JavaScript (or ECMAScript)

Sivan Toledo
Tel-Aviv University

Origins and Evolution

Developed by Netscape as a scripting language to be used within web pages (html files)

Evovled over the years:

Javascript 1.0, 1.1, ...	(Netscape/Mozilla)
JScript 1.0, 2.0, ...	(Microsoft)
ECMAScript v1, v2, ...	(International Standard)

This presentation documents JS 1.5/ECMA v3

We ignore backward compatibility techniques

JavaScript in Other Contexts

Supported by all widely-used browsers

Scripting within applications (Adobe Photoshop, Adobe Illustrator, Adobe PDF files, Flash files, etc)

Operating-system-level scripting language (WSH)

Desktop-applet systems (Konfabulator/ Yahoo Widgets, Apple's Dashboard)

Desktop-application development (Mozilla XUL)

Let's Get Started: A Minimal Script

```
<html>
  <script>

  var x = 3;
  var y = 47;

  alert( " 3 x 47 = " + (x*y) );

  </script>
</html>
```

Numbers

All numbers are 64-bit floating-point (IEEE-754)

```
12
3.14
3.14e-5
0xff      // hexa, decimal value 255
0377      // don’t use: decimal 377,
          // but old implementation may
          // take as octal (255 decimal)
Infinity
NaN
Number.MAX_VALUE
Number.MIN_VALUE
```

Strings

```
"Hello World"
'Hello World' // exactly the same
'the string "Hello World"'
"It doesn't matter"
"The letter Alef: \u05D0" // Unicode

"\n"  // newline (also \r, \t)
"\\"  // backslash
"\""  // double quotes (also \')

"Hello " + "World" // concatenation
s.length           // length of string
```

Booleans

```
true
false

n == 12 // an expression that returns
        // a boolean value
```

Objects (Really Just Dictionaries)

```
var bond = { first: "James",
             last:  "Bond",
             id:    7
           }; // object literal

bond.last // returns the string "Bond"

bond[ "id" ] // same as bond.id

var q = new Object();
q.first = "Mr. Q";
```

Arrays

```
var l = [ 1, 2, , , , 3 ]; // literal
var a = Array();
a[0] = "James";
a[1] = "Bond";
a[2] = 7;
a[4] = { first: "Mr. Q", id = 1 };

var b = new Array("James", "Bond", 7);

var c = new Array(7);
 // length-7 array, undefined elements
 // NOT a length-1 array with c[0]==7
```

null and undefined

`null`

a special value that represents no value

`undefined`

a special value that indicates that the variable (or object property or array element has not yet been assigned a value)

```
bond.nationality   // undefined
undefined == null  // true
undefined === null // false
```

Regular Expressions

Library objects, but with literals

```
/^EVENT/
/test/i
/[a-z]*/
```

Variables

Variables should be declared before they are used

Variables do not have a type; a variable can store a number, later a string, later an array

```
var e, pi = 3.14;
pi = "Apple";

alert(pi);    // valid, shows 3.14
alert(e);     // valid but undefined
alert(zeta);  // an error
```

Omitted Declarations

```
var i;      // declared, value is
            // defined

z = 3;      // implicit declaration
            // correct but should
            // be avoided

alert(y); // error
            // reading a variable does
            // not implicitly declare it
```

Functions

```
function show_message(s) {
  alert( "The message is: " + s );
}
show_message("Click OK to continue");
```

Local Scope

```
var pi = 3.14;
var e  = 2.72;

function foo() {
  alert( pi );                 // global
  var e = "Entertainment"; // local
  alert( e );
}

function bar() {
  i = 3; // implicit declaration of a
}        // global variable
```

Nested Scope

```
function outer() {
  var o1, o2;
  function inner() {
    var i;
    var o1;  // hides outer o1
  }
}
```

No Block Scope

```
function foo() {
  if (1 == 0) {
    var pi = 3.14;
  }

  alert( pi ); // no declaration
}
```

Operators: Numerical & Bitwise

```
*, /
%    reminder
-, + also string concatenation
++, --
!    boolean not
~    bitwise not (on integers)
|    bitwise or  (on integers)
&    bitwise and (on integers)
^    bitwise xor (on integers)
>>   arithmetic shift right
>>>  shift with zero padding
<<   shift left
```

