Objects

• Source syntax

\{name_1: \text{expr}_1, ..., name_k: \text{expr}_k\}

• Names can be identifiers or strings (or numbers, but 0 is taken as “0”)

• ECMA 6 allows \text{[string-valued-expression]}
Object semantics

- An object is a mutable dictionary mapping strings (only!) to JavaScript values.

- The order of the maplets may be preserved or arbitrarily scrambled; inside it might be a hash table, binary tree, or simple sequence.

- Whatever, an object is a data structure that you could implement yourself.
Object iteration

• for (key in object) ...
  iterates over the keys (property names) of an object.

• for (key in array) ...
  iterates over the keys (indices with defined values) of an array in order, as **strings**

• because arrays are objects
What, no arrays?

• No. An array is just an object with keys that look like non-negative integers and a .length property.

• An idea we’ve seen before: the implementation may represent objects more than one way, there may be an array-like representation.
Fields and subscripts

• Fetch: object.name, array[index]

• Store: object.name = x, array[index] = x

• Same thing: object.name = object[“name”]

• var a = [2]; a[“length”] ⇒ 1; a[”0”]⇒ 2
Functions

- function name(args) { ... return result; }

- Except for ‘function’ keyword and lack of types, looks much like C or Java.

- Functions are objects with some special fields. They are values. You can pass them as parameters and store them in variables.
Passing functions

• function add1(x) { return x+1; }

• [3,1,4].map(add1) ⇒ [4,2,5]

• [4,2,5].map(function (x) { return x*3 })

• ⇒ [12,6,15]
What’s map?

- Array.prototype.map = function (fn) {
  var k, a = []
  for (k in this) a[k] = fn(this[k])
  return a
}
Functions nest

- You can declare functions inside other functions.

- C was *exceptional* in forbidding this.

- Algol 60 made nested functions mainstream. Until C++, C was the only well known language that did not support them. Even Fortran supports this (a bit).
Scope

• A function can see its own variables and parameters, plus anything the next function out can see.

• A stack frame contains variables and parameters, a return address, a dynamic link to the caller’s frame, and a static link to the parent’s frame. JavaScript adds an object link that “this” refers to.
Whence the static link?

• A function value contains (a reference to) the instructions to be executed and

• an environment — (a reference to) the stack frame of the function containing it. When called, this becomes the static link of the new frame.

• Such a pair is called a closure because the function is “closed over its environment”.
Stack frame? Object!

- function times(x) {
  return function (y) { return x * y }
}

- var tripler = times(3); tripler(4) \Rightarrow 12

- times(3)(4) is OK too.
How does that work?

- call times(3) makes a new frame F binding x to 3.

- the body creates a new closure C whose environment is F.

- return transfers control back, with C as value. But C points to F, F is still in use, so F does not go away. It’s just an object.
Another example

- Array.prototype.filter = function (fn) {
  var k, a = []
  for (k in this)
    if (fn(this[k])) a.push(this[k])
  return a
}
The scope botch & why

• var and function declarations can be anywhere in a file or in a function.

• Wherever they are, they are “hoisted” to the top of that file or function.

• Without regard for any curly braces you might have expected to get in the way.
Scope botch continued

• `var y = 1`
  
  ```javascript
  function f(x) {
    { var y = 2 }
  } print(y)
  ```

• C, Java, Algol, Pascal: output would be 1

• JavaScript, it's 2.

• Tries to make order of declarations unimportant, call before definition.
Making objects

• You can create an object using an object literal. No classes necessary!

• A method is just a function closure that is the value of some property in an object.

• `var x = {a: 1, b: function () {return this.a},
            c: function (y) {this.a = y*2} }`
Method calls set “this”

• `object.name(arguments)`
  evaluates object and arguments,
  looks up name, sets `this` to `object`,

• then calls the function.

• Be careful!
Syntactic subtlety

• \( f = \text{object.name; } f(\text{arguments}) \)

• does everything that a method call does,

• except that it \textbf{doesn’t set this}.  

• When \textbf{this} is not set, it refers to the “global” object. In a browser, it’s the current window. In js/node it’s all the top level declarations.
Without classes, what?

• To create many similar objects, write a function.

• function Thingy(constr_args) { return {
    field: initial_value, ...
    method: function (...) { ... }, ...
} }

Another way

• function Thingy(constr_args) {
  this.field = initial_value; ...
  this.method = function (...) { ... }; ...
}

• Does not return the new object. And where is “this” set?
Constructor calls

• **new** constructor(arguments)

• evaluates the arguments, creates a new object, sets `this` to *that new object*, calls constructor, discards its result, and returns the new object.

• Convention: use a capital letter for a constructor function like this.
Beware!

- A constructor is just another function. If you call it without `new`, `this` won’t be set, and will default to the global object.

- `constructor(...)` will smash variables in your global object. You don’t want this!

- Start normal functions with lower case, constructor functions with upper case.
Factoring

• Classes let a whole lot of objects share one set of methods. What we’ve seen so far gives each object its own independent set.

• Objects in JavaScript inherit from other objects, not classes!

• When you use `new`, the new object will inherit from the constructor’s prototype
Prototypes

• function Point(x, y) { this.x = x; this.y = y }

• Point.prototype.toString = function () { return "(" + this.x + ", " + this.y + ")" }

• new Point(3, 5) \Rightarrow (3, 5)

• Point.prototype.move = function (x,y) {
  this.x += x; this.y += y; return this
}