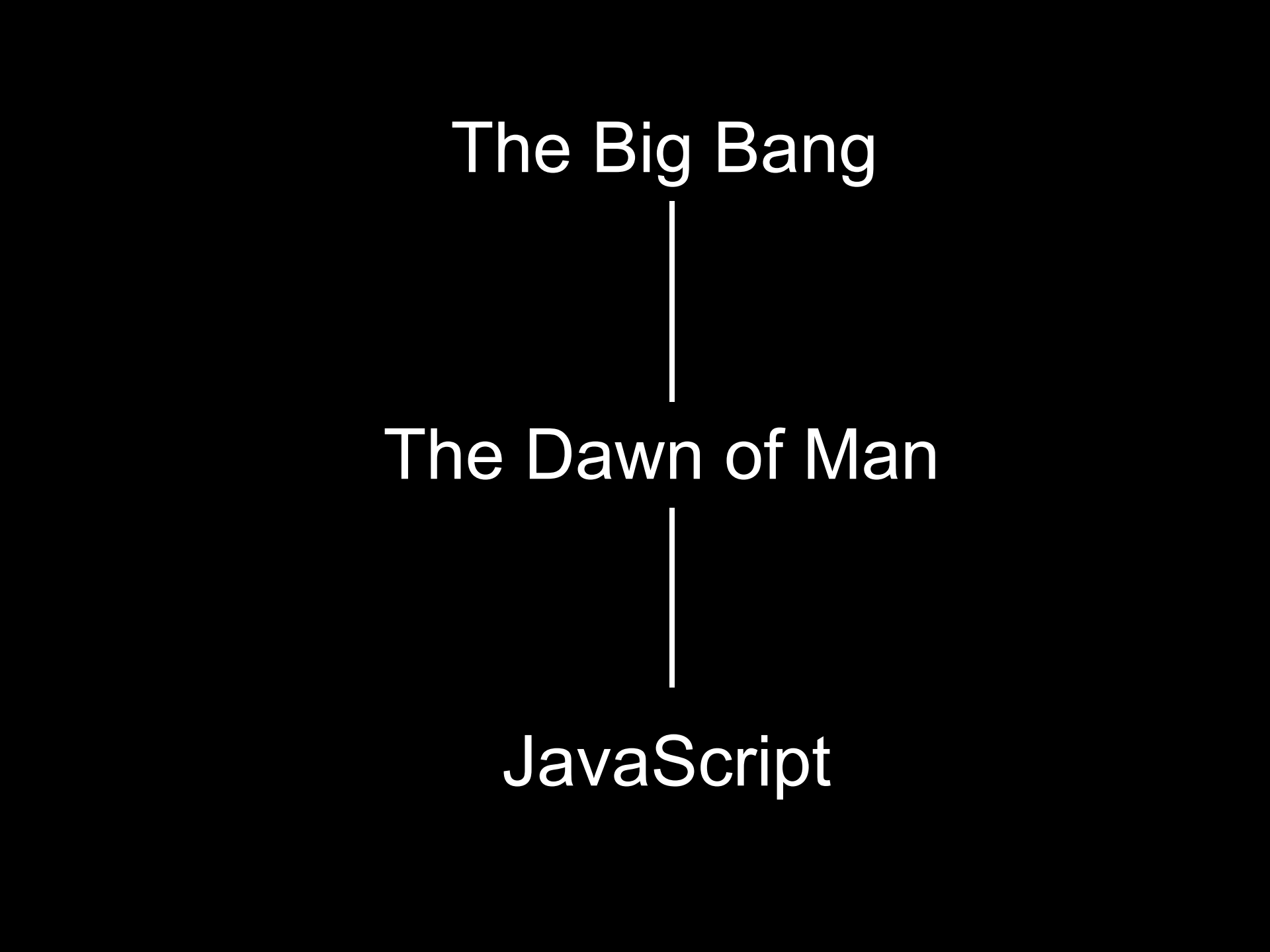
