JavaScript

Thilan Tran

Summer 2020

Contents

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 surround-

      ing 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
  - closuers 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);
}
```

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` , and `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 enlcosing 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); };
```

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 builin 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 (a) => {
     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
```

Object Prototypes

- `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

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, configu` 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
        * 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 property 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`
- 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
```

Classes

- 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
```

Asynchronous JavaScript

- a key JavaScript feature is the **event loop**:
  - 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
  - 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
- 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"*, where callbacks that should be executed in succession become deeply nested and cluttered

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** are an alternative to callbacks for asynchronous programming:
  - 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
  - `Promise.all` is used to initialize *multiple* asynchronous requests at once
  - *pros*:
    * sequential callbacks are no longer deeply nested
    * elegant error catching
    * easy to handle create and use multiple promises
  - *cons*:
    * syntax is still a little unnatural, is there a way to make async code look more similar to synchronous code?

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
```

- **async/await** is a modern syntactical sugar for promises:
  - ie. an *syntactical* extension on promises, still using promises under the surface
  - 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 **generators** is another less common method for asynchronous callbacks:
  - generators are functions that can be *paused* and *resumed*
    - generators are originally from Python
    - typically used for **lazy evaluation**
  - a newer ES6 feature

Generators in JS:

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

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

Async requests using generators:

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