Overview

- Scoping of variables.
- Object model.
- Functions are objects.

Variable Declarations

- Variables can be declared using `var` or the newer `let`. Note that declaration is simply `var x = 1` or `let x = 1`; there is no type as js variables do not have types.
- Constant variables (cannot be assigned after initialization) are declared `const`. Note that if a `const` variable contains an object, then the insides of that object can be changed.
- Every js system has an implicit global object: `window` in a browser, the current module in nodejs. If an undeclared variable is assigned to, then it is created in this global object. Can be avoided using `"use strict"`.
- Variables have lexical scope; i.e. their scope is limited to the syntactic construct within which they are declared.

Variable Hoisting

All variables declared using `var` are implicitly hoisted as though they were declared at the start of the containing function. (Note: logs edited for readability).

```
> var x = 3
undefined
> function f(a) {
    var y = 5;
    if (a > 1) {
      var x = y;
    }
    return x + y;
  }
undefined
> f(2)
10
>
```

Variable Hoisting Effect

The `var x` declaration in the previous function is hoisted to the start of `f()`. Hence the scope of all variables declared using `var` within a function is the **entire function**, irrespective of the point where the variable was actually declared.

```
function f(a) {
  var x;
  var y = 5;
  if (a > 1) {
    x = y;
  }
  return x + y;
}
```

This behavior can be prevented by using `let`.

Declaring Variables Using let

```
> var x = 3;
undefined
> function f_let(a) {
    let y = 5;
    if (a > 1) {
      let x = y;
    }
    return x + y;
  }
undefined
> f_let(2)
8
```

Behavior of `let` has fewer surprised; prefer `let` over `var` in new code.

Surprising Effects of Variable Hoisting

```
> var x = 1
undefined
> function f(a) {
  //var x effectively declared here
    x = 2;
    if (a) { var x = 3 } //declaration hoisted
    return x;
  }
undefined
> f(1)
3
> f(0)
2
> x
1
>
```

Unsurprising Effects using let

```
> let x = 1
> function f(a) {
    x = 2;
    if (a) { let x = 3 } //{...} block effectively a NOP
    return x;
  }
> x
1
> f(1)
2
> x
2
> f(0)
2
> x
2
```

Temporal Dead Zones using let

A `let` declaration takes effect at the start of the block in which it occurs. Leads to **temporal dead zones**.

```
> a = 1
1
> function f() {
    //let a at this point
    let b = a + 2; //not external a
    let a = 5;
    return b + a;
  }
undefined
> f()
ReferenceError: a is not defined
    at f (repl:1:24)
>
```

Declarations

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

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

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

Nested Functions and Closures

- A function can include nested function definitions.
- A nested function can include references to variables declared not within itself but in its enclosing function; i.e. it has a referencing environment.
- A **closure** captures both the code of a function and its referencing environment.
- In general, JS functions are always closures which capture their referencing environment.
- Can use closures to get stronger information hiding than that provided by objects.

Hiding Instance Variables: Bank Account

```
function Account(balance) {
  return {
    deposit: amount => balance += amount,
    withdraw: amount => balance -= amount,
    inquire: () => balance,
  };
}

a1 = new Account(100);
a2 = new Account(100);
a1.deposit(20);
a2.withdraw(20);
console.log(`a1: ${a1.inquire()}`);
console.log(`a2: ${a2.inquire()}`);
```

Bank Account Log

```
$ nodejs account.js
a1: 120
a2: 80
$
```

Object Basics

An object is merely a named collection of name-value pairs (which include functions). Values are referred to as object **properties**.

```
> x = { a: 9 }   //Object literal notation
{ a: 9 }
> x.a
9
> x = { a: 9,    //anon function is value for f
        f: function(a, b) { return a + b; } }
{ a: 9, f: [Function: f] }
> x.f(3, 5)
8
> x = 'a'
'a'
> { [x]: 42 }   //variable as name.
{ a: 42 }
>
```

Motivating Example for Prototypes: Complex Numbers

```
const c1 = {
  x: 1,
  y: 1,
  toString: function() {
    return `${this.x} + ${this.y}i`
  },
  magnitude: function() {
    return Math.sqrt(this.x*this.x + this.y*this.y);
  }
}
```