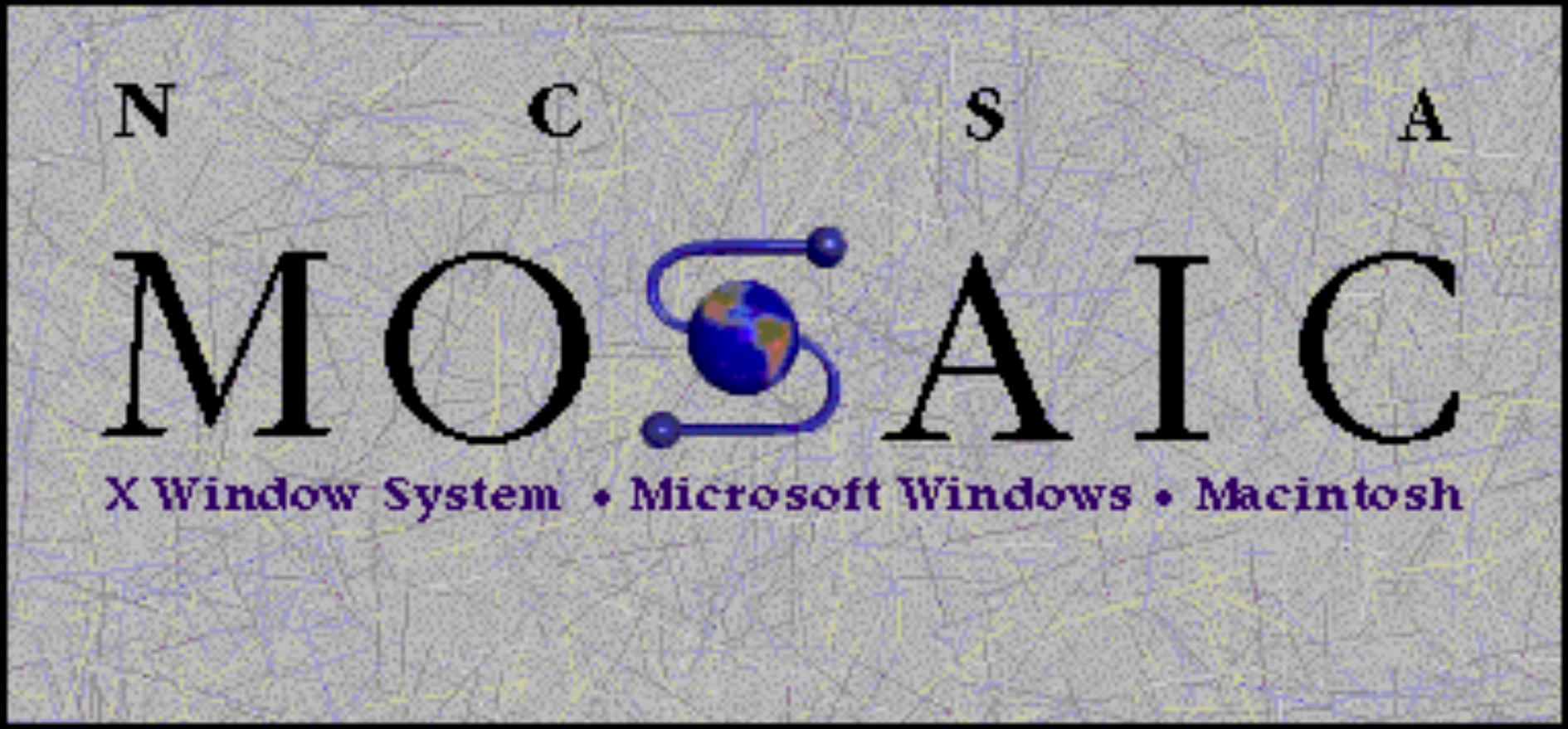
Chapter 2

