JavaScript

types: Number, String, Boolean, Object, null, undefined.

- Number (64-bit floating point) = Double
  - are represented approximately, which means: 0,1 + 0,2 ≠ 0,3
  - has a value "NaN" - Not a number
  - NaN is not equal to anything, even NaN ≠ NaN (true)

Number('_') - converts string to number
  similar to "+" prefix operator: (+"42") = 42

parseInt('_') - converts string to number, but:
  parseInt('12cm') = 12 (it stops at the first non-digit char)
  parseInt("08") = 0
  parseInt('08', 10) = 8

- String is a sequence of 0 or more 16-bit characters
  - Strings are immutable
  - similar strings are equal (===)
  - no difference between '' and ""
  - String(number) - converts number to string
  - methods: charAt, concat, indexOf, lastIndexOf, match, replace, search, slice, split, substring, toLowerCase, toUpperCase

- Boolean(value) - return true if value is truthy & false otherwise
  similar to (!!) prefix operator

- Null - a value that isn't anything
- Undefined - a value that isn't even that
  - the default value of vars & params
  - the value of missing member in object

Falsy values: false, null, undefined, "", 0, NaN

Truthy values: all other values (including all objects) are truthy

---

- Objects
  - Every thing else in a dynamic objects
    - new Object() => { }
    - a name of prop can be string
    - a value of prop can be any value except undefined
    - members can be accessed with dot notation (".") or subscript notation ("[]")

Loosely typed: any of these types can be stored in an variable, or passed as a parameter to any function.

- Operator +: If both operands are numbers
  then
    add them
  else
    convert them both to string
    concatenate them

- Devision can produce a non-integer result even operands are integer
  10/3 = 3,33... ⟶
  watch out!

- Tails of type coersion:

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

false == undefined // false
false == null      // false
null == undefined  // true

false == 'false'   // false
false == '0'       // true.

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

=== operator will give false for this ⟶ expressions

- && Operator: if first operand is truthy
  then result is next operand
  else result is first operand

- || operator: if first operand is truthy
  then result is first operand
  else result is ~~second~~ next operand

- Bitwise operators: & | ^ >> >>> <<

For in statement

```
for (var name in object)
  if(object.hasOwnProperty(name)){
    // within the loop
    // name is the key of current member
    // object[name] is the current value
  }
```

Throw statement: - throw new Error(reason)
- throw {
    name: ~
    message: ~
  }

JavaScript can produce these exceptions: Error, EvalError, RangeError, SyntaxError, TypeError, URIError.

Operator ~~with~~ with:
~~recommend~~ not use!

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

⟶ with do ⟶ any of

```
o.foo = koda
o.foo = o.koda
foo = koda
foo = o.koda
```

- Scope
  - {blocks} do not have scope
  - only functions have scope
  - vars defined in a function are not visible outside of the func
  - define all vars at the beginning of a func
- Return statement
  - return exp. (if there is no exp, then the return value is undefined)
  - return; (except for constructors, whose default return value is this)
  - every function always returns a value, if you dont say, it is undefined

- Objects. Everything else is objects
  - Objects can inherit from other objects.
  - Object is an unordered collection of name/value pairs
  - subscript notation is used to extract a prop with reserved word
    myObjects['goto'];

- if we have a function superDiv(width, height) {...}
  we can pass it an object {width: ..., height: ...}
  and it will work

- Linkage
  - Objects can be created with a secret link to another object, like so:
    Object.create(ob) - makes a new empty object with a link to object "ob"
  - If an attempt to access a name fails, the secret linked object will be used
  - The secret link is not used when storing. New members are only added to the primary object
  - Linkage provides simple inheritance

- Object methods:
  - all objects are linked directly or indirectly to Object.prototype
  - all objects inherit some basic methods.
  - hasOwnProperty(name) - means: "is the 'name' a true member of this object?"
  - no copy method
  - no equal method

