. `ToString, ToNumber, ToBoolean, ToPrimitive`
  - note that `ToString` is distinct from the `toString` method
- when any non-string is coerced to a string representation, `ToString` is used:
  - builtin primitive values have natural stringification, eg. `null` becomes `"null"`
    * note that very small or large numbers may be represented in exponent form
  - for regular objects, `ToString` uses the default `toString` which returns the internal `[[Class]]`
    * unless an object has its own `toString` method
    * eg. arrays have a overridden default `toString` that stringifies as the string concatention of its values, with a comma between each value
      · eg. `[1, 2, 3].toString()` gives `"1,2,3"`
- similarly, `JSON.stringify` is used to serialize a value to a JSON-compatible string value:
  - an optional second argument acts as a *replacer*:
    * an array or function that handles filtering certain object properties in the JSON
  - an optional third argument called *space*:
    * a number of spaces to use for indentation, or a string to replace spaces for indentation
  - values that are *not* JSON-safe have special cases:
    * `JSON.stringify` will automatically omit the `undefined` , function, and symbol values:
      · in an array, the value is replaced by null

            - if a property of an object, that property is excluded
        * attempting to JSON stringify an object with circular references throws an error
            - if an object value has a `toJSON` method defined, this method is called first to get a custom JSON-safe value for serialization
- when any non-number is coerced to a number, `ToNumber` is used:
    - eg. `true` becomes 1, `undefined` becomes `NaN`, `null` becomes 0
    - for string values, `ToNumber` emulates the rules for numeric literals, except if it fails, the result is `NaN` instead of an error
    - for objects and arrays, they are first converted to their primitive value equivalent using `toPrimitive`:
        * `toPrimitive` checks if the value has a `valueOf` method and if that method returns a primitive value, that is used for coercion
        * otherwise, `toString` will provide the value for the coercion, if present
        * if neither operation can provide a primitive, then an error is thrown
- when any non-boolean is coerced to a boolean, `ToBoolean` is used:
    - note that unlike other languages `1` is not *identical* to `true`, and `0` is not identical to `false`
    - `ToBoolean` coerces all *falsy* values to `false` and all *other* values to `true`
    - the falsy values are:
        * `undefined, null, false, +0, -0, NaN, ""`
    - all other objects are truthy:
        * eg. all objects, even wrappers of falsy primitives
        * eg. `"false", "0", "''", [], {}, function(){}` are all truthy
    - note there *are* some falsy objects that come from outside of JavaScript:
        * eg. `document.all` is a falsy object

Explicit Coercion

- to cast between strings and numbers, the builtin `String` and `Number` functions can be used, *without* the `new` keyword:
    - they use the abstract `ToString` and `ToNumber` operations defined earlier
    - other ways of explicit conversion:
        * calling `toString` (which wraps primitive values in a native first)
        * using the unary operators `+` and `-`:
            - special parsing rules prevent confusion with increment and decrement operators
            - unary `+` can also be used to coerce a `Date` object into a number
- similarly to coercing between strings and numbers, JS supports *parsing* a

number out of a string's contents:
  - using `parseInt` and `parseFloat`
    * the second argument takes the base to parse the number in
  - unlike coercion, parsing is tolerant of non-numeric characters and stops parsing when encountered, instead of giving `NaN`
  - these parse methods are designed to work on strings, so when a non string value as an argument is automatically coereced to a string first

Parsing gotchas:

```
parseInt(1/0, 19);      // 18, coerced to "Infinity", I in base-19 is 18
parseInt(0.0000008);    //  8, coerced to "8e-7"
parseInt(parseInt, 16); // 15, coerced to "function ...", f in base-16 is 15
```

- to coerce from non-booleans to booleans, `Boolean` without `new` can be used:
  - the unary `!` negate operator also explicitly coerced to boolean, while *flipping* the value
    * thus the double-negate `!!` can also be used to coerce to booleans
  - note that implicit boolean coercion would occur in a boolean context such as an `if` or ternary statement

Implicit Coercion

- generally, the `+` binary operator performs string concatenation if either operand is a string, and otherwise numeric addition:
  - however, when an object is an operand, it will use the `ToPrimitive` operation on the object:
    * which calls `valueOf` and then `toString` in an attempt to stringify the operand

Implicit coercion with `+`:

```
[1, 2] + [3, 4]; // "1,23,4"

42 + ""; // "42"
"" + 42; // "42"

var a = {
  valueOf: function() { return 42; },
  toString: function() { return  4; }
};
a + "";     // "42", using ToPrimitive
String(a); // "4"
```

```
[] + {}; // "[object Object]"
{} + []; // 0, {} is treated as an empty block
```

- on the other hand `-` is only defined for numeric subtraction
  - same with `*` and `/`

Implicit coercion with `-`:

```
"3.14" - 0; // 3.14
[3] - [1];  // 2, coerced to strings and then numbers
[1, 2] - 0; // NaN
```

- for implicit coercion of ES6 symbols:
  - explicit coercion of a symbol to a string is allowed
  - but implicit coercion of a symbol is *disallowed* and throws an error
    - symbol values cannot coerce to numbers either
  - symbols do explicitly and implicitly coerce to boolean true
- implicit coercion to boolean values is the most common form:
  - the following expression operations force a boolean coercion:
    1. the test in an `if` statement
    2. the test in a `for` header
    3. the test in `while` and `do..while` loops
    4. the test in ternary expressions
    5. the lefthand operand in `||` and `&&` operations
- unlike the logical operators in other languages, `||` and `&&` in JS work more like *selector* operators:
  - rather than returning booleans, these result in the value of one of their operands
  - both perform a boolean test (with `ToBoolean` if necessary) on the *first* operand:
    - for `||`, if the test is true, the expression results in the value of the first operand, and otherwise the second
    - for `&&`, if the test is true, the expression results in the value of the second operand, and otherwise the first
    - both are still performing *short-circuiting*
  - eg. `42 && "abc" === "abc"` and `null && "abc" === null`
  - similar to a kind of selecting ternary

Default assignment idiom with `||`:

```
function foo(a, b) {
  a = a || "hello";
  b = b || "world";
  console.log(a, b);
```

```
}

foo();          // prints "hello world"
foo("a", "b"); // prints "a b"
foo("c", "");  // prints "c world"
```

Guarding idiom with `&&` :

```
function foo() { console.log(a); }

var a = 42;
a && foo(); // prints 42
```

Equality

- JavaScript has two equality operators:
  - `==` AKA *loose* equality allows coercion in the equality comparison, while `===` AKA *strict* eqaulity disallows coercion
  - `!=` is the same as the `==` comparison, but negated, and similarly for `!==` with respect to `===`
- both equality operators follow the same initial algorithmic comparison steps:
  1. if the two values are of the same type, they are simply and naturally compared via identity
     - exceptions are `NaN` never being equal to itself and `+0` and `-0` being equal to each other
  2. for all objects, two values are equal only if the are both references to the *exact same* value
     - thus `==` and `===` act the same for two objects
     - no coercion occurs here
  - the remainder of the algorithm is different for loose equality:
    * if the values are of different types, one or both of the values need to be implicitly coerced, so that they end up as the same type
- coercion cases for loose equality:
  1. when comparing strings to numbers:
     - the string is implicitly coerced to a number using the `ToNumber` operation
  2. when comparing *anything* to booleans:
     - the boolean is implicitly coerced to a number
     - eg. both `"42" == true` and `"42 == false"` are false!
     - to test for truthy values, simply use `if (val)` to implicit coerce to boolean

3. when comparing `null` and `undefined` :
   - `null` and `undefined` are always equal and coerce to each other
4. when comparing objects to nonobjects:
   - the object is implicitly coerced to a primitive using the `ToPrimitive` operation
   - eg. `42 == [42]` and `new String("abc") == "abc"` are true

Edges cases from modifying native prototypes:

```
var i = 2;
Number.prototype.valueOf = function() {
  return i++;
};

var a = new Number(42);
if (a == 2 && a == 3) {
  console.log("gotcha"); // prints gotcha
}
```

Edge cases in falsy comparisons:

```
// all true comparisons:
"0" == false;
0 == false;
"" == false;
[] == false;
"" == 0;
"" == [];
0 == [];

[] == ![];     // same as [] == false due to unary !
2 == [2];
"" == [null]; // [null] coerces to ""
0 == "\n"     // whitespace strings coerce to 0
```

- coercion cases for relational comparison:
  1. `ToPrimitive` coercion is done on both values
  2. if either result is not a string, both values are coerced to numbers and compared numerically
  3. otherwise, they are compared lexicographically
  - note that for `a <= b` , `b < a` is evaluated and negated instead
    * similary for `a >= b` , `a < b` is evaluated and negated

Relational comparisons:

```
[42] < ["43"];      // true
["42"] < ["043"];   // false, lexicographically compared
[4, 2] < [0, 4, 3]; // false, "4,2" > "0,4,3"
Number([42]) < Number("043") // true, numerically compared

{b: 42} < {b: 43};  // false, both are "[object Object]"
{b: 42} > {b: 43};  // false, both are "[object Object]"
{b: 42} == {b: 43}; // also false, object comparison
{b: 42} <= {b: 43}; // true!
{b: 42} >= {b: 43}; // true!
```

Grammar

- JavaScript operators all have well-defined rules for precedence and associativity:
  - eg. `&&` has precedence over `||`, both have precedence over `=`
  - note that the statement-series `,` operator has the lowest precedence
  - eg. assignment and ternaries are right-asssociative, while most other operators are left-asssociative
- JavaScript has a feature called **automatic semicolon insertion (ASI)**:
  - JS assumes a semicolon in certain places even if omitted
  - only in certain places were the JS parser can *reasonably* insert a semicolon
  - useful for `do..while` loops that require a semicolon after
  - may cause unintended behavior with `return, continue, break, yield`
- function argument nuances:
  - there is a TDZ for ES6 default parameter values as well:
    * eg. `function foo(a = 42, b = a + b + 1)` is invalid, while `function foo(a = 42, b = a + 5)` is OK
  - omitting an argument is similar to passing an `undefined` value, except:
    * the builtin `arguments` array will not have entries if certain arguments are omitted
- `try..finally` nuances:
  - the `finally` clause *always* runs, right after the other clauses finish:
    * but if there is a `return` in a `try` clause, the `finally` clause runs immediately before *exiting* from the function
    * similarly for throwing errors, `continue`, and `break`
  - a `return` inside a `finally` can also override a previous `return`
- `switch` statement nuances:
  - `default` clause is optional
  - the matching between the cases and main switch expression is identical to the `===` algorithm
  - however, it is possible to still use loose equality with `true` switch expression
    * note that we are still explicitly matching `true`, so truthy values will fail to match