Operators: Booleans & Comparisons

```
|| logical or (short evaluation)
&& logical and
!  logical not

<, <=
>, >=
```

String comparisons are alphabetic using Unicode values; usually not what you want: `Z < a`

Numerical comparisons with NaN return `false`

Operators: Assignment (& Argument Passing)

```
= assignment (also += etc)
```

```
x = y = z = 0;
```

Numbers and Booleans are copied on assignment and passed by value to functions

Strings are immutable and compared by contents, so they also have a "by value" behavior

Everything else passed by reference

Operators: Equality

`==` `equality`

By value for numbers, strings, booleans

Strings can be converted to numbers

Booleans can be converted to 0/1

Objects converted to string/number (more later)

By reference for objects, arrays, functions

`null` and `undefined` are equal

Operators: Identity

`=== identity`

Different types (e.g., number/string) never identical
Same values, object, array, function are identical
But no number identical to `NaN` (not even `NaN`)
(use `isNaN(x)` instead)
Strings with exactly the same contents (characters)
& length are identical
`null` values are all identical
`undefined` values are identical
`(null  === undefined) == false`

Operators: Conditional

```
alert(x + " is " +
      (x % 2) == 0 ? "even" : "odd"
     );
```

Not too useful

Operators: typeof

```
typeof( x )
```

Return values

```
  "number"
  "string"
  "boolean"
  "object" (also for arrays)
  "function"
  "undefined"
```

Operators: new, delete, void

`new (later; creates a new object/array)`

`delete bond.id;`
deletes a property of an object or an element of an array; the opposite of declaring a variable/property etc.

`var x = void f(); // returns undefined`
useful (rarely) for throwing away a return value

One Bit of Syntax

Statements end in ;

JavaScript automatically adds ; at the end of lines if a valid statement ends at the end of the line

```
i = 3
j = 4;
return
        true
```

First 3 lines are valid statement, but the third will return from a function without returning a value

Comments

```
i = 3; // comment to end of line

j = 4; /* comment */

alert( 55 /* error code */ );

/*
   a
   long
   multiline
   comment
*/
```

Statements

```
i = 3;            // single expression
{ i=3; j=4; } // block

if ( x % 2 == 0 )
  alert("even");
else                // else is optional
  alert("odd");
```

Switch statement

```
switch (n) {
  case 1:
    // do something ...
    break;
  case i+j:
    // do something else ...
    break;
  default:
    // do a third thing ...
}
```

beware of fall through

Looping Statements

```
while ( ... ) {
  // do something
}

do {
  // something else
} while ( ... );
```

the test is at the end of the body

```
for (i=0; i<10; i++) alert(i);
```

More on For Loops

```
for ( initialization-expression;
      continue-condition;
      increment-after-body ) body
```

```
for (var j=100; j>=0; j--) ...
```

special syntax: var allowed inside init expression

```
for (property in bond)
  alert("bond." + property + " = "
        bond[ property ] );
```

another kind of special syntax for objects and arrays; not all object properties are enumerated

Break, Continue, and Labels

`break;`
exits from the innermost loop/switch entirely

`continue;`
jumps to the end of the body of the innermost loop

```
outer: for (i=0; i<10; i++) {
  inner: for (j=0; j<10; j++) {
    if (i+j == 20) break outer;
    if (i*j == 12) continue outer;
```

With Statement

An oddity; best to avoid

```
with (bond) {
  alert( "The name is "
         + last // bond.last
         + ", "
         + first + " " + last );
```

Defining Functions

```
function min(x,y) {
  return (x < y) ? x : y;
}

var max = function(x,y) {
    return (x > y) ? x : y;
  };

var r = function r(x) {
  if (x<1) return 1;
  else return r(x-1);
}
```

Functions are First-Class Objects

All the forms of defining a function in the previous slide do the same thing: they define a function and bind it to a variable

We can also define a function without binding it to a variable (when we pass it as an argument)

We can store a function in a variable, array element, or object property; we can assign functions to variables, and so on