And Then There Was JavaScript


The Big Bang
The Dawn of Man
JavaScript

The First Important Discovery of the 21st Century

JavaScript has good parts.


N C S A
MOSAIC
X Window System • Microsoft Windows • Macintosh

Welcome to HyperCard

Home


HyperCard Tour


HyperCard Help


Practice


New Features


Art Bits


Addresses


Phone Dialer

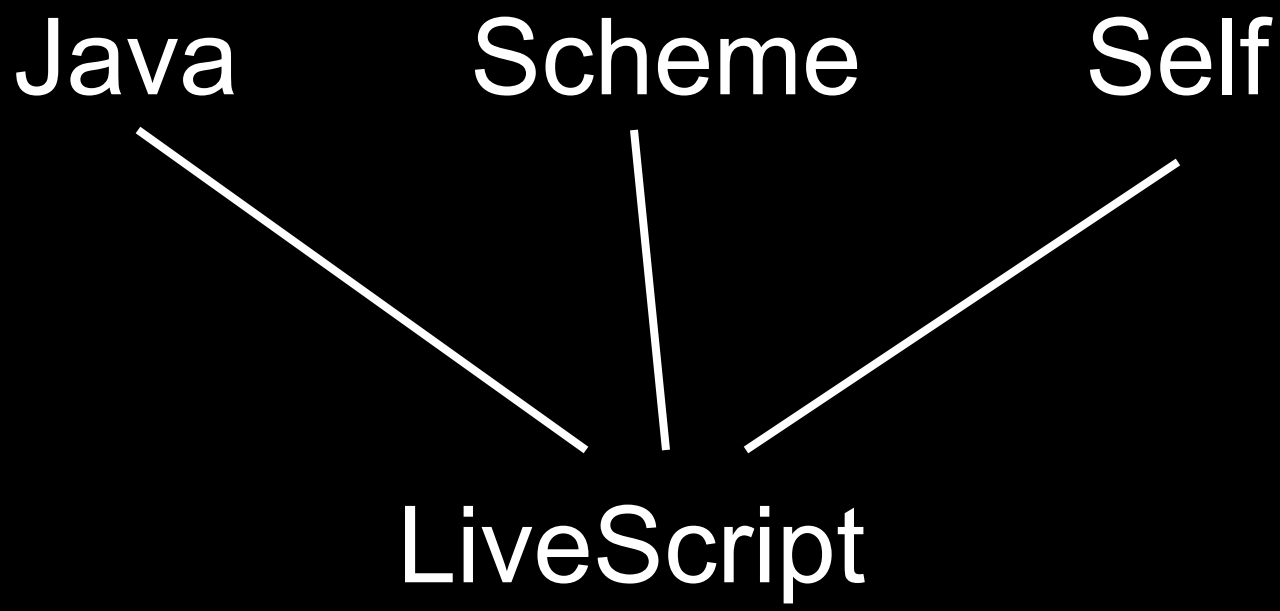

Graph Maker

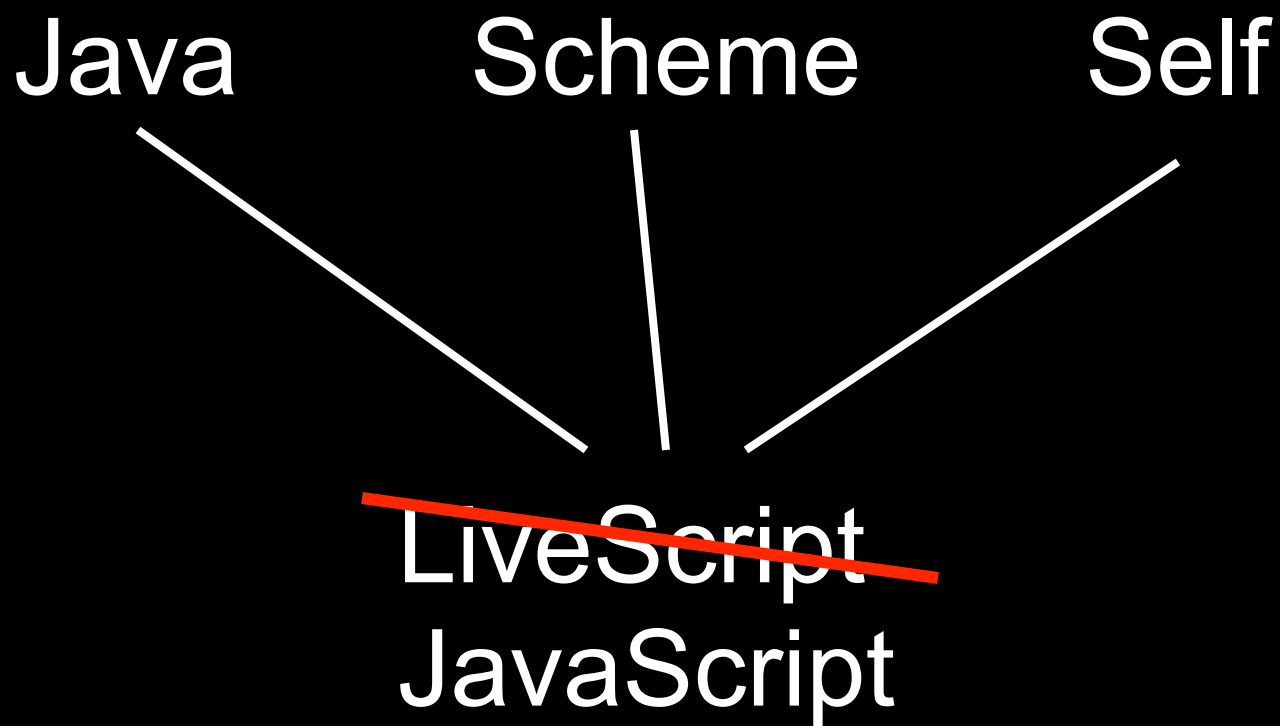

QuickTime Tools

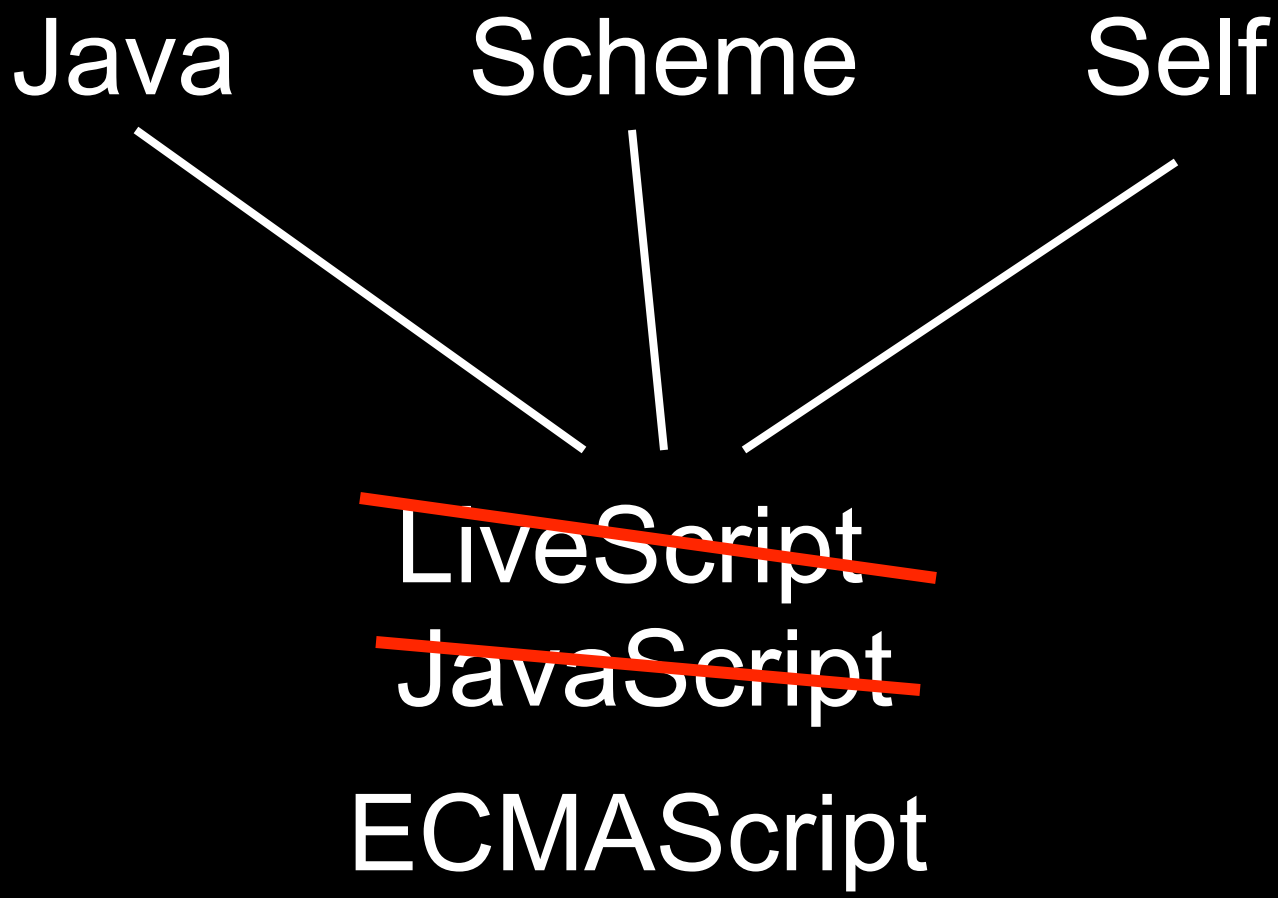

AppleScript Mail Merge


AppleScript Text Controls


YUI
YUICONF 2009
OCTOBER 28-29


Java
Scheme
Self
LiveScript


Java
Scheme
Self
LiveScript
JavaScript


Java
Scheme
Self
LiveScript
JavaScript
ECMAScript

ECMAScript

- 1999 Third Edition ES3
- 2009 Fifth Edition ES5
  - Default
  - Strict

- For the short term, work in the intersection of ES3 and ES5/Strict
- For the long term, work with ES5/Strict.
- Avoid ES5/Default.

Where do Bad Parts come from?

- Legacy
- Good Intentions
- Haste

- For the most part, the bad parts can be avoided.
- The problem with the bad parts isn't that they are useless.

Objects

An object is a dynamic collection of properties.

Every property has a key string that is unique within that object.