Using coercive equality:

```
var a = "42";
switch (true) {
  case a == 10:
    ...
```

```
    break;
  case a == 42:
    ...
    break;
}
```

Scope

- **scope** is the set of rules for *storing* variables in a location and *finding* those variables later:
  - scoping has some other uses beyond just determining how to lookup variables:
    * scoping can be used for **information hiding** ie. hiding variables and functions
    * hiding names also avoids collisions between variables with the same names
      · collisions also avoided through use of global namespaces or modules
  - for JS, *when* and *how* the scoping rules are set depends on its compilation process
  - traditional compilation process:
    1. tokenizing / lexing the source code
    2. parsing it into a syntax tree
    3. generating machine code from the syntax tree
  - unlike *traditional* compiled languages, JS is not compiled in advance, it is compiled as the program runs, microseconds before code is executed:
    * less time for optimization
    * must use tricks such as lazy compilation and hot recompliation to be efficient
- the JavaScript **engine** is responsible for start-to-finish compilation and execution:
  - calls upon the **compiler** to parse and generate code
  - uses **scope** in order to retrieve a look-up list of variables and their accessibility rules
    * due to nested scope, if a variable is not found in the immediate scope, the engine consults the next *outercontaining* scope, until the global scope has been reached
    * any variable declared within a scope is *attached* to that scope
  - eg. for the statement `var a = 2` :
    1. compiler will declare a variable (if not previously declared) in the current scope
    2. compiler *generates* code that will be run by the engine that actually *looks* up the variable in the scope and assigns to it, if found
    * note that the lookup that occurs can be for a LHS variable or a RHS variable:
      · LHS ie. *target* variable to assign to, eg. `a = 2`
      · RHS ie. *source* of the assignment, eg. `console.log(a)`

    - note that scope-related assignments will *implicitly* occur when assigning to function parameters
- LHS and RHS lookups are *distinct* in behavior when the variable has not been declared:
    - when a RHS lookup fails to find a variable, anywhere in the nested scope, a `ReferenceError` is thrown by the engine
    - when a LHS lookup arrives at global scope without finding a variable:
        * if the program is not in strict mode, the global scope will create a *new* variable of that name in the global scope and hand it back to the engine
        * in strict mode, implicit global variable creation is disallowed, so a `ReferenceError` is again thrown by the engine
    - note that a `ReferenceError` indicates a scope resolution failure, while other errors at this time indicate scope resolution was succesful, but an illegal action was attempted

Lexical and Dynamic Scope

- there are two models of scoping, **lexical** or static scoping and **dynamic** scoping:
    - with lexical scoping, the scoping rules are *defined* at lexing time ie. compile time:
        * based on where variables and blocks of scopes are *authored*, using nested scoping rules
        * no matter where or how a function is invoked, its lexical scope is only defined by *where* it was declared
        * most programming languages, including JavaScript, use lexical scoping rules
    - with dynamic scoping, lookup happens *dynamically* at runtime:
        * eg. `this` in JS is dynamically scoped, since its value depends on how its function was called
        * eg. Bash scripting, some Perl modes
    - scope lookup stops once the first match is found, and the same identifier name can be *shadowed* by inner scopes
- JavaScript does provide some ways to *dynamically* modify its lexical scoping rules:
    - can lead to dangerous side effects
    - eg. using `eval`, or the now deprecated `with` expression
        * both of these methods are restricted by strict mode
        * both of these methods force the compiler to limit or avoid optimizations, so code will run *slower*

Changing lexical scope with `eval` :

```
function foo(str, a) {
  eval(str);
  console.log(a, b);
}

var b = 2;
foo("var b = 3;", 1); // prints 1, 3
```

Example of the now deprecated `with` keyword:

```
var obj = { a: 1, b: 2, c: 3 };

// tedious reassignment
obj.a = 2;
obj.b = 3;
obj.c = 4;

// with shorthand
with (obj) {
  a = 3;
  b = 4;
  c = 5;
};
```

Changing lexical scope using `with` :

```
function foo(obj) {
  with (obj) { a = 2; }
}

var o1 = { a: 3 };
var o2 = { b: 3 };

foo(o1);
console.log(o1.a); // prints 2

foo(o2);
console.log(o2.a); // prints undefined
console.log(a);    // prints 2, global has been *leaked*
   // with keyword creates a new lexical scope, but a is missing,
   // so lookup goes to the global level and creates a new declaration (non-strict)
```

Lexical Scope with Arrow Functions

- ES6 introduced a new syntactic form of function declaration called the **arrow function**
  - *pros*:
    - the *"fat arrow"* is a shorthand for the `function` keyword
    - performs a lexical binding for `this`, rather than following the normal `this` binding rules
  - *cons*:
    - arrow functions are all anonymous

Illustrating the problem of lexical scope with `this`:

```
var obj = {
  id: "foo",
  identify: function idFn() {
    console.log(this.id);
  }
}
var id = "bar";

obj.identify(); // prints foo
setTimeout(obj.identify, 100); // prints bar, this binding is lost
                               // since this is bound dynamically

// explicit fix:
var obj = {
  id: "foo",
  identify: function idFn() {
    var self = this;
    setTimeout(function log() { // have to move setTimeout inside
      console.log(self.id);
    }, 100);
  }
}

// bind fix:
var obj = {
  id: "foo",
  identify: function idFn() {
    setTimeout(function log() { // have to move setTimeout inside
      console.log(this.id);
    }.bind(this), 100);
  }
```

```
}

// fat-arrow fix:
setTimeout(() => { obj.identify(); }, 100);
```

Function Scope

- JavaScript `var` declarations follow **function scope** where the declarations within a function are effectively hidden from the outside:
  - ie. follow a scope *unit* of functions
  - there are serveral considerations for functions as scope
- functions expressions can be *anonymous* (omitting the name) or *named*:
  - function declarations cannot omit the name
  - drawbacks to anonymous functions:
    1. anonymous functions have no name to display in stack traces
    2. without a name, the function can only refer to itself through the deprecated `arguments.callee`
    3. without a name, code may be less readable or understandable
  - note that *inline* functions can still be named, they are not forced to be anonymous

Anonymous vs. named inline functions:

```
setTimeout(function() {
  console.log("1 sec passed");
}, 1000);

setTimeout(function timeoutHandler() {
  console.log("1 sec passed");
}, 1000);
```

- by wrapping a function in parentheses, function expressions and **immediately invoked function expressions (IIFE)** can be created:
  - useful for avoiding polluting the enclosing scope, since the identifier of the function (if named), is found *only* inthe scope within the IIFE, and is inaccessible outside the IIFE

Variations on IIFEs:

```
(function() {...})();       // anonymous IIFE
(function() {...}());       // equivalent IIFE
(function IIFE() {...})(); // named IIFE
```

```
(function IIFE(global) {...})(window); // passing in arguments to IIFE

(function IIFE(def){ // alternative inverted IIFE definition used in
  def(window);       // the Universal Module Definition (UMD) project
})(function def(global) {...});
```

Block Scope

- although functions are the most common unit of scope used in JS, **block scoping** is another popular scoping unit:
  - used by many languages, eg. C/C++, Java, Python
  - *pros*:
    - allows for even *more* information hiding, at a finer granularity *within* functions
    - allows for more efficient garbage collection and *faster* reclamation of memory
    - easier to add additional, *explicit* scoped blocks (rather than creating new functions)
  - JavaScript *does* provide some facilities for achieving block scope:
    - `with, try/catch, let, const`
- the `with` statement is an example of block scope since the created scope is only within the statement, not the enclosing function
- the variable declaration in the `catch` clause of a `try/catch` is block scoped to the `catch` block
- the `let` keyword, introduced by ES6, attaches the variable declaration to the scope of the *containing* block, specified by brackets:
  - `let` declarations will also *not* hoist to the entire scope of the block
  - when used in blocks, a `let` declaration in the loop header will actually *rebind* the variable on *each* iteration of the loop, which is useful for handling closures
  - ES6 also added the `const` keyword, which also creates a block-scoped variable whose value is fixed

Using `let` loops:

```
for (let i = 0; i < 10, i++) {
  console.log(i);
}
console.log(i); // ReferenceError with let instead of var
```

```
// let loop rebinding: (equivalent code to let loop)
{
  let j;
  for (j = 0; j < 10; j++) {
    let i = j;
    console.log(i);
  }
}
```

Polyfilling block scope:

```
{ // ES6
  let a = 2;
  console.log(a);
}
console.log(a);

// is polyfilled to:
try {throw 2} catch(a) {
  // ES3 catch has block scope!

  // alternatively, use an IIFE? isn't an IIFE faster than try/catch?
  // IIFE performs faster, but wrapping a function around arbitrary code changes
  // the meaning of the code, eg. this, return, break, and continue change meanings
  console.log(a);
}
console.log(a);
```

Hoisting

- generally, a JavaScript program is *interpreted* line-by-line:
  - this is mostly true, except for the case of declarations
  - the engine will have the compiler *compile* the code in its entirety (usually) before it interprets ie. runs it:
    - part of the compilation phase is to find and associate declarations with their appropriate scopes
      - thus all declarations are processed *first*, before any part of the code is executed
  - ie. declarations are **hoisted** or moved from where they appear in the flow of the code to the top of the code
    - note that only the declarations themselves are hoisted, not any as-

signments or other executable logic
    - thus function *expressions* are *not* hoisted
  - thus `a = 2` and `var a = 2` have two *distinct* statements, one of which (the declaration) is hoisted
  - note that functions are always hoisted *first*, then variables
- note that declarations appearing inside normal blocks (such as `if-else` blocks) are hoisted to the enclosing scope, instead of being conditional

