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
> delete x.a
true
> x
{}
> delete x.a    //false returned only for non-config
true            //property in non-strict mode
>
```

Object Basics Continued

```
> 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 }
> { x }
{ x: 'a' }
>
```

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, but is *no longer recommended*).

Object Methods

The `Object` class has many useful methods. Some particularly useful ones:

Object.create(proto) Returns new object with prototype `proto`.

Object.assign(target,...sources) Assign `source` properties to `target`, with later source properties overwriting earlier ones. Returns `target`.

Object.getOwnPropertyNames(obj) All non-inherited property names.

Object.keys(), Object.values(), Object.entries() Enumerable keys, values and key-value pairs.

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 = Object.create(Parent.prototype)`. Hence `Child` will inherit properties from `Parent`.
- Older code may use a pattern like `Child.prototype = new Parent()`. Problems include the fact that the `Parent` constructor is run (which may have undesirable side-effects) and how do we handle arguments to the `Parent` constructor if it expects arguments.
- Note that we create a new object for Child's prototype rather than simply `Parent` as we do not want assignments to `Child.prototype` to affect `Parent`.
- Since the constructor is stored in an object's prototype, in both cases we need to fix up the constructor: `Child.prototype.constructor = Child`.
- 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;
  }

  //possibly poor design
  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
>
```

Review 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).

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 } }
```

Property Attributes Continued

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

Property Getter

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

Property Setter

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

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

Property Setter Continued

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

Enumerating Object Properties using for-in

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

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

Enumerating Example

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

Enumerating Example Continued

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

Another Enumerating Example

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

Iterating using for-of

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

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

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

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

Building Iterables

Can build iterables by implementing the *iterable protocol*.

- Implementing a zero argument method with name given by `Symbol.iterator`.
- When this function is invoked, it must return an object implementing the **iterator protocol**.
- So two protocols involved: **iterable** and **iterator**. Generators will simplify.

Building Iterators

An object implementing the **iterator protocol** must have a `next()` method which returns an object having at least the following two properties:

`done` A boolean which is set to true iff the iterator is done. If true, then `value` optionally gives the return value of the iterator.

`value` Any JavaScript object giving the current value returned by the iterator. Need not be present when `done` is true.

Sequence Iterable

Build a sequence iterable to allow iterating through a sequence of integers. Example edited log:

```
> for (const v of makeSeq(3, 5)) { console.log(v); }
3
4
5

> for (const v of makeSeq(3, 10, 2)) { console.log(v); }
3
5
7
9
>
```

Sequence Iterable Log Continued

```
> for (const i of makeSeq(1, 2)) { //nested seq obj lifetimes
    for (const j of makeSeq(3, 4)) {
      console.log(i, j);
    }
  }
1 3
1 4
2 3
2 4
>
```

Sequence Iterable Code

In seq.js:

```
function makeSeq(lo, hi, inc=1) {
  return {
    [Symbol.iterator]() { //fn property syntax
      let value = lo;
      return {
        next() {
          const obj = { done: value > hi, value };
          value += inc;
          return obj;
        },
      };
    },
  };
}
```

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'
>
```