Get, set, and delete

- get
  object.name
  object[expression]

- set
  object.name = value;
  object[expression] = value;

- delete
  delete object.name
  delete object[expression]

Object literals

- An expressive notation for creating objects.

```
var my_object = {foo: bar};

var my_object = Object.defineProperties(
        Object.create(Object.prototype), {
    foo: {
        value: bar,
        writeable: true,
        enumerable: true,
        configurable: true
    }
});
```

Classes v Prototypes

Working with prototypes

- Make an object that you like.
- Create new instances that inherit from that object.
- Customize the new objects.

- Taxonomy and classification are not necessary.

Delegation

Differential Inheritance

new prefix operator

```
function new(func, args) {
    var that = Object.create(func.prototype),
        result = func.apply(that, args);
    return (typeof result === 'object' &&
        result) || that;
}
```

JavaScript didn’t get it quite right

```
function bump_count(word) {
    if (word_count[word]) {
       word_count[word] += 1;
    } else {
       word_count[word] = 1;
    }
}
```

- Accidental collisions:
  Fails when `word === 'constructor'`

JavaScript didn’t get it quite right

```
function bump_count(word) {
    if (typeof word_count[word] ===
            'number') {
        word_count[word] += 1;
    } else {
        word_count[word] = 1;
    }
}
```

The `for in` problem

- Functions inherited from a prototype are included in the `for in` enumeration.

```
for (name in object) {
    if (object.hasOwnProperty(name) {
        …
    }
}
```

- This can fail if an object has a `hasOwnProperty` property.

The `for in` problem

- Functions inherited from a prototype are included in the `for in` enumeration.

```
for (name in object) {
    if (Object.prototype
        .hasOwnProperty
        .call(object, name)) {
            ...
    }
}
```

Keys must be strings

Automatic type coercion.

Object and …

- Number
- Boolean
- String
- Array
- Date
- RegExp
- Function

Number

Numbers

- Only one number type

  No integer types

- 64-bit floating point
- IEEE-754 (aka “`Double`”)

Good that we don't have `int`

```
a > 0
b > 0

c = a + b;
```

- **Possible results:**

```
c < a
c < b
```

Number literals

```
.01024e4
1.024e+3
  10.24E2
102.4E+1
1024.e0
1024.00
  1024
10240e-1
```

Associative Law does not hold

```
(a + b) + c === a + (b + c)
```

- Produces `false` for some values of `a`, `b`, `c`.