First Class Example

```
function map_array(a, f) {
  var b = new Array( a.length );
  for (var i=0; i<a.length; i++)
    b[i] = f( a[i] );
  return b;
}

map( [1,2,3],
     function(x) {return x*x;}
   );
```

The Arguments Object

Functions can be invoked with any number of arguments

```
min(3,4);
min(5);
min(2,3,4,5);

function min(x,y) {
  var m = 0;
  for (var i=0; i<arguments.length; i++)
  return m; }
```

More about Arguments

`min.length`
The expected number of arguments, 2 in this case

`arguments.callee`
The function that the arguments were passed to; useful mostly for recursive invocation of anonymous functions

Function Properties

```
min.help = "Returns its smallest"
           + " argument";

function good_memory(x) {
  if (good_memory.old == undefined)
    alert("Nice to meet you");
  else
    alert("Last time you were "
      + good_memory.old);
  good_memory.old = x;
}
```

Returning from a Function

```
return;
```

Ends the execution of the function and returns the value `undefined`

```
return 3;
```

Returns with a value

Exception Handling

```
var f;
try {
  f = http_get("http://www...");
  ... // do something with f
} catch (e) {
  alert("HTTP GET failed: "+e.message);
} finally {
  ... // cleanup actions
}
```

The {...} are necessary

Details of Try-Catch-Finally

`catch` is optional, but rarely omitted

`finally` is optional, usually omitted

`finally` block is executed no matter how program left the `try` block: normally, an exception, `break`, `continue`, `return`

Throwing an Exception

```
throw -1;
```

Control passes to the closest exception-handling block on the call stack, perhaps (often) in another function, higher up in the call stack

The thrown object is bound to the argument of the `catch` phrase, if there is one

If there is no `catch` phrase, the `finally` block is executed but the exception is thrown again

Objects

Dictionaries of **properties**

A property has a name (e.g., `id`) and can be bound to a value (or to `undefined`)

Properties can be added by assigning to them and can be deleted (`delete bond.id`)

Object creation:

```
var o = new Object();
o.id = 9; // a new property
var me = { first: "Sivan",
           last:  "Toledo" }
```

Constructors

```
function Person(first, last) {
  this.first = first;
  this.last  = last;
}

var me = new Person( "Sivan",
                     "Toledo" );
```

A normal function

When invoked through the new operator, a new object is allocated and is bound to `this`

Objects with Methods

```
function Person_age() {
  var today = new Date();
  return ( today.getFullYear()
           - this.birth_year ); }

function Person(first, last, byear) {
  ...
  this.birth_year = byear;
  this.age = Person_age;
}
...
alert("My age is " + me.age() );
```

The Prototype Property

```
function Person(first, last, byear) {
  ...
  this.birth_year = byear;
  // this.age = Person_age;
}
Person.prototype.age = function() {
  ...
}

...

alert("My age is " + me.age() );
```

Prototype Rules

When JavaScript invokes a constructor (through `new`), it copies the prototype property of the constructor to the new object; so all the objects that a constructor creates share its prototype

When JavaScript is trying to read a property of an object (including a method, as in `me.age()`) and does not find it in the object, it searches the prototype, then the prototype of the prototype, etc

More Prototype Rules

When JavaScript is trying to write a property that does not exist in an object, it adds the property to the object, never to the prototype; to add property p to the prototype of an object o, use
`o.prototype.p = ...`

When JS creates a new function, its prototype is an empty `Object`; so all "classes" inherit from Object methods like `toString()`

Can use this to set up class hierarchies

Object-Oriented JavaScript Summary

- Classes are represented by a constructor function and the properties of its prototype object
- The prototype is initially an empty Object, but we can set it to another object to inherit its properties
- Essentially a way to define single inheritance
- The objects that a single constructor creates can have different properties through conditional or late definitions and through `delete`

More Object-Oriented Details

The `constructor` property of the prototype normally refers to the constructor; can be used to construct more objects like it and as a reflection mechanism (is this object a `Person`?)

