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
>
```

A Glimpse At Objects

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 }
>
```

Wrappers

The primitive types string, number, boolean can be wrapped as objects using constructors `new String()`, `new Number()`, `new Boolean()`. Wrapping and unwrapping are done automatically as needed.

```
> x = new Number(3) //use wrapper constr
[Number: 3]
> typeof x          //x is an object
'object'
> typeof 3
'number'
> x + 1             //automatically unwrapped
4
> x.a = 2           //object property assign
2
> x.a + x           //property + unwrap
5
```

Wrappers Continued

We can even define properties on primitive literals, but not of much use.

```
> 3.x = 22
3.x = 22
  ^^
SyntaxError: Invalid or unexpected token
> > 3.0.x = 22
22
> 3.0.x
undefined
>
```

Wrapper object automatically created, but since we do not retain a reference to it, we cannot access it.

Conversions

When used without `new`, `Number()`, `String()`, `Boolean()` can be used to explicitly convert between primitives. (JavaScript has very complex implicit conversion rules; more details later):

```
> Number('3')
3
> Number(false)
0
> Number(undefined)
NaN
> Number(null)
0
> Number(true)
1
> String(true)
'true'
> String(3+1)
'4'
```

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

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
$
```

Sharing Instances

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

```
const bigInstances = { };

function BigInstance(id, ...) {
  if (bigInstances[id]) return bigInstances[id];
  //construct new instance as usual
  ...
  bigInstances[id] = this;
}
```

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]
>
```

Mapping Arrays

The `map()` function returns a new array which is the result of calling its argument array on each element of the calling array.

```
> function times3(x) { return 3*x; }
undefined
> [1, 2, 3].map(times3)
[ 3, 6, 9 ]
> [1, 2, 3].map(x => 7*x);
[ 7, 14, 21 ]
> [7, 3, 2, 4].map(x => x % 2 === 0)
[ false, false, true, true ]
>
```

Reducing Arrays

The `reduce()` function using a function `f(accumulator, element)` to reduce an array to a single value.

```
> [1,2,3,4,5].reduce((acc, value) => acc + value)
15
> [1,2,3,4,5].reduce ((acc, value) => acc + value, 7 )
22
> [12].reduce((acc, value) => acc + value)
12
> > [].reduce((acc, value) => acc + value, 15)
15
> [].reduce((acc, value) => acc + value)
TypeError: Reduce of empty array with no initial value
...
```

Applying a Function to Each Array Element

`forEach()` applies function to each element. Like many other functions callback takes 3 arguments: elementValue, elementIndex plus full array.

```
indexes = []
[]
> [1, 2, 3, 4].forEach(( v, i ) => {
    if (v%2 === 0) indexes.push (i);
   })
undefined
> indexes
[ 1, 3 ]
>
```

Other Higher-Order Array Functions

Includes `every()`, `find()`, `findIndex()`, `filter()`, `reduceRight()`, `some()`.

```
> [1, 2, 3, 4].find(x => x%2 === 0)
2
> [1, 2, 3, 4].findIndex(x => x%2 === 0)
1
> [1, 2, 3, 4].every(x => x%2 === 0)
false
> [1, 2, 3, 4].some(x => x%2 === 0)
true
> [1, 2, 3, 4].reduce((acc, v) => acc - v)
-8                  //(((1-2)-3)-4
> [1, 2, 3, 4].reduceRight((acc, v) => acc - v)
-2                  //1-(2-(3-4))
>
```

Functions

- 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.
- During execution of a function, there is always an implicit object, referred to using `this`.

Traditional Function Definition

```
> function max1(a, b) { return a > b ? a : b }
undefined
> max1(4, 3)
4
> x = max1 //storing function in variable
[Function: max1]
> x.name
'max1'
> x.length
2
```

Defining Function Using a Function Expression

```
> x = max2 = function(a, b) { return a > b ? a : b }
[Function: max2]
> max2(4, 3)
4
> x(4, 3)
4
> x.name
'max2'
> x.length
2
>
```

Arrow Functions

```
> x = max4 = (a, b) => a > b ? a : b
[Function: max4]
> x(4, 3)
4
> x.name
'max4'
> x.length
2
>
```

Newer feature, does not bind `this` or other function "variables".
Cannot be used for constructors; best for non-method functions.

Defining Function Using 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
>
```

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.

Closure Example: Quadratic Equations

```
function quadEqn(a, b, c) {
  const discr = b*b - 4*a*c;
  return {
    a: () => a,
    b: () => b,
    c: () => c,
    root1: function() {
      return (-b + Math.sqrt(discr))/(2*a);
    },
    root2: function() {
      return (-b - Math.sqrt(discr))/(2*a);
    }
  };
}
```

Closure Example: Quadratic Equations Continued

```
const eqn1 = quadEqn(1, 4, -21);
const eqn2 = quadEqn(4, 4, -8);
const eqn3 = quadEqn(1, 2, -3);
[eqn1, eqn2, eqn3].forEach(function(e) {
  const s = `${e.a()}*x**2 + ${e.b()}*x + ${e.c()}`;
  console.log(`${s}: (${e.root1()}, ${e.root2()})`);
});
```

Closure Example: Quadratic Equations Log

```
$ ./quadEqn.js
1*x**2 + 4*x + -21: (3, -7)
4*x**2 + 4*x + -8: (1, -2)
1*x**2 + 2*x + -3: (1, -3)
$
```

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
```

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'
>
```

Function Properties for Memoization: Fibonacci

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
$
```

Object Context

Every function has access to **this** which is a reference to the current object. The value of `this` depends on how the function was called:

- 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`.
- When a function `f()` is called without a receiver, the use of `this` within the receiver refers to the global object; `window` for browsers, the current module for nodejs.
- 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)
```

- 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; }
... }
{ x: 22, f: [Function: f] }
> 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)
```

- 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)`.

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 }
{ x: 42 }
> f = a.getX.bind(b)
[Function: bound getX]
> 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; }
undefined
> Obj.prototype.f = function() {
... return function() { return this.x; }
... }
> Obj.prototype.g = function() {
... return () => this.x;
... }
> obj = new Obj()
Obj { x: 42 }
> 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
>
```