Motivating Example for Prototypes: Complex Numbers Continued

```
const c2 = {
  x: 3,
  y: 4,
  toString: function() {
    return `${this.x} + ${this.y}i`
  },
  magnitude: function() {
    return Math.sqrt(this.x*this.x + this.y*this.y);
  }
}

console.log(`${c1.toString()}: ${c1.magnitude()}`);
console.log(`${c2.toString()}: ${c2.magnitude()}`);
```

Motivating Example for Prototypes: Complex Numbers Continued

```
$ nodejs ./complex1.js
1 + 1i: 1.4142135623730951
3 + 4i: 5
```

Note that each complex number has its own copy of the `toString()` and `magnitude()` functions.

Using a Prototype Object to Hold Common Functions

```
complexFns = {
  toString: function() {
    return `${this.x} + ${this.y}i`
  },
  magnitude: function() {
    return Math.sqrt(this.x*this.x + this.y*this.y);
  }
}
```

Using a Prototype Object to Hold Common Functions: Continued

```
//use complexFns as prototype for c1
const c1 = Object.create(complexFns);
c1.x = 1; c1.y = 1;

//use complexFns as prototype for c2
const c2 = Object.create(complexFns);
c2.x = 3; c2.y = 4;

console.log(`${c1.toString()}: ${c1.magnitude()}`);
console.log(`${c2.toString()}: ${c2.magnitude()}`);
```

Prototype Chains

- Each object has an internal `[[Prototype]]` property.
- When looking up a property, the property is first looked for in the object; if not found then it is looked for in the object's prototype; if not found there, it is looked for in the object's prototype's prototype. The lookup continues up the prototype chain until the property is found or the prototype is `null`.
- Note that the prototype chain is only used for property lookup. When a property is assigned to, the assignment is made directly in the object; the prototype is not used at all.
- Prototype can be accessed using `Object.getPrototypeOf()` or `__proto__` property (supported by most browsers, being officially blessed by standards).

Constructors

- Every function has a `prototype` property. The `Function` constructor initializes it to something which looks like `{ constructor: this }`.
- Any function which is invoked preceeded by the `new` prefix operator is being used as a constructor.
- Within the body of a function invoked as a constructor, `this` refers to a newly created object instance with `[[prototype]]` internal property set to the `prototype` property of the function.
- Hence the `prototype` property of the function provides access to the prototype for the object instance; specifically, assigning to a property of the function `prototype` is equivalent to assigning to the object prototype.
- **By convention**, constructor names start with an uppercase letter.

Constructor Example

```
function Complex(x, y) {
  this.x = x;
  this.y = y;
}

Complex.prototype.toString = function() {
  return `${this.x} + ${this.y}i`
};
Complex.prototype.magnitude = function() {
  return Math.sqrt(this.x*this.x + this.y*this.y);
};

const c1 = new Complex(1, 1);
const c2 = new Complex(3, 4);

console.log(`${c1.toString()}: ${c1.magnitude()}`);
```

Constructor Return Value

- Normally a constructor function does **not** explicitly return a value. In that case, the return value is set to a reference to the newly created object.
- However, if the return value is explicitly set to an object (not a primitive), then that object is return'd from the constructor.
- Makes it possible to have constructor hide instance variables using closure.
- Makes it possible to have a constructor share instances by not returning the newly created instance.

Sharing Instances

Can use constructor return value to cache object instances to avoid creating a new instance unnecessarily.

```
const bigInstances = { };

function BigInstance(id, ...) { //... is pseudo-code, not rest
  if (bigInstances[id]) return bigInstances[id];
  //construct new instance as usual
  ...
  bigInstances[id] = this;
}
```

Inheritance

- We could implement classical inheritance using a pattern like `Child.prototype = new Parent()`. Hence `Child` will inherit properties from `Parent`.
- Note that we use `new Parent()`, rather than simply `Parent` as we do not want assignments to `Child.prototype` to affect `Parent`.
- Problematic in that we need to apply this pattern. Could wrap within a function `inherit()`, but still messy (see Crockford).
- Also, classical inheritance is generally problematic.