`toString()` and `toLocaleString()` convert to string, initially by inheritance from Object

`valueOf()` should convert to a primitive value (e.g., number) when possible

Arrays versus Objects

Arrays are special; they are not objects

Indexes are non-negative integers < 2^{32}-1

```
a[-3.14] = ...
```

Creates a property, not an array element

```
bond[7] = true
```

Creates a property named "7", not an array element

Length of Arrays

`length` is the index of the last element + 1

Arrays may be sparse (implementation dependent)

Assigning to an element beyond last lengthens the array

Array can be lengthened and shortened by assignment to the `length` property

Array Methods

```
a.join(), a.join("\t")
```

Returns a string with a concatenation of the elements with the given separator (or comma)

```
a.sort(), a.sort(comparison_function)
```

Sorts the array in place; the comparison function should return a number <0, 0, or >0

```
a.reverse() // in place
```

More Array Functions

`a.concat(...)`
Concatenates more elements of arrays to the end of the array

`a.slice(begin,end)`, `a.slice(begin)`
Returns a copy of a section of the array from `begin` to `end-1` (or to the end); arguments are integers; negative arguments are length from the end

Splicing Arrays

```
a.splice(delete_from_here)
a.splice(delete_from_here, how_many)
```

Deletes a section of an array, from a given location to the end, or a given number from a location; deletes in place and returns the deleted section

```
a.splice(dfh, hm, ...)
```

Inserts new elements (arguments 3 and up) instead of the deleted section of the array (which may be empty)

Arrays as Stacks and Queues

`a.push(element), a.unshift(element)`
Appends an element to the **end** (push) or **beginning** (unshift) of the array; both return the new length

`a.pop(),a.shift()`
Returns and deletes the **last** (pop) or **first** (shift) element of an array

Regular Expressions

Patterns that match strings

Objects with a special literal syntax

```
var pattern1 = /Java/;
var pattern2 = new RegExp( "Script" );
```

Usage example:

```
var s  = "35,55";
var xy = s.match(/([0-9]+),([0-9]+)/);
var x  = Number(xy[1]);
var y  = Number(xy[2]);
```

Specifying Patterns: Literal Characters

Most letters and digit match themselves

`\u05d0` for Unicode characters

`\n, \r, \0, \x20, \cM, ...`

Special characters:

`^ $ . * + ? = ! : | \ / ( ) [ ] { }`

To specify any of them, use `\^`, `\$`, etc.

Character Classes

User-defined classes:

`[abc]`	a or b or c
`[^abc]`	not a, b, or c
`[a-z]`	one letter a to z
`[a-zA-Z]`	one letter, upper or lower case

Predefined classes (Unicode and ASCII):

`.`	any except newline characters
`\w, \W`	ASCII word char, not ASCII word
`\s, \S`	any whitespace char, not whitespace
`\d, \D`	ASCII digit, not ASCII digit

Specifying Repetitions

`/a*/`	zero or more a's
`/a+/`	one or more a's
`/a?/`	zero or one a's
`/a{7}/`	exactly 7 a's
`/a{3,6}/`	at least 3 a's and at most 6
`/a{4,}/`	at least 4 a's

These are all greedy; they will match as many repetitions as possible

There are non-greedy specifications (append ?)

Specifying Alternates

```
/good|bad/
```

Matches either "good" or "bad"

Matching is left-to-right; if "good" matches we ignore the other possibilities (even if they match a longer substring)

Parentheses in Regular Expressions

Parentheses are used for two separate purposes

```
/Sivan( Toledo)?/
```

Grouping: "Sivan" or "Sivan Toledo"

```
/(['"])[^\1]*\1/
```

Referring to the substring that matched a parenthesized part of the pattern; here \1 refers to the text (either ' or ") that matched the first parenthetical sub-pattern

More Parentheses

Most importantly, the pattern-matching interface can tell the program what substring matched each parenthetical sub-pattern,

```
var s  = "35,55";
var xy = s.match(/([0-9]+),([0-9]+)/);
var x  = Number(xy[1]);
var y  = Number(xy[2]);
```