- Integers under 9007199254740992 (9 quadrillion) are ok.

9007199254740992 ===

```
(a + 1) - 1 === a
```

Can be false.

Decimal fractions are approximate

```
a = 0.1;
b = 0.2;
c = 0.3;

(a + b) + c === a + (b + c)
false
```

Number methods

- toExponential
- toFixed
- toLocaleString
- toPrecision
- toString
- valueOf

Number methods

```
if (!Number.prototype.trunc) {
    Number.prototype.trunc =
      function trunc(number) {
        return number >= 0
            ? Math.floor(number)
            : Math.ceil(number);
    };
}
```

Numbers are first class objects

- A number can be stored in a variable.
- A number can be passed as a parameter.
- A number can be returned from a function.
- A number can be stored in an object.

- A number can have methods.

Math object

- abs
- acos
- asin
- atan
- atan2
- ceil
- cos
- exp
- floor
- log
- max
- min
- pow
- random
- round
- sin
- sqrt
- tan

Math object

- E
- LN10
- LN2
- LOG10E
- LOG2E
- PI
- SQRT1_2
- SQRT2

```
function log2(x) {
    return Math.LOG2E *
        Math.log(x);
}
```

`NaN`

- Special number: Not a Number
- Result of undefined or erroneous operations
- Toxic: any arithmetic operation with **`NaN`** as an input will have **`NaN`** as a result
- **`NaN`** is not equal to anything, including **`NaN`**
- **`NaN === NaN`** is **`false`**
- **`NaN !== NaN`** is **`true`**

```
x = x + 1
```

Mathematic nonsense.

$$x = x + 1$$

Infinity

```
x = x + 1
```

Infinity

Number.MAX_VALUE

```
x = x + 1
```

Infinity

Number.MAX_VALUE

9007199254740992

```
x = x + 1
```

But not `NaN`

because `NaN === NaN` is `false`

```
x = x + 1
```

But not `NaN`

because `NaN === NaN` is `false`

`NaN !== NaN` is `true`!

Boolean

`true`

`false`

String

Strings

- A sequence of 0 or more 16-bit Unicode characters
  - UCS-2, not quite UTF-16
  - No awareness of surrogate pairs
- No separate character type
  - Characters are represented as strings with length of 1
- Strings are immutable
- Similar strings are equal (`===`)
- String literals can use single or double quotes with \ escapement.
- Use `"` for external strings.
- Use `'` for internal strings and characters.

+

+ can concatenate or add.

```
'$' + '1' + '2' === '$12'
```

```
'$'.concat('1').concat('2')
```

Convert a number to a string

- Use number's method (`toString`)
- Use `String` function

```
str = num.toString();
str = String(num);
```

Convert a string to a number

- Use the Number function.
- Use the + prefix operator.
- Use the `parseInt` function.

```
num = Number(str);
num = +str;
```

parseInt function

```
parseInt(str, 10)
```

- Converts the value into a number.
- It stops at the first non-digit character.

```
parseInt("12em") === 12
```

- The radix (10) should always be used.

```
parseInt("08") === 0
parseInt("08", 10) === 8
```

String `length`

- string.`length`

- The `length` property determines the number of 16-bit characters in a string.
- Extended characters are counted as 2.

String methods

- charAt
- charCodeAt
- compareLocale
- concat
- indexOf
- lastIndexOf
- localeCompare
- match
- replace
- search
- slice
- split
- substring
- toLocaleLowerCase
- toLocaleUpperCase
- toLowerCase
- toString
- toUpperCase
- trim
- valueOf

trim

```
if (typeof String.prototype.trim !==
        'function') {
    String.prototype.trim = function () {
        return this.replace(
            /^\s*(\S*(\s+\S+)*)\s*$/,
            "$1");
    };
}
```

supplant

```
var template = '<table border="{border}">' +
    '<tr><th>Last</th><td>{last}</td></tr>' +
    '<tr><th>First</th><td>{first}</td></tr>' +
    '</table>';

var data = {
    first: "Carl",
    last: "Hollywood",
    border: 2
};

mydiv.innerHTML = template.supplant(data);
```

supplant

```
if (typeof String.prototype.supplant !==
        'function') {
    String.prototype.supplant = function (o) {
        return this.replace(/{([^{}]*)}/g,
            function (a, b) {
                var r = o[b];
                return typeof r === 'string' ?
                    r : a;
            });
    };
}
```