Illustrating hoisting:

```
a = 2;
var a;
console.log(a); // prints 2

// declaration is hoisted as:
var a;
a = 2;
console.log(a);

console.log(a); // prints undefined
var a = 2;

// declaration is hoisted as:
var a;
console.log(a);
a = 2;
```

Hoisting in function declarations:

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

// declarations are hoisted as:
function foo() {
  var a;
  console.log(a);
  a = 2;
}
foo();
```

Hoisting in function expressions:

```
foo(); // TypeError
bar(); // ReferenceError

var foo = function bar() {...};

// declarations are hoisted as:
var foo;
foo(); // TypeError since foo has no value yet, due to using function expression
bar(); // ReferenceError since name of named function expression is not accessible
       // in the *enclosing* scope
foo = function() { var bar = ...self... };
```

Hoisting functions first:

```
foo(); // prints 3, not 1 or 2
var foo;
function foo() { console.log(1); }
foo = function() { console.log(2); };
function foo() { console.log(3); }

// declaration is hoisted as:
function foo() { console.log(1); }
function foo() { console.log(3); } // subsequent declaration overrides previous one
// var foo is a *duplicate* and thus ignored declaration
foo();
foo = function() { console.log(2); };
```

- note that `let` and `const` *are* actually still hoisted:
  - all JavaScript declarations are hoisted
  - the difference is that there is a **temporal dead zone** between the hoisted declaration and the actual declaration of the variable for ES6 block scoped variables
    - accessing a `let` or `const` before they are declared thus throws a `ReferenceError` since they are accessed in this dead zone

Illustrating hoisting of `let`:

```
let x = "outside";
(function() {
  // x declaration hoisted here, start of TDZ for x
  console.log(x); // throws a ReferenceError instead of printing "outside",
                  // so x *is* hoisted
  // TDZ ends
  let x = "inner";
```

```
})
```

Closures

- **closure** is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope:
  - ordinarilly, we would expect the entirety of the scope of a function to go away after execution, when the garbage collector runs:
    - however, with closures, this is not the case, and the scope of a returned function can still be accessed
    - implemented using *nesting links*, and placing certain call frames on the heap instead of the stack
  - closures happen naturally in JavaScript as a result of writing code that rely on lexical scope:
    - whenever an inner function is *transported* outside of its lexical scope ie. treated as first-class values, it maintains a closure reference to its original lexical scope
    - eg. timers, event handlers, AJAX requests, callback functions, etc.

Illustrating closures:

```
function foo() {
  var a = 2;
  function bar() {
    console.log(a);
  } // bar has a *closure* over the scope of foo and rest of its accessible scopes
    // ie. bar *closes* over the scope of foo, because bar is nested inside foo
  return bar;
}

var baz = foo();
baz(); // prints 2, closure in action here, since baz is executed
       // *outside* of its declared lexical scope
```

Concrete closure examples:

```
function wait(msg) {
  setTimeout(function timer() { console.log(msg); }, 1000);
}
wait("Hello!"); // uses closures

function debugButton(name, selector) {
  $(selector).click(function activator() {
    console.log("activating " + name);
  });
```

```
}
// uses closures
debugButton("Continue", "#continue");
debugButton("Quit", "#quit");
```

Closure and loops:

```
for (var i = 1; i <= 5; i++) {
  setTimeout(function timer() {
    // each timer function is closed over same shared *global* scope,
    // due to the declaration of i using var

    console.log(i); // when each timer runs after setTimeout triggers, i is 6

    // the *desired* functionality is to capture a different copy
    // of i at each iteration, ie. a per-iteration block scope
  }, i*1000);
} // prints 6 6 6 6 6, one 6 each second

// solving with IIFE:
for (var i = 1; i <= 5; i++) {
  (function(j) { // use an IIFE to *create* a new lexical scope within global scope
    setTimeout(function timer() {
      console.log(j);
    }, j*1000);
  })(i);
} // prints 1 2 3 4 5

// solving using let:
for (let i = 1; i <= 5; i++) {
  // let has per-iteration rebinding
  setTimeout(function timer() {
    console.log(i);
  }, i*1000);
} // prints 1 2 3 4 5
```

Modules

- the **module** code pattern leverages closures in order to reveal a certain public API while hiding implementation details, and requires:
  1. an outer enclosing function, that must be invoked at least once to create

   a new module *instance*
2. the enclosing function must return back at least one inner function
   - the inner function thus has closure over the *private* scope

Example module pattern:

```
function Module() {
  var foo = "bar";
  var qaz = [1, 2, 3];

  function doFoo() { console.log(foo); }
  function doQaz() { console.log(qaz.join("!")); }

  return {
    doFoo: doFoo,
    doQaz: doQaz
  };
}

var mod = Module();
mod.doFoo(); // prints bar
mod.doQaz(); // prints 1!2!3
```

- ES6 added first-class syntax support for modules:
  - each file is treated as a separate module
  - modules can import other modules or specific API members, and export their own public API members
  - ES6 module APIs are static and thus import errors can be checked at runtime
  - a module can:
    * `export` an identifier to the public API for the current module
    * `import` one or more members from a module's API into the current scope

this and Binding Rules

- JavaScript's `this` mechanism allows functions to be reused against multiple different *context* objects:
  - a more elegant mechanism than *explicitly* passing along an object or context reference as a parameter
  - a common misconception of `this` is that it refers to the function itself or to a function's lexical scope:
    * however, altough `this` may point to a calling function, it does not always do so
    * there is also no way to use a `this` reference to look something up in a lexical scope
      · ie. there is no bridge between lexical scopes
  - `this` is a dynamic, runtime binding that is *contextual* based on the conditions of the function's *invocation*
    * the `this` reference is a property on the activation record of the function in the call stack
    * ie. based on the function's **call-site**

Utility of `this` :

```
function identify() { return this.name; }
function speak() {
  var greeting = "Hello, I'm " + identify.call(this);
  console.log(greeting);
}

var me  = { name: "Bob" };
var you = { name: "Blob" };

identify.call(me); // prints Bob
speak.call(you);   // prints Hello, I'm Blob
```

Allowing a reference to get a reference to itself:

```
function foo(num) {
  console.log(num);
  this.count++; // not bound to foo!
}
foo.count = 0;

for (let i = 0; i < 10; i++) {
  if (i > 5) {
```

```
    foo(i);
  }
}

console.log(foo.count); // prints 0?

// forcing the binding on this to point to foo:
for (let i = 0; i < 10; i++) {
  if (i > 5) {
    foo.call(foo, i);
  }
}

console.log(foo.count); // prints 4
```

Binding Rules

---

1. the first binding rule, **default binding**, applies for *standalone* function invocation:
   - the default, catch-all rule when none of the others apply
     - the default binding points `this` at the global object
   - variables declared in the global scope are synonymous with the *global object* properties of the same name
   - note that in strict mode, the global object is not eligible for the default binding, so `this` is instead set to undefined

Default binding:

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

2. in **implicit binding**, the call-site may have a context ie. owning object:
   - the function call is preceeded by an object reference
     - implicit binding points `this` to *that* object
   - note that only the top or last level of an object property reference chain matters to the call-site
   - a problem with implicit binding occurs when an implicitly bound function loses its binding, and falls back to the default binding:
     - occurs commonly with function callbacks

    - some frameworks will also forcefully modify `this` during the callback
    - need a way to *fix* the `this`

Implicit binding:

```
var obj2 = {
  a: 2,
  foo: foo // doesn't matter whether foo is defined here or added as a reference
};
var obj1 = {
  a: 42,
  obj2: obj2
};
obj1.obj2.foo(); // prints 2
```

Implicit binding loss:

```
function doFoo(fn) {
  fn(); // call-site is what matters, fn becomes another reference to foo
}
var obj = {
  a: 2,
  foo: foo
};
var a = "global";
doFoo(obj.foo); // prints global, not 2
```

3. **explicit binding** forces a function call to use a particular object for the `this` binding, *without* putting a property function reference on the object:
    - uses `call` or `apply` , which both take in an object to use for `this` as the first argument
    - *hard binding* is a form of explicit binding that fixes the issue of binding loss
        - provided by `bind` in ES5, which returns a new function that is hardcoded to call the original function with `this` specified context
    - some APIs will provide an optional *context* parameter that uses a form of explicit binding to use that context
    - `apply` also helps to spread out an array as parameters (replaced by the ES6 spread operator)
    - `bind` also is useful for currying functions

Explicit binding:

```
var obj = { a: 2 };
foo.call(obj); // prints 2
```

Hard binding:

```
var obj = { a: 2 };
var bar = function() {
  foo.call(obj); // actual call-site
}
bar(); // prints 2
setTimeout(bar, 100); // also prints 2
bar.call(window);     // still prints 2

// simple example hard binding helper:
function bind(fn, obj) {
  return function() {
    return fn.apply(obj, arguments);
  };
}
var bar = bind(foo, obj);

// same functionality provided by ES5 bind function:
var bar = foo.bind(obj);
```

API calls with context:

```
function foo(el) {
  console.log(el, this.id);
}
var obj = { id: "bar" };
[1, 2, 3].forEach(foo, obj); // prints 1 bar 2 bar 3 bar
```

4. the `new` binding is a special binding rule that is used with the `new` operator:
   - note that the `new` operator in JS has *no connection* to object-oriented functionality
   - in JS, **constructors** are just functions that *happen* to be called with the `new` operator:
     - not attached to classes, nor are they instantiation a class
     - not a special type of function either, more like a construction call *of* a function
   - in a construction call:
     1. a brand new object is constructed
     2. the new object is `[[Prototype]]` linked
     3. the new object is set as the `this` binding for that function call

4. unless the function returns its own alternate object, the function call will *automatically* return the new object

`new` binding:

```
function foo(a) {
  this.a = a;
}
var bar = new foo(2);
console.log(bar.a); // prints 2
```

Precedence

- default binding is the lowest priority rule of the four:
  - next, implicit binding has the next lowest priority
  - followed by explicit binding, and then `new` binding with the highest priority:
    * note that `call` and `apply` override a `bind` hard binding
    * in addition, if `null` or `undefined` is passed as a binding parameter, default binding applies instead
      · a *safer* alternative may be to pass in an *"ghost"* object instead that is guaranteed to be totally empty
      · eg. `Object.create(null)` is *"more empty"* than `{}`
  - this may be suprising since the previous hard binding helper does *not* have a way to override the hard binding, but `new` binding still supercedes it:
    * this is because the builtin ES5 `bind` is more sophisticated, and actually checks if the hard-bound function has been called with `new` or not
    * overriding hard binding is useful because it allows for a function that can construct objects with some of its arguments preset from a `bind`, while ignoring the previously hard-bound `this`
      · ie. helps with partial application and currying
  - note that *indirect* references to a function can be created, eg. the result value of an assignment expression
    * these obey default binding, rather than another type of binding expected from the assignment expression
  - an alternative binding rule is **soft binding**, where a function can still be manually rebound via implicit or explicit binding, but has an alternative if the default binding would otherwise apply
    * unlike hard binding, which *cannot* be manually overrided with implicit binding or explicit binding

- finally, ES6 introduced a new kind of function that has its own binding rules:
  - instead of following standard `this` binding rules, arrow functions adopt the `this` binding from their *enclosing* scope
  - this lexical binding *cannot* be overriden, even with `new`
  - commonly used with callbacks
  - similar in spirit to using `var self = this` to lexically capture `this`, vs. using `this` -style binding with `bind`

Arrow function bindings:

```
function foo() {
  return () => {
     console.log(this.a);
  };
}
var obj1 = { a: 2 };
var obj2 = { a: 3 };
var bar = foo.call(obj1);
bar.call(obj2); // prints 2, not 3, not explicitly rebound
```

Objects

- `object` in JavaScript is one of its primary types:
  - a function is a subtype of object, technically a *callable* object
  - arrays are also a structured form of object
  - can be created using a literal form, or constructed form
  - object have **properties** that can be set and accessed:
    - through `.` or `[]` operator
    - note that property names are *always* strings, so other property name types will be *coerced* to strings
    - ES6 adds **computed property names**, where an expression surrounded by `[]` can be used in the key position of an object literal declaration
      - useful with ES6 `Symbol` s
  - note that although functions can be a property of an object, these are not exactly *methods* that are bound to the object like in other languages:
    - the function property is simply another reference to the function, even if it was declared *and* defined within the object
    - the only distinction between the references would occur if the function had a `this` reference and an implicit binding was used
  - arrays are objects that are numerically indexed:
    - as objects, arrays can have *additional* named properties, without changing its `arr.length` property
    - note however that property names that coerce to numbers will be treated as numeric indices

Duplicating Objects

- duplicating objects has the issue of *shallow* vs. *deep* copies:
  - in some situations, deep copies may create an infinite circular duplication, since extra duplications must occur
  - while shallow copies will only create new *references*, instead of additional concrete duplications
  - one copying solution is to duplicate JSON-safe objects:
    - `var newObj= JSON.parse(JSON.stringify(obj))`
    - not always sufficient for objects that are not JSON-safe
  - ES6 provides a shallow copy function:
    - `var newObj = Object.assign({}, obj)`
    - takes target object, and one or more source objects
    - copies enumerable, owned keys to the target via assignment only,

    and returns the target
  * ES6 spread operator does the same, while being slightly more performant

Properties

- ES5 provides **proerty descriptors** that allow properties to be described with extra characteristics:
  - `Object.getOwnPropertyDescriptor(obj, name)` gets the property descriptor for `obj.name`
  - `Object.defineProperty(obj, name, descriptor)` creates or modifies an existing property with the characteristics in descriptor
  - the descriptor is an object that specifies:
    * the `{ value, writable, enumerable, configurable }` characteristics
    * writing to a non-writable property fails and causes an error in strict mode
    * a configurable property can be updated by `Object.defineProperty`
      · a non-configurable property also cannot be removed with `delete`
    * enumerable controls whether the property will show up in object-property enumerations such as the `for..in` loop or `Object.keys`
      · order of iteration over an object's properties is not guaranteed
      · thus note that `for..in` applied on arrays gives the numeric indices *as well as* any enumerable properties
      · `Object.getOwnPropertyNames` gives all properties, enumerable or not
- there are different ways to achieve *shallow* **immutability** using ES5:
  1. combining `writable: false` and `configurable: false` essentially creates a *constant* that cannot be changed, redefined, or deleted
  2. `Object.preventExtensions` prevents an objects from having new properties added to it
  3. `Object.seal` creates a *sealed* object, which essentially calls `Object.preventExtensions` and also marks existing properties as `configurable: false`
     - cannot add or delete properties (though existing properties *can* be modified)
  4. `Object.freeze` creates a frozen object, which essentially calls `Object.seal` and also marks existing properties as `writable: false`
     - prevents any changes to the object
- in terms of property accesses, the access doesn't *just* look in the object for a matching propery:

- instead, according to the spec, the code perfomrs a `[[Get]]` operation that:
  1. inspects the object for a property of the requested name
  2. if found, returns the value accordingly
     * *otherwise*, `undefined` is returned instead
     * note that this is diferent from referencing variables, where a variable that cannot be resolved from lexical scope lookup will give a `ReferenceError`
- to set a property, the code performs a `[[Put]]` operation that:
  1. if the property is an accessor descriptor, call the setter
  2. if the property is not writable, either fail or throw an error
  3. otherwise, set the value to the existing property
  * if property is not yet present, the operation is even more complex
- ES5 introduced a way to override part of these default operations on a per-property level:
  * using *getters* and *setters*
  * when a property has a getter or setter, its definition becomes an **accessor descriptor**:
    · an accessor descriptor does not have `value` and `writable` fields
    · has the additional `set` and `get` characteristics
    · in contrast to a normal **data descriptor** property
  * if only a getter is defined, setting the property later will silently fail

ES5 getters and setters:

```
var myObj = {
  get a() { return 2; }
};

Object.defineProperty(myObj, "b", {
  get: function() { return this.a * 2; },
  enumerable: true
});

myObj.a; // gives 2
myObj.b; // gives 4
```

- the `in` operator checks if a property is *in* an object, *or* if exists at a higher level of the `[[Prototype]]` chain object traversal
  - eg. `("a" in myObj)`
  - note that the `in` operator does not check for *values* inside a container, just properties
- on the other hand, `myObj.hasOwnProperty` checks if *only* `myObj` has the prop-

erty or not, ignoring the prototype chain
  - however, it is possible for an object to not link to `Object.prototype` , in which case the test will fail
    * in this case, a more robust check is:
      · `Object.prototype.hasOwnProperty.call(myObj, "a")`
- the `for..of` loop added by ES6 allows for iterating over the values of objects directly:
  - however, it requires an iterator object created by a default `@@iterator` function
    * loop then iterates over return values using the iterator object's `next` method
    * iterators act similar to generator functions
  - arrays have this function built in, but it can be manually defined for objects

Defining an iterator for an object:

```
var myObj = {
  a: 2,
  b: 3,
  [Symbol.iterator]: function() {
    var self = this;
    var idx = 0;
    var ks = Object.keys(self);
    return {
      next: function() {
        return {
          value: self[ks[idx++]],
          done: (idx > ks.length)
        };
      }
    };
  }
};

for (var v in myObj) {
  console.log(v, myObj[v]);
} // prints a 2 b 3

for (var v of myObj) {
  console.log(v);
} // prints 2 3
```

Object Prototypes

- the `[[Prototype]]` property is an internal property of all objects, which is a reference to another object:
  - at creation, almost all objects are given a non `null` value for this property
  - different operations use a `[[Prototype]]` chain lookup process to find properties:
    * the default `[[Get]]` operation follows the `[[Prototype]]` link of an object if it cannot find the requested property on the object directly
      · if no matching property is *ever* found by the end of the chain, the return result is `undefined`
    * a `for..in` loop also lookups all enumerable properties that can be reached via an object's chain
      · similarly, the `in` operator will check the entire chain of the object for existence of a property, regardless of enumerability
  - the top of the `[[Prototype]]` chain is usually builtin `Object.prototype` :
    * this object inclues various common utilities, such as `toString` , `valueOf` , and `hasOwnProperty`

Illustrating object chain lookups:

```
var foo = { a: 2 };
var bar = Object.create(foo); // create object linked to foo
bar.a; // gives 2

for (var k in bar) {
  console.log(k);
} // prints a

("a" in bar); // gives true
```

- **shadowing** occurs when a property name ends up both on an object and a higher level of the prototype chain starting at that object:
  - the property directly on the object *shadows* the other property
  - thus there are three scenarios for an assignment `obj.foo = "bar"` when `foo` is at a higher level of the prototype chain:
    1. if a normal data accessor property is higher in the chain, and it is not read-only, then a new `foo` property is added directly to `obj` , resulting in a shadowed property
    2. if `foo` is higher in the chain but it is read-only, then the setting of the existing property as well as the creation of the shadowed

property on `obj` are disallowed and silently fails

3. if `foo` is higher in the chain and it is a setter, the setter will always be called
    * no new property is shadowed on `obj`, and the setter is not redefined

* however, in cases 2 and 3, `Object.defineProperty` can *still* be used to shadow a property

Object-Oriented Design

- although JavaScript has *some* class-like syntactic elements such as `new` and `instanceof`, JS does *not* actually have classes:
  - however, since classes and object-oriented design are design patterns, it is possible to implement approximations for classical class functionality
  - under the surface, these class approximations are *not* the same as the classes in other languages
  - in traditional classes, inheritance and polymorphism are both achieved using some sort of *copy* behavior:
    - ie. child class really *contains* a copy of its parent class, rather than having some sort of referential relative link to its parent
- JavaScript's object mechanism does *not* automatically perform copying behavior when you inherit or instantiate:
  - since there are no classes in JavaScript to instantiate or inherit from, only objects
  - this missing behavior is *emulated* using explicit and implicit **mixins**
    - mixins are one way to achieve class-like behavior

Mixins

- since JS does not provide a way to copy behavior ie. properties from another object:
  - we can create and use a utility that manually copies these properties, usually called `extend` or `mixin` by libraries and frameworks
    - this mixin approach *mixes* in the nonoverlapping contents of two objects
    - ie. **explicit mixin**
  - *pros*:
    - achieves an approximation of inheritance and polymorphism
    - can partially emulate multiple inheritance by mixing in multiple objects
  - *cons*:
    - the objects still operate separately due to the nature of copying
      - eg. adding properties to one of the objects does not affect the other after the mixin
    - JS functions cannot really be duplicated, so a duplicated *reference* is created instead
      - if one of the shared function objects is modified, both objects would be affected via the shared reference

    - in the similar **parasitic mixin** pattern, we initially make a copy of the definition from the parent class ie. object, and then mix in the child class
- JS did not support a facility for *relative* polymorphism (prior to ES6):
    - thus *explicit* pseudopolymorphism is used in the miin in the statement `Vehicle.drive.call(this)`
    - absolutely rather than relatively make a reference to the `Vehicle` object
    - *cons*:
        * this pseudopolymorphism creates *brittle*, manual linking which is very difficult to maintain when compared to relative polymorphism

Mixin utility:

```
function mixin(src, target) {
  for (var key in src) {
    if (!(key in target)) {
      target[key] = src[key];
    }
  }
  return target;
}

var Vehicle = {
  engines: 1,
  ignition: function() {...},
  drive: function() {...}
};

var Car = mixin(Vehicle, {
  wheels: 4,
  drive: function() {
    Vehicle.drive.call(this);
    ...
  }
});
```

- **implicit mixins** are also closely related to explicit pseudopolymorphism:
    - essentially *borrows* functionality from another object's function and calls it in the context of another object
        * once again, *mixes* in behavior from two objects
    - exploiting `this` binding rules
    - still a explicit, brittle call that cannot be made into a more flexible relative reference

Example of implicit mixins:

```
var Foo = {
  qaz: function() {
    this.count = this.count ? this.count+1 : 1;
  }
}

Foo.qaz();
Foo.count; // gives 1

var Bar = {
  qaz: function() {
    Foo.qaz.call(this);
  }
}

Bar.qaz();
Bar.count; // gives 1, not shared state with Foo
```

Using Prototypes

- all functions in JavaScript by default get a public, nonenumerable property called `prototype` , which points at an arbitrary object:
  - each object created from calling `new Obj()` will end up prototype-linked to the `Obj.prototype` object
  - this behavior is *similar* to the copying of behavior that occurs when instantiating traditional classes:
    - but no copying in JS, instead creates links between objects
    - this mechanism is **prototypal inheritance**, the dynamic version of clasical inheritance
    - not quite *inheritance*, since inheritance implies copying, but rather *delegation*, where an object can delegate properties and function access to another object
      - ie. delegating behavior to another object *upwards* the prototype chain
  - the `prototype` object of each function also has a `.constructor` property that points to the function:
    - although this `.constructor` property will appear on newly created objects due to the chain lookup, this property does not necessarily indicate *which* function constructed the object

        * ie. constructor does not mean constructed by

Common misunderstandings of using "classes" in JS:

```
function Foo(name) {
  this.name = name;
}

Foo.prototype.myName = function() {
  return this.name;
};
Foo.prototype.constructor === Foo; // true, builtin property of prototype

// the myName property on the Foo.prototype is *not* being copied over
// but *linking* occurs, and Object.getPrototypeOf(a) === Foo.prototype
var a = new Foo("a");
var b = new Foo("b");

// this lookup follows the prototype chain to Foo.prototype
a.myName(); // gives a
b.myName(); // gives b

a.constructor === Foo; // true, but *only* due to following the prototype chain
b.constructor === Foo; // true,                          """"
```

Illustrating `.constructor` nuances:

```
function Foo() {...}
Foo.prototype = {...} // creating a new prototype object, missing .constructor

var a = new Foo();
a.constructor === Foo;     // false
a.constructor === Object; // true, delegated all the way to Object.prototype
```

Prototypal Inheritance

- prototypal inheritance uses `Object.create` to create a new prototype object that is linked to another prototype object:
  - if simple assignment was used eg. `Bar.prototype = Foo.prototype` :
    * copies the reference to the prototype object, so that modifying it changes the now *shared* prototype object
    * defeats goal of inheritance
  - if `Bar.prototype = new Foo()` was used instead:

    * does in fact create a new object that is linked to `Foo.prototype`
    * however, this may have side effects from using the constructor call
      · this `Foo` constructor call should be called later when the `Bar` descendants are created
  – in ES6, should use `Object.setPrototypeOf(Bar.prototype, Foo.prototype)` in order to modify the existing prototype object

Using prototypes to create delegation links that emulate inheritance:

```
function Foo(name) {
  this.name = name;
}

Foo.prototype.myName = function() {
  return this.name;
};

function Bar(name, label) {
  Foo.call(this, name);
  this.label = label;
}

// new Bar.prototype linked to Foo.prototype,
// Bar.prototype.construtor is gone!
Bar.prototype = Object.create(Foo.prototype);

Bar.prototype.myLabel = function() {
  return this.label;
};

var a = new Bar("a", "obj a");
a.myName();  // gives a
a.myLabel(); // gives obj a
```

- object relationships can be tested using:
  – the `instanceof` operator which takes a plain object and a function:
    * eg. `a instanceof B` answers whether in the entire prototype chain of `a`, does the object pointed to by `B.prototype` ever appear
  – the `obj.isPrototypeOf` function:
    * eg. `a.isPrototypeOf(b)` answers whether `a` appears anyhere in the prototype chain of `b`
  – `Object.getPrototypeOf` directly retrieves the `[[Prototype]]` of an object
    * `obj.__proto__` is an alternate way to access the internal

`[[Prototype]]`
- standardized in ES6, actually a getter and setter

Using Create

- `Object.create` creates a new object linked to the specified object:
  - gives power of delegation of the `[[Prototype]]` mechanism
  - without unnecessary complications of `.prototype` and `.constructor` references, etc.
  - `Object.create(null)` creates an object that has an empty prototype link:
    - thus object cannot delegate anywhere
    - no prototype chain, so `instanceof` always returns false
    - these objects are *dictionaries* that can be used purely for storing data
  - `Object.create` supports additional functionality in its second argument:
    - the second argument specifies property names to add to the newly created object via their property descriptors

Polyfilling basic `Object.create` functionality:

```
if (!Object.create) {
  Object.create = function(o) {
    function F(){}
    F.prototype = o;
    return new F();
  };
}
```

Delegation-Oriented Design

- because JavaScript does not use traditional copy-base inheritance, it may be more appropriate to use a **delegation-oriented design** rather than a **object (prototypal)-oriented design**:
  - in traditional OOP, child tasks inherit from a parent class, and then add or override functionality, creating *specialized* behavior
  - in delegated design, rather than *composing* related objects together through inheritance, related objects are kept as *separated* objects, and instead one object will *delegate* to the other when needed
    - ie. objects are peers of each other and delegate among themselves, rather than having parent and child relationships
  - note that JS disallows creating a *cycle* where two or more objects are mutually delegated to each other

Class-based vs. delegation-based design:

```
// class-based approach (in another language):
class Task {
  id;
  Task(ID) { id = ID; }
  output() { print(id); }
}

class LabeledTask inherits Task {
  label;
  LabeledTask(ID, Label) { super(ID); label = Label; }
  output() { super.output(); print(label); }
}

// vs. delegation in JS:
Task = {
  setID: function(ID) { this.id = ID; }
  output: function() { console.log(this.id); }
};

LabeledTask = Object.create(Task);
LabeledTask.prepareTask = function(ID, Label) {
  this.setID(ID);
  this.label = Label;
};
LabeledTask.outputTaskDetails = function() {
  this.outputID();
  console.log(this.label);
};
// note that both data members are data properties on the delegator (LabeledTask),
// not on the delegate (Task), due to the this-binding
```

Another prototypal vs. delegation example:

```
// prototypal approach in JS:
function Foo(who) {
  this.me = who;
}
Foo.prototype.identify = function() {
  return "I am " + this.me;
};

function Bar(who) {
```

```
  Foo.call(this, who);
}
Bar.prototype = Object.create(Foo.prototype);
Bar.prototype.speak = function() {
  return "Hello " + this.identify();
};

var b1 = new Bar("b1");
var b2 = new Bar("b2");
b1.speak();     // gives Hello I am b1
b2.identify(); // gives I am b2

// vs. delegation in JS:
Foo = {
  init: function(who) {
    this.me = who;
  },
  identify: function() {
    return "I am " + this.me;
  }
};

Bar = Object.create(Foo);
Bar.speak = function() {
  return "Hello " + this.identify();
};

var b1 = Object.create(Bar);
b1.init("b1");
var b2 = Object.create(Bar);
b2.init("b2");
b1.speak();     // gives Hello I am b1
b2.identify(); // gives I am b2
```

Introspection

- **type introspection** has to do with inspecting an instance to find out what *kind* of object it is:
  - in JS, introspection different depending on whether an prototypal or delegation-based approach is taken

Prototypal vs. delegation introspection:

```
// with prototypal design:
function Foo() {...}
Foo.prototype...

function Bar() {...}
Bar.prototype = Object.create(Foo.prototype);

var b = new Bar();