- Make an empty object
  - new Object()
  - {}
  - Object.create(Object.prototype)

- Reference
  - Objects are passed by reference
  - The === operator compares object references, not value
    true only if both operands are the same object

- Delete
  Members can be removed from an object with delete operator
  delete myObj[name];
  delete myObj.name

- Arrays
  - inherits from Object
  - my = ['one', 'two', 'three'] ⇒ {'0': 'one', '1': 'two', '2': 'three'}
  - advantage: No need to provide a length or type when creating an array
  - dont use for..in with arrays. It works but members wont come out in right sequence.
  - ~~myArray = {}~~ also we can create an array with: new Array().
  - array methods: concat, join, pop, push, slice, sort, splice
  - delete element: array.splice(from, length);
    If we: delete array[1] then we will get "undefined" in pos 1

Functions:

- inherits from Object and can store name/value pairs
- var foo = function foo() {...}
- functions can be defined inside of other functions
- Static scoping or Lexical scoping: the inner func has access to the vars and params of func that it is contained within.
- Closure - the scope that an inner func enjoys, continues even after the parent func have returned.
  - programming pattern
- If a func is called with too many args, the extra args are ignored.
- If a func is called with too few args, the missing values will be undefined
- 4 ways to call a func:
  - function form: functionObject(args) (this is set to the global object)
  - method form: thisObj.methodName(args) (this is set to thisObj)
    thisObj['methodName'](args)
  - constructor form: new FuncObj(args) (new obj is created & ~~this~~ assigned to this)
  - apply form: funcObj.apply(thisObj, [args])
- In constructor call form if there is no an explicit return value ⇒ this will be returned.
- this is an extra param. Its value depends on the calling form
- When a function is invoked, in addition to its params, it also gets a special param called arguments: any[], which contains all of the args from invocation

- Typeof operator returns a string identifier the type of a value.

  Note: typeof(array) = ~~object~~
  typeof(null) = object

- You can extend any built-in type by: Number.prototype
  example: String.prototype.trim = function() { ... }
- Distinguish array
  Array.isArray(myArr);
- eval function:
  - compiles & executes a string & returns the result
  - it is what the browser uses to convert strings into actions
- Avoid: new Boolean(), new String(), new Number().
- (global) Object is the container for all global variables & all built-in objects
  - Sometimes this points to it
  - on browser, window is the global object
- Any variable with is not properly declared is assumed to be global by default.
- every object is a separate namespace
- Anonymous functions can be used to wrap your app:
  (function() { ... }());
- Good & safe pattern without global vars:

```
MYAPP.Trivia = (function () {
    // define your common vars here
    // define your common functions here
    return {
        funcOne: function(...) { ... },
        funcTwo: function(...) { ... }
    };
}());
```

- Javascript has no cast operators
- new Date() - return current date
- RegExp:
  - patterns are enclosed in slashes
- the language definition is neutral on threads.

Inheritance: is object-oriented code reuse

Two schools: Classical, Prototypal

- Classical inheritance: objects are instances of Classes. A class inherits from another class.
- Pseudoclassical in Javascript, because instead of classical approach it uses operators that look classical, but behaves prototypally.
- There are 3 mechanisms to create a class & instance
  - Constructor:
    function Constructor() { }
  - The prototype member of function:
    Constructor.prototype.someMethod = function (...) { ... }
  - The new operator: var newObj = new Constructor();

- new operator: new Constructor() returns a new object with a link to Constructor.prototype.

  new Constructor = Constructor.prototype

- If the new is omitted, the global object is clobbed by the constructor.
- when a function object is created, it is given a prototype member which is an object containing a constructor member which a reference to the function object.
- In order to make prototype more friendly; use this pattern:

```
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};

Constructor.method('first_method', function (a, b) { ... })
```

- If we replace the original prototype object with an instance of an object of another class, the we can inherit another's class stuff
- public method is a fuction that uses this to access its object.