JavaScript Classes

- Added in es6 to make programmers coming in from other languages more comfortable.
- Create a new class using a class declaration.
- Create a new class using a class expression.
- Inheritance using `extends`.
- Static methods.
- Can extend builtin classes.
- Very thin layer around prototype-based inheritance. See this for tradeoffs.

Shapes Example

```
class Shape {
  constructor(x, y) {
    this.x = x; this.y = y;
  }
  static distance(s1, s2) {
    const xDiff = s1.x - s2.x;
    const yDiff = s1.y - s2.y;
    return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
  }
}
```

Shapes Example Continued

```
class Rect extends Shape {
  constructor(x, y, w, h) {
    super(x, y);
    this.width = w; this.height = h;
  }
  area() { return this.width*this.height; }
}

class Circle extends Shape {
  constructor(x, y, r) {
    super(x, y);
    this.radius = r;
  }
  area() { return Math.PI*this.radius*this.radius; }
}
```

Shapes Example Driver and Log

```
const shapes = [
  new Rect(3, 4, 5, 6),
  new Circle(0, 0, 1),
];

shapes.forEach((s) => console.log(s.x, s.y, s.area()));

console.log(Shape.distance(shapes[0], shapes[1]));
```

```
$ ./shapes.js
3 4 30
0 0 3.141592653589793
5
$
```

Class Constants

- Cannot define `const` within a `class`; following results in a syntax error:

```
class C {
  static const constant = 42;
}
```

- Use following pattern:

```
const C = 42;

class C {
  static get constant() { return C; }
}

console.log(C.constant);
```

Object Equality Examples

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

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

Arrays

Arrays are like objects except:

- It has an auto-maintained `length` property (always set to 1 greater than the largest array index).
- Arrays have their prototype set to `Array.prototype` (`Array.prototype` has its prototype set to `Object.prototype`, hence arrays inherit object methods).

Arrays vs Objects Examples

```
> a = []
[]
> o = {}
{}
> a.length
0
> o.length
undefined
> a[2] = 22
22
> a.length
3
> a[2]
22
```

Arrays vs Objects Examples Continued

```
> a.join('|')
'||22'
> a.length = 1 //truncates
1
> a[2]
undefined
> a.length
1
> a.constructor
[Function: Array]
> o.constructor
[Function: Object]
>
```

Property Attributes

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

Property Attributes Continued

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

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

Property Getter

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

Property Setter

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

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

Property Setter Continued

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

Enumerating Object Properties using for-in

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

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

Enumerating Example

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

Enumerating Example Continued

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

Another Enumerating Example

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

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

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

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

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

Monkey Patching to Add a New Function

Built-in types can be changed at runtime: **monkey-patching**.

```
> '  abcd  '.trim()
'abcd'
> '  abcd  '.ltrim()   //trim only on left
TypeError: "  abcd  ".ltrim is not a function
> String.prototype.ltrim =
    String.prototype.ltrim || //do not change
    function() { return this.replace(/^\s+/, ''); }
[Function]
> '  abcd  '.ltrim()
'abcd  '
>
```

Monkey Patching to Modify an Existing Function

```
> const oldFn = String.prototype.replace
undefined
> String.prototype.replace = function(a1, a2) {
    const v = oldFn.call(this, a1, a2);
    console.log(`${this}.replace(${a1},
${a2})=>${v}`);
    return v;
  }
[Function]
> 'aabcaca'.replace(/aa+/, 'x')
aabcaca.replace(/aa+/, x)=> xbcaca
'xbcaca'
> 'aabcaca'.replace(/a/g, (x, i) => String(i))
aabcaca.replace(/a/g, (x, i) => String(i))=> 12bc5c7
'12bc5c7'
>
```

Functions

- Traditional way of packaging up parameterized code for subsequent execution.
- Functions are **first-class**: need not have a name ("anonymous"), can be passed as parameters, returned as results, stored in data structure.
- Functions can be nested within one another.
- **Closures** preserve the referencing environment of a function.