// all true tests:
Bar.prototype instanceof Foo;
Foo.prototype.isPrototypeOf(Bar.prototype);
b1 instanceof Foo;
b1 instanceof Bar;
Foo.prototype.isPrototypeOf(b1);
Bar.prototype.isPrototypeOf(b1);

// with delegation design:
var Foo = {...};

var Bar = Object.create(Foo);
Bar...

var b = new Bar();

//all true tests:
Foo.isPrototypeOf(Bar);
Foo.isPrototypeOf(b);
Bar.isPrototypeOf(b);
```

- another common introspection method is to use **duck typing**:
  - simply check that an object has a capability, instead of testing for its type
  - can be a more brittle and risky test, eg. ES6 promises assume unconditionally that an object with a `then` method is a promise

Duck typing:

```
if (a.duckWalk && a.duckTalk) {
  a.duckWalk();
  a.duckTalk();
}
```

ES6 Classes

- ES6 introduced new syntax to make class-based inheritance in JavaScript cleaner:
  - note that this class mechanism is still using the *existing* JS delegation mechanism
    - not traditional copy-based inheritance
  - *pros*:
    - fewer references to `.prototype`
    - new, more natural `extends` keyword
      - can extend on natives, such as arrays or error objects
    - provides `super` for relative polymophism
    - `constructor` method
  - *cons*:
    - no way to declare class member properties (only methods)
      - requires `.prototype` syntax
    - accidental shadowing can occur
    - some issues with dynamic `super` bindings

ES6 class example:

```
class Widget {
  constructor(width, height) {
    this.width = width || 50;
    this.height = height || 50;
    this.$elem = null;
  }
  render($where) {
    if (this.$elem) {
      this.$elem.css({
        width: this.width + "px",
        height: this.height + "px"
      }).appendTo($where);
    }
  }
}

class Button extends Widget {
  constructor(width, height, label) {
    super(width, height);
    this.label = label || "Default";
    this.$elem  = $("<button>").text(this.label);
  }
```

```
  render($where) {
    super($where);
    this.$elem.click(this.onClick.bind(this));
  }
  onClick(evt) {
    ...
  }
}
```

Asynchronous JavaScript

- **asynchronous** programming is an important of JavaScript:
  - programs are written in *chunks*, some of which will execute *now* and some of which will execute *later*:
    - code that should be executed later introduces *asynchrony* into the program
    - eg. making an AJAX request, or even I/O like `console.log` may be deferred and completed asynchronously
  - there are different ways to specify what JS code should run later, eg. on *completion* of another event:
    - callbacks, promises, generators, etc.
- a key JavaScript feature is the **event loop**:
  - JS itself does not actually have a *direct* notion of asynchrony
    - the event loop handles executing different chunks of the program over time
  - different *handlers* can be registered for certain events, so that these handlers run when the events occur:
    - unlike normal *synchronous* code, these events can occur *asynchronously* ie. at any time
    - eg. when a `setTimeout` timer fires, it places the callback into the event loop
      - thus `setTimeout` timers may not fire with perfect accuracy depending on the current queue of events on the loop
  - different language structures used for asynchronous functions include callbacks, promises, async/await, and generators
  - eg. a common functionality that is handled using asynchronous functions are AJAX requests:
    - **Async JS and XML (AJAX)** requests communicate with a server using an HTTP request, without having to reload the current page
    - ie. retrieving XML (or more recently, JSON) data asynchronously using JS
  - note that JavaScript (and the event loop) runs on a *single* thread:
    - so *functions* are executed atomically, ie. *run-to-completion* behavior
    - however there is still nondeterminism in the *ordering* of asynchronous events:
      - eg. two AJAX requests may each complete and call their callbacks at arbitrary times with respect to the other
      - conditional completion checks or latches can be used to make such behavior deterministic
  - this single threaded event loop still offers **concurrency**:

    * although only one event can be handled at a time, sequentially, on the event loop, multiple *tasks* or *"processes"* may simultaneously be pushing events onto the event loop
    * these events may become *interleaved* with one another
    * this allows for concurrency ie. task-level parallelism, as opposed to operation-level parallelism through multithreading
- ES6 added a new concept layered on top of the event loop queue called the **Job queue**:
  - this queue is an additional event queue, but has higher priority than the event loop queue

Callbacks

- using **callbacks** is the most basic method of writing asynchronous event handlers:
  - *pros*:
    * simple, making use of continuation passing style (CPS)
    * used in other JS language structures, eg. *synchronous* functional callbacks such as `forEach`, `map`, `filter`, etc.
  - *cons*:
    * can quickly lead to *"callback hell"* or the *"pyramid of doom"*, where callbacks that should be executed in succession become deeply nested and cluttered
      · in addition to the cluttered nesting, callback hell has the issue of *hardcoded* brittle behavior due to the difficulty of tracing the possible paths of execution
      · another issue of inversion of control since we are delegating control to usually a third-party library, and only specifying a callback
      · leads to many special cases to handle, eg. callback may be called too early, too late, or multiple times, or callback may swallow errors, etc.
    * ie. callbacks express asynchronous flow in a nonlinear, nonsequential way
  - some possible extensions on callbacks that help with some issues:
    * *split* callbacks for success and error
    * *error-first* callback style where the callback accepts an error argument as the first argument
    * always make sure callbacks are predictably asynchronous

Using callbacks in vanilla JS and jQuery:

```
// vanilla JS request:
var http = new XMLHttpRequest();
http.onreadystatechange = function() { // callback
  // 4 different ready states while request is loading
  if (http.readyState == 4 && http.status == 200) {
    console.log(JSON.parse(http.response));
  }
};
http.open("GET", "data/tweets", true);
http.send();

// jQuery alt:
$.get("data/tweets", function(tweets) { // callback
  console.log(tweets);
});
```

Illustrating callback hell:

```
$.get("data/topTweets", function(topData) { // callback
  $.get("data/tweets/" + topData[0].id, function(tweet) {
    $.get("data/users/" + tweet.userId, function(userData) {
      console.log(userData);
    })
  })
});
```

Promises

- **promises** are an alternative to callbacks for asynchronous programming, and an easily repeatable mechanism for encapsulating future values:
  - promises are *objects* that represent actions that haven't yet finished
  - promises are then *chained* using the `.then` property in order to specify how data should be handled after it is finished retrieving
    * the `.catch` property is used to handle errors, at *any* point in the promise chain, even in callbacks
      · alternatively, `.then` also takes a second argument to handle *rejection* from the chained promise
      · ie. `.then` takes `fulfilled` and `rejected` callbacks as arguments
    * control is uninverted from the callback pattern, since the async function is unaware of other code *subscribing* to its events

    * instead, the control goes back to the calling code when the event handlers are run
  - once a promise has been resolved, it becomes *immutable*:
    * this makes it safe to pass the value around, ie. if multiple parties are observing the resolution of a promise, one party cannot affect another party's ability to observe the resolution
    * important aspect of promise design
  - *pros*:
    * sequential callbacks are no longer deeply nested
    * promises can be easily chained together asynchronously
    * elegant error catching
    * easy to handle create and use multiple promises
    * *uninverts* the inversion of control since we are not handing off the continuation of the program to a third party
  - *cons*:
    * syntax is still a little unnatural, is there a way to make async code look more similar to synchronous code?
    * still some issues with error handling, ie. no external way to guarantee to observe all errors
      - eg. simply catching the end of a promise chain may not catch all errors since any step in the chain may perform error handling already
    * promises only have a single fullfilment value
      - usually solved with a value wrapper, or splitting values into dfferent promises
    * promises can only be resolved once, eg. what about events or streams of data?
    * promises are uncancelable
  - note that the ES6 promise implementation uses duck typing to identify promises:
    * a **thenable** is any object with a `.then` method
    * thenables will be treated with special promise rules, even if they were not intended to be treated as a promise
- promise patterns:
  - `Promise.all` is used to initialize *multiple* asynchronous requests at once:
    * order doesn't matter, just wait on all the async tasks to finish
    * takes an array of promises, and returns a promise that fullfils to an array of each fullfilment message of the passed promises, in order
      - main returned promise is fulfilled only if all the constiutent promises are also fulfilled
  - `Promise.race` acts as a latch pattern or promises:
    * takes an array of promises, but only resolves with a single value of the first resolved promise

        - an empty array will *never* resolve
      * also rejects if any promise resolution is a rejection
    - also `Promise.none` , `Promise.any` , `Promise.first` , `Promise.last`

Implementing a promise over a vanilla JS callback:

```
function get(url) {
  return new Promise(function(resolve, reject){
    // resolve applies to the .then function,
    // while reject should fall to the .catch function (passing the error code)
    var xhhtp = new XMLHttpRequest();
    xhttp.open("GET", url, true);
    xhttp.onload = function() {
      if (xhttp.status == 200) {
        resolve(JSON.parse(xhttp.response));
      } else {
        reject(xhttp.statusText);
      }
    };
    xhttp.onerror = function() {
      rejext(xhttp.statusText);
    };
    xhttp.send();
  });
}
```

Using promises:

```
get("data/topTweets")
  .then(function(topData) {
  return get("data/tweets/" + topData[0].id);
}).then(function(tweet) { // chaining promises
  return get("data/users/" + tweet.userId);
}).then(function(userData) {
  console.log(userData);
}).catch(function(error) {
  console.log(error);
});
```

Using `Promise.all` to wait for multiple promises *concurrently*:

```
const p1 = Promise.resolve("hello");
const p2 = 10;
const p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, true);
```

```
});
const p4 = new Promise((resolve, reject) => {
  setTimeout(resolve, 3000, 'goodbye');
});