Array

Arrays

- **`Array`** inherits from **`Object`**.
- Indexes are converted to strings and used as names for retrieving values.
- Very efficient for sparse arrays.
- Not very efficient in most other cases.
- One advantage: No need to provide a length or type when creating an array.

length

- Arrays, unlike objects, have a special **length** property.
- It is always 1 larger than the highest integer subscript.
- It allows use of the traditional **for** statement.

```
for (i = 0; i < a.length; i += 1) {
    ...
}
```

- Do not use **for in** with arrays

Array Literals

- An array literal uses `[]`
- It can contain any number of expressions, separated by commas

```
myList = ['oats', 'peas', 'beans'];
```

- New items can be appended

```
myList[myList.length] = 'barley';
```

- The dot notation should not be used with arrays.

Array methods

- concat
- every
- filter
- forEach
- indexOf
- join
- lastIndexOf
- map
- pop
- push
- reduce
- reduceRight
- reverse
- shift
- slice
- some
- splice
- toLocaleString
- toString
- unshift

sort

```
var n = [4, 8, 15, 16, 23, 42];
n.sort();
// n is [15, 16, 23, 4, 42, 8]
```

Deleting Elements

```
delete array[number]
```

- Removes the element, but leaves a hole in the numbering.

```
array.splice(number, 1)
```

- Removes the element and renumbers all the following elements.

Deleting Elements

```
myArray = ['a', 'b', 'c', 'd'];

delete myArray[1];

// ['a', undefined, 'c', 'd']

myArray.splice(1, 1);

// ['a', 'c', 'd']
```

Arrays v Objects

- Use objects when the names are arbitrary strings.
- Use arrays when the names are sequential integers.
- Don't get confused by the term Associative Array.

Date

The `Date` function is based on Java's Date class.

It was not Y2K ready.

RegExp

```
/\/(\\[^\x00-\x1f]|\[(\\[^\x00-\x1f]|[^\x00-\x1f\\\/])*\]|[^\x00-\x1f\\\/\[])+\/[gim]*/
```

Function

All values are objects

Except `null` and `undefined`.

`null`

A value that isn't anything

`undefined`

- A value that isn't even that.
- The default value for variables and parameters.
- The value of missing members in objects.

typeof

- The `typeof` prefix operator returns a string identifying the type of a value.

type	`typeof`
object	`'object'`
function	`'function'`
array	`'object'`
number	`'number'`
string	`'string'`
boolean	`'boolean'`
`null`	`'object'`
`undefined`	`'undefined'`

Array.isArray

```
alert(Array.isArray([]));   // true


if (typeof Array.isArray !== 'function') {
    Array.isArray = function (value) {
        return Object.prototype
            .toString.apply(value) ===
                '[object Array]';
    };
}
```

Falsy values

- `false`
- `null`
- `undefined`
- `""` (empty string)
- `0`
- `NaN`

- **All other values (including all objects) are truthy.**

`"0"` `"false"`

Loosely Typed

- Any of these types can be stored in an variable, or passed as a parameter to any function.
- The language is not untyped.

Reference

- Objects can be passed as arguments to functions, and can be returned by functions
  - Objects are passed by reference.
  - Objects are not passed by value.
- The `===` operator compares object references, not values
  - `true` only if both operands are the same object

C

- JavaScript is syntactically a C family language
- It differs from C mainly in its type system, which allows functions to be values

Identifiers

- Starts with a letter or _ or **$**
- Followed by zero or more letters, digits, _ or **$**
- By convention, all variables, parameters, members, and function names start with lower case
- Except for constructor functions which start with upper case
- Initial _ should be reserved for implementations
- **$** should be reserved for machines.