Current Object Context

- During execution of a function, there is always an implicit object, referred to using **this**.
- **this** cannot be assigned to.
- Usually, **this** depends on how the function was called, it can be different for the same function during different calls.
- In global contexts (outside any function), **this** refers to the global object.
- When `strict` mode is not in effect, in a simple function call `fn()` without any receiver, **this** will refer to the global object.
- When `strict` mode is in effect, in a simple function call `fn()` without any receiver, **this** will refer to the value within the calling context.

Current Object Context Continued

- When called with a receiver using the dot notation, `this` refers to the receiver. So when `f()` is called using `o.f()`, the use of `this` within the call refers to `o`.
- It is possible to set the context dynamically using `apply()`, `call()` and `bind()`.
- When a function call is preceeded by the `new` operator, the call is treated as a call to a constructor function and `this` refers to the newly created object.

Using call()

Allows control of `this` when calling a function with a fixed number of arguments known at program writing time.

- All functions have a `call` property which allows the function to be called. Hence

```
let f = function(...) { ... };
let o = ...;
f.call(o, a1, a2); //like o.f(a1, a2)
                   //but o may not contain a f()
```

- Within function body, `this` will refer to `o`.
- `call` allows changing `this` only for functions defined using `function`, not for fat-arrow functions.
- In older JS, common idiom used to convert arguments to a real array is `let args = Array.prototype.slice.call(arguments)`

Example of Using `call` to control `this`

```
> obj1 = {
    x: 22,
    f: function(a, b) { return this.x*a + b; }
  }
> obj2 = { x: 42 }
{ x: 42 }
> obj1.f(2, 1)
45
> obj1.f.call(obj1, 2, 1) //obj1 as this.
45
> obj1.f.call(obj2, 2, 1) //obj2 as this
85
```

Using apply()

Allows control of `this` when calling a function with a number of arguments not known at program writing time.

- All functions have a `apply` property which allows the function to be called. Hence

```
let f = function(...) { ... };
let o = ...;
f.apply(o, [a1, a2]); //like o.f(a1, a2)
                      //but o may not contain a f()
```

- Within function body, `this` will refer to `o`.
- `apply` equivalent to `call` using spread operator; i.e. `f.apply(o, args)` is the same as `f.call(o, ...args)`.

All assertions in this-play.js pass:

```
//strict mode is not on

//top-level this in nodejs
assert(this === module.exports);

//plain function call without strict
function f1() {  return this; }
assert(f1() === global);

//plain function call with strict
function f2() {
  'use strict';
  return this;
}
assert(f2() === undefined);
```

Playing with this Continued

```
const obj1 = { a: 22, f: function() { return this; } }

//like plain function call
const g = obj1.f;
assert(g() === global);

//normal object call
assert(obj1.f() === obj1);

//change this using call
assert(obj1.f.call(obj1) === obj1);
assert(obj1.f.call(Array) === Array);
```

Using bind()

`bind()` fixes `this` for a particular function.

```
> x = 44
44
> a = { x: 2, getX: function() { return this.x; } }
> a.getX()
2
> f = a.getX() //a.x
2
> f = a.getX
> f() //global x
44
> b = { x: 42 }
> f = a.getX.bind(b)
> f() //b.x
42
```

Using bind() Continued

Can also be used to specify fixed values for some initial sequence of arguments. Can be used to implement currying.

```
> function sum(...args) {
    return args.reduce((acc, v) => acc + v);
  }
... ... undefined
> sum(1, 2, 3, 4, 5)
15
> add12 = sum.bind(null, 5, 7) //passing this as null
[Function: bound sum]
> add12(1, 2, 3, 4, 5)
27
>
```

Difference in `this` between `function` and Fat-Arrow

- Within a nested function defined using `function`, `this` refers to global object.
- Within a nested function defined using the fat-arrow notation, `this` refers to that in the containing function.