Promise.all([p1, p2, p3]).then(values =>
  console.log(values);
); // runs all the promises, values is ["hello", 10, true, "goodbye"] after 3 sec
```

- addressing the previous issues of trust from callbacks:
  1. callback called too early:
     - ie. task finishes synchronously and sometimes asynchronously, leading to race conditions
     - promises by definition are not susceptible to this, since even immediately resolved promises *cannot* be observed synchronously
     - ie. the callback provided to `.then` is always called asynchronously, even if the promise is already resolved
  2. callback called too late:
     - when a promise is resolved, all registered callbacks on it will be called in *order*
     - nothing happening inside those callbacks can delay the calling of the other callbacks
  3. callback never called:
     - *nothing* can prevent a promise from notifying its resolution
     - even if a promise never gets resolved, there is a provided `race` mechanic that prevents the promise from hanging indefinitely
  4. callback called multiple times:
     - promises can only be resolved once, and become immutable
  5. failing to pass along parameters:
     - promises still resolve even when called with no explicit value
       * value is resolved as `undefined`
  6. errors and exceptions becoming swallowed:
     - if an exception occurs while a promise is being resolved, the exception is caught and forces the promise to become rejected
     - ie. promises turn even JS exceptions into asynchronous behavior, whereas they were previously caused a synchronous reaction
     - however, note that if there is an exception in the registered callback, it is *not* caught by the rejection handler, since promises are immutable once resolved
- it is important to note that promises do not *replace* callbacks, instead we pass a callback onto a promise:
  - but how can we guarantee trust and that the promise itself is really a

genuine promise?
  - ES6 promises also provides `Promise.resolve` :
    1. passing an immediate, non-promise, non-thenable value to `Promise.resolve` returns a promise that fulfills to that value
    2. passing a genuine promise to `Promise.resolve` simply returns the same promise
    3. passing a non-promise, thenable value to `Promise.resolve` will unwrap the value until a concrete, final, non-promise value is extracted
    * thus the return value from `Promise.resolve` is always a real promise,
    * way to generate trust
  - `Promise.reject` cretes an already rejected promise, without unwrapping

Generators

- **generators** are functions that can be *paused* and *resumed*:
  - a newer ES6 feature
    * generators are originally from Python
  - typically used for **lazy evaluation**
  - breaks from the ordinary JS *run-to-completion* behavior
  - the `yield` keyword can be used for bidirectional message passing
    * can also be used for obtaining synchronous-like return values from async function calls
    * as well as synchronously catching errors from those async calls
  - can also `throw` errors into as well as from generators
  - using generators is another method for expression asynchronous flow control
- note that to run a generator, an **iterator** is first created:
  - thus multiple instances of the same generator can run at the same time
  - iterator aside:
    * an iterator is an interface for stepping through a series of values from a producer
    * calls `next` each time you want the next value
      · `next` returns `{ done, value }`
    * the `for..of` loop can be used to consume a standard iterator
    * an **iterable** is an object that contains an iterator
      · `iterable[Symbol.iterator]()` creates the iterator
    * arrays have default iterators that go over their values
  - note that a generator is not technically an iterable, executing a generator returns an iterator

- its iterator is also an iterable
- thus we can use a `for..of` loop with a generator as `for (var v of generator()) ...`
- can exit from a generator using `it.return(val)`

- *transpilation* of ES6 generators to pre-ES6 code can be done with a closure-based solution that keeps track of state of the *"generator"*:
  - each state represents the different generator states between `yield` calls

Generator example:

```
function* gen(index){
  while (index < 2)
    yield index++;
  return 42;
}

var it = gen(0);    // construct an iterator
var x = it.next(); // object x has attributes value 0 and boolean done
var y = it.next(); // y has value 1 and done false
var z = it.next(); // z has value 42 and done true
```

Using `yield` for message passing with generators:

```
function* foo(x) {
  var y = x * (yield "Hello"); // two-way message passing!
  return y;
}

var it = foo(6);
var res = it.next();
res.value; // gives Hello
res = it.next(7); // pass 7 in to yield
res.value; // gives 42
```

Hardwiring iterator control for generators with promises:

```
function* main() {
  try {
    var text = yield get(url);
    console.log(text);
  } catch (err) {
    console.log(err);
  }
}
```

```
var it = main();
var promise = it.next().value;
promise.then(
  function(text) {
    it.next(text);
  },
  function(err) {
    it.throw(err);
  }
);
```

Generators with promises using a generator runner:

```
function genWrap(generator){
  var gen = generator();
  function handle(yielded){
    if(!yielded.done){
      yielded.value.then(
        function(data){
          return handle(gen.next(data));
        },
        function(err) {
          gen.throw(err);
        }
      );
    }
  }
  return handle(gen.next());
}

genWrap(function*(){
  var top   = yield get("data/topTweets");
  var tweet = yield get("data/tweets/" + top[0].id);
  var user  = yield get("data/users/" + tweet.userId);
  console.log(user);
});
```

Concurrency with generators:

```
genWrap(function*(){
  var p1 = get(url1);
  var p2 = get(url2);

  // p1 and p2 are made in parallel
```

```
  var r1 = yield p1;
  var r2 = yield p2;

  // more parallel requests
  var rest = yield Promise.all([...]);

  // p3 gated until after all previous promises complete
  var r3 = yield get(...);
  console.log(r3);
});
```

- the keyword `yield*` performs yield-delegation:
  - this allows generators call another generator, and integrate into each other
    - allows for cleaner, more modularized generator code
    - delegation also allows for more complex message passing and even recursive behavior with generators
  - ie. transfers or *delegates* the iterator control over to another iterable (not necessarily just another generator)

Yield-delegation example:

```
function* foo() {
  yield 2;
  yield 3;
}

function* bar() {
  yield 1;
  yield* foo(); // yield-delegation here
  yield 4;
}

var it = bar();
it.next().value; // 1
it.next().value; // 2
it.next().value; // 3
it.next().value; // 4
```

Async/Await

- **async/await** is a modern syntactical sugar for promises:

- ie. an *syntactical* extension on promises, still using promises under the surface
  * essentially using generators, with even less clutter
- adopted in other languages, such as Python's `asyncio` library
- the `await` keyword awaits the *resolution* of a promise
  * can only be used within an `async` function
- *pros*:
  * cleaner code than promises, async code that *looks* synchronous
- *cons*:
  * a `try-catch` block is the only way to catch errors

Using async/await:

```
async function getTopUser() {
  try {
    const topData  = await get("data/topTweets"); // alternative to .then syntax
    const topTweet = await get("data/tweets/" + topData[0].id);
    const userData = await get("data/users/" + topTweet.userId);
    console.log(userData);
  } catch (error) {
    console.log(error);
  }
}
```

Using the DOM

- the **document object model (DOM)** is the data representation of the objects that comprise the structure and content of a document on the web:
  - ie. a programming *interface* for HTML documents that represents the page as nodes and objects
  - whenever a script is created, the API for `document` or `window` elements can be used to manipulate the document

Data Types

- the `Document` type corresponds to the root document object itself:
  - properties: `body, fonts, images, cookie, location`
  - methods: `createElement, getElements..., querySelector, addEventListener`
- every object within a document is of type `Node` of some kind:
  - eg. an element, text, or attribute node
  - properties: `nodeType, nodeValue, textContent`
  - linking properties: `firstChild, nextSibling, childNodes`
    * `parentNode, parentElement`
    * note that `childNodes` will contain *all* nodes, including text or attributes nodes
  - methods: `appendChild, removeChild, replaceChild, hasChildNodes`
- the `ParentNode` type contains methods and properties common to all node objects with children:
  - eg. for element or documents objects, returned by `node.parentNode`
  - properties: `childElementCount, children, firstElementChild`
    * note that the `children` property returns an `HTMLCollection` with *only* element children, rather than the `NodeList` returned by `childNodes`
      · in addition, the `HTMLCollection` is a *live* object that is automatically updated when the underlying object is changed
  - methods: `append, querySelector, replaceChildren`

Recursing through child nodes:

```
function eachNode(root, cb) {
  if (!cb) { // just return node list
    const nodes = [];
    eachNode(root, function(node) {
      nodes.push(node);
```

```
    });
    return nodes;
  }

  if (!callback(root)) {
    return false;
  }

  if (root.hasChildNodes()) {
    const nodes = rootNode.childNodes;
    for (let i = 0; i < nodes.length; i++) {
      if (!eachNode(nodes[i], cb)) {
        return;
      }
    }
  }
}
```

- the `Element` type is based on nodes, and refers to a node of type element returned by the DOM API
  - inherits from its node interface as well as implementing a more advanced element interface
  - properties: `attributes, classList, className, innerHtml, style` and many more
  - methods: `addEventListener, getElements..., scroll` and many more
  - in an HTML document, the `HTMLElement` further extends this type:
    * offers methods such as `blur, click, focus`
- a `NodeList` is an array of elements:
  - eg. array returned by `querySelectorAll`
    * note that `getElements...` returns an `HTMLCollection` instead of a node list
  - items can be accessed via `list[idx]` or `list.item(idx)`
- a `NamedNodeMap` is like an array of nodes, but items are accesed by name *or* index
- `Attribute` nodes are object references that expose a special interface for attributes:
  - nodes just like elements, but more rarely used

ES6 Features

New Syntax

- block scoping:
  - introduced `let` and `const` for block scoping
    - note the unique redeclaraton of `let` variables in a loop, useful for closures
    - note that `const` freezes the assignment of a value, not the value itself
  - as well as the temporal deadzone for accessing them early
- the *spread* or rest operator `...` :
  - when used in front of any iterable, it *spreads* it out into individual values
    - can also be used to *gather* a set of values into an array, usually in function arguments
  - used in different contexts such as function arguments, inside another array declaration, etc.
  - eg. `foo(...[1,2,3])` is a replacement for `foo.apply(null, [1,2,3])`
  - eg. `function foo(...args)` gathers all the arguments into the array `args`
- default parameter values for functions:
  - eg. `function foo(x = 11, y = 31)`
  - default values can be more than simple values:
    - can be any valid expression, even a function call or IIFE
- *destructuring* or structured assignment:
  - new dedicated syntax for array and object destructuring
  - eg. `var [a,b,c] = foo()` , `var {x:a, y:b, z:c} = bar()` , or also `var {x,y,z} = bar()`
    - note that when destructuring, the object literal follows the `<target>: <source>` pattern rather than the opposite for declarations
  - extensions on destructuring:
    - destructuring returns the right hand value, so destructuring assignments can be *chained* together
    - values can be discarded eg. `var [,b] = [1,2]` :
      - destructuring missing values will become undefined
    - the spread operator can be used to gather together elements, eg. `var [a, ...rest] = [1,2,3]`
    - can also use `=` to set default value assignment

  - destructuring can also be used with parameter assignment in functions

Using expressions in destructuring:

```
var foo = [1,2,3];
var bar = {x:4, y:5, z:6};

var key = "x", o = {}, a = [];
({[key]: o[key]} = bar); // quotes needed to prevent from parsing {} as block
console.log(o.x); // prints 4

({x: a[0], y: a[1], z: a[2]} = bar);
console.log(a);   // prints [4,5,6]

[o.a, o.b, o.c] = foo;
console.log(o.a, o.b, o.c); // prints 1 2 3

var x = 10, y = 20;
[y, x] = [x, y];
console.log(x, y); // prints 20 10
```

Chaining destructuring assignments:

```
var a, b, c, x, y, z;

[a, b] = [c] = foo;
({x} = {y, z} = bar);

console.log(a, b, c); // prints 1 2 1
console.log(x, y, z); // prints 4 5 6
```

Default value assignment:

```
var [a=3, b=4, c=5, d=6] = [1,2,3];
console.log(a, b, c, d);  // prints 1 2 3 6

var {x, y, z, w: WW = 20} = {x:4, y:5, z:6};
console.log(x, y, z, WW); // prints 4 5 6 20
```

Destructuring parameter gotcha:

```
function foo({x = 10} = {}, {y} = {y:10}) {
  console.log(x, y);
}

foo();                 // prints 10 10
```

```
foo({}, undefined); // prints 10 10
foo({}, {});        // prints 10 undefined
foo(undefined, {}); // prints 10 undefined
foo({x:2}, {y:3});  // prints 2 3
```

- object literal extensions:
  - *concise* properties:
    - to define a property in an object that is the same name as an identifier, can shorten from `x: x` to just `x`
    - similarly for methods (and generators) in objects, can shorten from `x: function() {...}` to just `x() {...}`
      - note however that this makes the function expression *anonymous*, which may have issues with recursion
      - in such cases, it is safer to write out the full expression `x: function x() {...}`
  - computed property name:
    - an object literal definition can use an expression to compute the assigned property name
- objects and prototypes:
  - new `Object.setPrototypeOf`, and new `super`
    - note that `super` can only be used in concise methods
- template literals ie. *interpolated* strings:
  - similar to f-strings in Python
  - interpolated strings are still type string, except they act like IIFEs in that they are automatically evaluated *inline*
  - note that any valid expression can appear in an interpolated expression
  - eg. `` `hello ${name}!` ``
- *tagged* template literals:
  - a special function call without parentheses
  - the function receives:
    - a first argument of all the plain strings (between interpolated expressions)
    - the remaining arguments that are the results of the evaluated interpolated expressions

Example tagged template literals:

```
function foo(strings, ...values) {
  console.log(strings, values);
}

var desc = "awesome";
foo`Everything is ${desc}!`; // prints ["everything is ", "!"] ["awesome"]
```

```
// example function to collapse a template literal
function tag(strings, ...values) {
  return strings.reduce(function(s, v, idx) {
    return s + (idx > 0 ? values[idx-1] : "") + v;
  }, "");
}
tag`Everything is ${desc}!`; // gives "Everything is awesome!"
```

- arrow functions:
  - new more concise syntax for arrow expressions with the *fat arrow*
  - lexically binds `this`
    - replaces the `var self = this` and `.bind(this)` fixes to bind `this`
  - note that all arrow functions are anonymous function expressions
- `for..of` loops that loops over the values produced by an iterator
  - standard builtin types that provide iterables are arrays, strings, generators, and collections
- also, extended Unicode support, more tricks for regular expressions, and a new primitive `symbol` type

Organization

- **iterators** are structured patterns from producing information from a source, one-at-a-time:
  - the `Iterator` interface requires the `next` method that returns an `IteratorResult`
    - as well as optional `return` and `throw` methods to end production of values by iterator
  - `IteratorResult` has two required properties `value` (`undefined` if missing) and boolean `done`
    - typically the last value still has `done: false` , and `done: true` signals completion after all relevant values are returned
    - calling `next` on an exhausted iterator is not an error, will simply return the same completed `IteratorResult`
  - an `Iterable` has the `@@iterator` method that produces an iterator
  - consuming iterables:
    - the `for..of` loop
    - spread operator
    - array destructuring
  - ES6 also introduced generators, as seen previously

Custom fibonacci iterator:

```
var Fib = {
  [Symbol.iterator]() {
    var n1 = 1, n2 = 1;
    return {
      // this makes the iterator an iterable as well
      [Symbol.iterator]() { return this; },
      next() {
        var current = n2;
        n2 = n1;
        n1 = n1 + current;
        return { value: current, done: false };
      },
      return(v) {
        console.log("fib sequence stopped");
        return { value: v, done: true };
      }
    }
  }
}

for (var v of Fib) {
  console.log(v);
  if (v > 20) break;
} // prints 1 1 2 3 5 8 13 21
  //        fib sequence stopped
```

- ES6 modules:
  - uses `import` and `export` :
    * `export` exports the name *bindings* of variables:
      · that is, if a value is changed inside a module after its export, the imported binding will *resolve* to the current value
      · default and named exports
      · anything not exported stays *private* within the scope of the module
    * `import` imports from another module:
      · all imported bindings are *immutable* and read-only
      · note that declarations as a result of importing are also hoisted
      · ES6 can solve circular `import` dependencies
  - file-based, ie. one module per file
  - statically defined API for each module
  - singletons, eg. importing a module gets a reference to one centralized instance

   - aims to replace traditional module patterns eg. AMD, UMD, and CommonJS

Traditional module patterns:

```
// asynchronous module definition (AMD), eg. RequireJS:

// define(dependencies, callback), RequireJS handles loading dependencies
define(['jquery', 'underscore'], function($, _) {
  function a() {...}; // private method, not exposed
  function b() {...};
  function c() {...};

  // exposed API
  return { b: b, c: c };
})

// CommonJS, similar to NodeJS modules:
var $ = require('jquery');
var _ = require('underscore');

function a() {...};
function b() {...};
function c() {...};

module.exports = { b: b, c: c };

// universal module definition (UMD), both AMD and CommonJS compatible:
(function(root, factory) {
  if (typeof define === 'function' && define.amd) {
    define(['jquery', 'underscore'], factory);
  } else if (typeof exports === 'object') {
    module.exports = factory(require('jquery'), require('underscore'));
  } else {
    // browser globals
    root.returnExports = factory(root.jQuery, root._);
  }
}(this, function($, _) {
  function a() {...};
  function b() {...};
  function c() {...};

  return { b: b, c: c };
}));
```

ES6 exporting:

```
function foo() {...}
var bar = 42;
export var baz = [1, 2, 3];
export { foo as qaz, bar };
export { foo as FOO, bar as BAR } from "qux"; // re-export
```

ES6 default export nuances:

```
function foo() {...}

export default foo; // exports binding to a function *expression*
                    // so if foo is rebound, import reveals the *original* function
// vs.
export { foo as default }; // exports binding to foo *identifier*
                           // import is updated if foo is rebound
```

ES6 importing:

```
import foo, { bar, baz as BAZ } from "foo";
import * as qux from "qux"; // import all, default is qux.default
```

Collections

- JavaScript typed arrays provide structured access to binary data using array-like semantics:
  - the *type* refers to the *view* layered on top of an `ArrayBuffer` ie. a buffer of bits
  - different views eg. `Uint8Array` , `Int16Array` , `Float32Array`
  - a single buffer can have multiple views, and a view can also be set at a certain offset or length
  - eg. `var buf = new ArrayBuffer(32)` creates a buffer
  - eg. `var arr = new Uint16Array(buf)` creates a view over that buffer
- ES6 maps can use a non-string value as a key, *unlike* normal objects:
  - use `get` , `set` , and `delete` for mutating
  - supports `size` , `includes` , `has` methods
    - and `values` , `keys` , `entries` iterator methods
  - `WeakMap` is a map variation that only takes objects as keys:
    - when the object that is a key is garbage collected, the entry is alwo removed

- ES6 sets are collection of unique values:
  - duplicates are ignored
  - similar api to maps:
    - with `add` instead of `set`
    - no `get` , only `has`
  - a `WeakSet` holds its values (only objects) weakly

API Additions

- arrays:
  - `Array.of` is a an alternative constructor that avoids the default `Array` constructor gotcha of creating an empty slots array when passed a single number
    - eg. `Array.of(3)` creates an array with element 3, while `Array(3)` creates an array with `length` 3, but empty slots
  - `Array.from` replaces `Array.prototype.slice.call` for duplicating arrays or transforming array-likes into arrays
    - `Array.from` also avoids empty slots
    - also takes a callback to transform each value
  - `copyWithin` copies a portion of an array to another location in the same array
  - `fill` fills an existing array entirely or partially with a specified value
  - `find` and `findIndex` give more flexibilty and control over the matching logic offered by `indexOf`
- objects:
  - `Object.is` is similar to `===` , except it correctly distinguishes `NaN -0 +0`
  - `Object.getOwnPropertySymbols` , `Object.setPrototypeOf` , `Object.assign`
- numbers:
  - many new mathematic utilities, eg. `cosh` , `hypot` , `trunc`
  - `Number.EPSILON` , `Number.MAX_SAFE_INTEGER` , `Number.MIN_SAFE_INTEGER`
  - `Number.isNaN` , `Number.isFinite` , and `Number.isInteger`
- strings:
  - unicode aware string operators, eg. `String.fromCodePoint` , `codePointAt` , `normalize`
  - `String.raw` tag function to get raw strings without escape sequence processing
  - `repeat` to use repeat strings
  - `startsWith` , `endsWith` , `includes`