Comments

```
// slashslash line comment

/*
    slashstar
    block
    comment
*/
```

Operators

- Arithmetic
  `+  -  *  /  %`
- Comparison
  `==  !=  <  >  <=  >=`
- Logical
  `&&  ||  !`
- Bitwise
  `&  |  ^  >>  >>>  <<`

  Ternary
  `?:`

+

- Addition and concatenation
- If both operands are numbers,

  then

    add them

  else

    convert them both to strings
    concatenate them

```
'$' + 3 + 4 = '$34'
```

+

Unary operator can convert strings to numbers

```
+"42" = 42
```

Also

```
Number("42") = 42
```

Also

```
parseInt("42", 10) = 42
```

```
+"3" + (+"4") = 7
```

/

- Division of two integers can produce a non-integer result
  - `10 / 3 = 3.3333333333333335`

%

- The remainder operator, not the modulo operator.

```
-1 % 8    // -1, not 7
```

== !=

- Equal and not equal
- These operators can do type coercion
- It is always better to use === and !==, which do not do type coercion.

Evils of type coercion

- `'' == '0'` // false
- `0 == ''` // true
- `0 == '0'` // true

- `false == 'false'` // false
- `false == '0'` // true

- `false == undefined` // false
- `false == null` // false
- `null == undefined` // true

- `' \t\r\n ' == 0` // true

&&

- The guard operator, aka *logical and*
- If first operand is truthy
  then result is second operand
  else result is first operand
- It can be used to avoid null references

```
if (a) {
   return a.member;
} else {
   return a;
}
```

- can be written as

```
return a && a.member;
```

||

- The default operator, aka *logical or*
- If first operand is truthy
  then result is first operand
  else result is second operand
- It can be used to fill in default values.

```
var last = input || nr_items;
```

- (If `input` is truthy, then `last` is input, otherwise set `last` to `nr_items`.)
- May not work as expected if the first operand is a number, because 0 is falsy.

!

- Prefix *logical not* operator.
- If the operand is truthy, the result is **`false`**. Otherwise, the result is **`true`**.
- `!!` produces booleans.

Bitwise

`&` `|` `^` `>>` `>>>` `<<`

- The bitwise operators convert the operand to a 32-bit signed integer, and turn the result back into 64-bit floating point.

Statements

- *expression*
- `if`
- `switch`
- `while`
- `do`
- `for`
- `break`
- `continue`
- `return`
- `try/throw`

Break statement

- Statements can have labels.
- Break statements can refer to those labels.

```
loop: for (;;) {
    …
        if (…) {
            break loop;
        }
    …
}
```

For statement

- Iterate through all of the elements of an array:

```
for (i = 0; i < array.length; i += 1) {

        // within the loop,
        // i is the index of the current member
        // array[i] is the current element

}
```

For in statement

- Iterate through all of the members of an object:

```
for (name in object) {
    if (object.hasOwnProperty(name)) {

        // within the loop,
        // name is the key of current member
        // object[name] is the current value

    }
}
```

Switch statement

- Multiway branch.
- The switch value does not need to a number. It can be a string.
- The case values can be expressions.
- Danger: Cases fall through to the next case unless a disruptive statement like **`break`** ends the case.

Switch statement

```
switch (expression) {
case ';':
case ',':
case '.':
    punctuation();
    break;
default:
    noneOfTheAbove();
}
```

Throw statement

```
throw new Error(reason);

throw {
    name: exceptionName,
    message: reason
};
```

Try statement

```
try {
    ...
} catch (e) {
    switch (e.name) {
    case 'Error':
        ...
        break;
    default:
        throw e;
    }
}
```

Try Statement

- The JavaScript implementation can produce these exception names:

```
'Error'
'EvalError'
'RangeError'
'SyntaxError'
'TypeError'
'URIError'
```

Next time:

Act III

Function the Ultimate

With statement

- Intended as a convenient short-hand
- Ambiguous
- Error-prone
- Don't use it

```
with (o) {
    foo = koda;
}
```

- `o.foo = koda;`
- `o.foo = o.koda;`
- `foo = koda;`
- `foo = o.koda;`

With statement

```
if ('foo' in o) {
  o.foo = 'koda' in
    foo ? o.koda :
    koda;
} else {
  foo = 'koda' in
    foo ? o.koda :
    koda;
}
```

```
with (o) {
    foo = koda;
}
```

- `o.foo = koda;`
- `o.foo = o.koda;`
- `foo = koda;`
- `foo = o.koda;`