We can also use parentheses only for grouping:
`/(?:...)+/`

The Position of a Match

`/^#define/`	a #define at the beginning of the
	string, or the beginning of a line
	in multi-line mode (see later)
`/\\/`	a \ at the end of the string/line
`\b`	word boundary: between a \w and a
`/a?/`	\W, or between a \w and the end or
`\B`	not a word boundary
`(?=...)`	look-ahead pattern; make sure it's
`(?!...)`	there but don't match; negative l-a-h

Match Flags

`/Sivan/i`	case **in**sensitive (ignore upper/lower case)
`/\./g`	global, all the occurrences of a period
`/^#def/m`	multiline, ^ and $ are relative to lines

Flags can be combined, as in `/Zebra/gi`

Using Regular Expressions: Searching

```
var s = "ABC..."; // must be a string
```

```
var position = s.search(/XYZ/i);
```

Returns the position of the first substring that matches the pattern, or -1 if the pattern did not match

Ignores the global flag (returns at most one match)

Using Regular Expressions: Search & Replace

```
s.replace(/sivan/i, "Sivan");
```
Replaces the first occurrence of "Sivan" (any upper/lower combination) with "Sivan"

```
s.replace(/sivan/gi, "Sivan");
```
Replaces all occurrences

```
s.replace("sivan", "Sivan");
```
Faster without a regular expression (but `search` is not; it always converts its argument to a RegExp)

Using RegExp's: Advanced Search & Replace

Occurences of `$1`, `$2`, ..., `$99` in the second argument of `String.replace` are replaced with matched substrings:

```
s.replace(/'([^']+)'/g, "\"$1\"");
```

Replaces 'some text' with "some text"

A few more $ expressions (including $$ for a literal $ sign)
A function as a second argument; computes a replacement given the matched substring

Using Regular Expressions: Matching

```
var m = s.match( /\d+/g );
```

Returns an array of all the matches (numbers)

```
var xy = s.match(/([0-9]+),([0-9]+)/);
```

Returns an array; `xy[0]` is the entire match, and the other elements of the array are substrings that matched parenthetical sub-expressions

Without a global flag, the `index` property of the output array holds the location of the match (the output of `s.seach`)

Using Regular Expressions: Splitting

```
var authors = "Smith , Cohen, and Ron";

var authors_array = authors.split(
      /\s*,\s*(and)?\s*\ );
```

returns `["Smith", "Cohen", "Ron"]`

Also a simpler version with a a string separator:

```
var blue = "0,0,1.0";

var rgb  = blue.split( "," );
```

Regular Expression Objects

Created using a literal or the RegExp constructor

`r.exec(string)` is similar to `string.match(r)`, but if invoked repeatedly on string, it returns a sequence of matches, not the same one repeatedly; it remembers where to search using its `lastIndex` property

`r.test(string)` is similar, but only returns a boolean value

Explicit & Implicit Data Type Conversions

```
var n = Number("32");
var b = Boolean(0);
var s = String(32);

var n = s – 0;  // convert to number
var b = !!n;    // to boolean
var s = n + ""; // to string
```

Advanced Conversions

```
var n = 33;
var s;
s = n.toString();   // "33"
s = n.toString(2);  // binary "100001"
s = s.toString(16); // hexadecimal

n = 314.15927;
s = toFixed(1);     // "314.2"
s = toExponential(2); // "3.14e+2"
s = toPrecision(2);   // "3.1e+2"
s = toPrecision(4);   // "314.2"
```

Advanced Conversions to Number

```
n = parseInt("33");             // 33
n = parseInt("33 cm");          // 33
n = parseInt("0x21");           // 33
n = parseInt("21",16);          // 33
n = parseInt("100001",2);       // 33
n = parseInt("garbage");        // NaN

n = parseFloat("3.14 Km/s"); // 3.14
```

Library Classes

```
Object
  Arguments
  Array
  Boolean
  Date
  Error: EvalError,RangeError,URIError,
   ReferenceError,SyntaxError,TypeError
  Function
  Math
  Number
  RegExp
  String
```

Global Scope Properties & Functions

```
Infinity      NaN
undefined

decodeURI()   decodeURIComponent()
encodeURI()   encodeURIComponent()
escape()      unescape()
eval()
isFinite()    isNaN()
parseInt()    parseFloat()
```

And the constructors listed above