Difference in this between function and Fat-Arrow Example

```
> x = 22
22
> function Obj() { this.x = 42; }
> Obj.prototype.f = function() {
    return function() { return this.x; }
  }
> Obj.prototype.g = function() {
    return () => this.x;
  }
> obj = new Obj()
> obj.f()() //this refers to global obj
22
> obj.g()() //this refers to defn obj
42
>
```

Common Idiom Used for Workaround for function this

```
> Obj.prototype.h = function() {
    const that = this;
    return function() { return that.x; }
  }
[Function]
> obj.h()() //access enclosing this via that
42
> obj.f()() //unchanged
22
>
```

Immediately Invoked Function Expressions

- When a JS file is loaded into a browser, it may define identifiers in the global `window` object.
- Possible that these identifier definitions may clash with identifier definitions loaded from other JS files.
- **Immediately Invoked Function Expression** (IIFE) idiom uses closures to encapsulate code within a browser:

```
(function() { //anonymous function
  //ids defined here cannot be accessed from outside
  //code can manipulate browser
})();   //immediately invoke anon function
```

Nodejs wraps each module within a function wrapper:

```
(function(exports, require, module,
            __filename, __dirname) {
// Module code actually lives in here
});
```

and calls the above function with appropriate arguments.

Argument Checking

JavaScript does not require that the number of actual arguments match the number of formal parameters in the function declaration.

- If called with a fewer number of actuals, then the extra formals are `undefined`.
- If called with a greater number of actuals, then the extra actuals are ignored.

```
> function f(a, b, c) {
    return `a=${a}, b=${b}, c=${c}`;
  }
undefined
> f(1, 2)
'a=1, b=2, c=undefined'
> f(1, 2, 3, 4)
'a=1, b=2, c=3'
```

Function Arguments Pseudo-Array

- Within a function, the variable `arguments` acts like an array in terms of property `length` and array indexing.

```
> function f() {
    let z = []
    for (a of arguments) { z.push(a*2); }
    return z;
  }
undefined
> f(1, 2, 3)
[ 2, 4, 6 ]
```

- Unfortunately, `arguments` is only a pseudo-array and does not support the full set of array methods and properties.

Converting arguments to a Real Array

In newer versions of JS, `Array.from()` can be used to convert to an array:

```
> function f() {
    return arguments.reduce((acc, v) => acc*v);
  }
undefined
> f(1, 2, 3)
TypeError: arguments.reduce is not a function
...
> function g() {
    return Array.from(arguments) //real array
      .reduce((acc, v) => acc*v);
  }
undefined
> g(1, 2, 3)
6
```

ES6 Rest Parameters

If arguments are declared using ES6 `rest` syntax, then that variable provides a real `Array`:

```
> function f(...args) {
    return args.map(x => 2*x);
  }
undefined
> f(1, 2, 5)
[ 2, 4, 10 ]
```

Note that `...` is also used to *spread* array as separate arguments into function call or initialization/assignment RHS.

Functions Are Objects

```
> function add(a, b) { return a + b; }
undefined
> typeof add
'function'
> add.constructor
[Function: Function]
> add.x = 22
22
> add[42] = 'life'
'life'
> add(3, 5)
8
> add.x
22
> add[42]
'life'
```

Memoize function by caching return values in fn property.

```
function fib(n) {
  return (n <= 1) ? n : fib(n - 1) + fib(n - 2);
}

//memoizing fibonacci caches results
//in function property
function memo_fib(n) {
  memo_fib.memo = memo_fib.memo || {};
  if (memo_fib.memo[n] === undefined) {
    memo_fib.memo[n] =
      (n <= 1) ? n
      : memo_fib(n - 1) + memo_fib(n - 2);
  }
  return memo_fib.memo[n];
}
```

Function Properties for Memoization: Fibonacci Continued

```
const N = 45;
[fib, memo_fib].forEach(function(f) {
  console.time(f.name);
  console.log(`${f.name}(${N}) = ${f(N)}`);
  console.timeEnd(f.name);
});
```

Fibonacci Log

```
$ ./fib.js
fib(45) = 1134903170
fib: 10080.337ms
memo_fib(45) = 1134903170
memo_fib: 0.216ms
$
```

Defining Function Using Function() Constructor

Many dynamic languages allow converting strings into code. JavaScript supports this using `eval()` as well as via a `Function` constructor.

```
> x = max3 = new Function('a', 'b',
                          'return a > b ? a : b')
[Function: anonymous]
> max3(4, 3)
4
> x(4, 3)
4
> x.name
'anonymous'
> x.length
2
>
```