Prototypal Inheritence:

- Instead of classical inheritance JavaScript has Prototype inheritance it accomplish the same thing, but differently.
- Prototypal inheritance is about using Object.create(inherit from Object)
- An object contains a secret link to parent object

```
var oldObj = {
    firstMethod: function () {...}
    ...
}
var newObject = Object.create(oldObj)
newObj.secondMethod = function() {...}
var myObj = Object.create(newObj),
myObj.firstMethod();
```

- If newObj has foo() then chain will not be consulted when accessing member foo.
- If access of a member of newObj fails, then search for the member in oldObject.
- If that fails, then search for the member in Object.prototype.
- Changes in oldObj may be immediately visible in newObj.
- Changes to newObj have no effect on oldObj.
- Implementation of Object.create:

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        function F() {};
        F.prototype = o;
        return new F();
    };
}
```

- Namespace is a an object

Functional inheritance:

- How to create an object:

```
function myPowerConstructor (x) {
    var that = classMaker (x); // it can be new, {} or Object.create
    var secret = f(x); // private method
    var priv = function () { ... secret.x (that ...) }; // public method
    return that;
}
```

- Functional inheritance: example is based on using greater func inside others

```
function symbol () {
    return {...};
}

function delim (s) {
    var tmp = symbol();
    tmp.method = ...;
    tm.field = ...
    return tmp;
}
```

or

```
function gizmo (id) {
    return {...}
}

function hoozit (id) {
    var that = gizmo (id);
    that.test = function (test) {
        return ...
    }
    return that;
}
```

- Pseudo class inheritance example:

```
function Gizmo (id) { this.id = id; }
Gizmo.prototype.toString = function () {
    return "gizmo" + this.id;
};
function Hoozit (id) { this.id = id; }
Hoozit.prototype = new Gizmo ();
Hoozit.prototype.test = function (id) {
    return this.id === id;
}
```

★ Conclusion:

- Prototypal inheritance works really well with public methods.
- Functional inheritance works really well with privileged and private & public methods.
- Pseudo classical inheritance for elderly programmers who are old and set in their ways

- when creating function in a loop, don't let the function directly use any vars that are mutated by the loop.
- Use closure to protect the loop vars

```
while (...) {
    function (...) {
        x ... y ... z
    }
}
```

// here you use one instance of func for all through the loop

⇒ use this ⇒

```
while (...) {
    ... function (x, y, z) {
        return function (...) {
            x ... y ... z
        };
    } (x, y, z) ...
}
```

// while here you create new instance of func in every iteration.

EcmaScript 5

- Strict mode: "use strict"
  - removes some of the most problematic features from the language
    - Limitations on eval, arguments, delete, with
  - possible performance benefit.
  - for safety
  - If you put "use strict" its scope would be that file.
  - If you put it in function its scope will be function.
- Syntax Relaxation:
  - object.for & object.class are now legal.
  - {extra: "comma",} is now legal
  - [1,2,3,].length == 3
  - <BOM> is now whitespace
- Strings:
  - string.trim()
  - \<CLF> now allowed in string literals (line breaks)
- Arrays:

```
Array.isArray(arr)
array.indexOf(searchEl, fromIndex)
array.lastIndexOf(searchEl, fromIndex)
array.every(func, thisArg)   // context
array.some(func, thisArg)
array.forEach(...)
array.map(...)
array.filter(...)
array.reduce(...)
array.reduceRight(...)
```

- Objects:

```
Object.create(obj, props)
Object.keys(obj)
Object.getOwnPropertyNames(obj)
Object.getPrototypeOf(obj)   // get __proto__
Object.defineProperty(obj, name, attrs)
Object.defineProperties(obj, props)
Object.getOwnPropertyDescriptor(object, name)
Object.preventExtension(obj)   // can't add new props
Object.seal(obj)   // can't add new props & can't configure existed props
Object.freeze(obj)
Object.isExtensible(obj)
Object.isSealed(obj)
Object.isFrozen(obj)
```

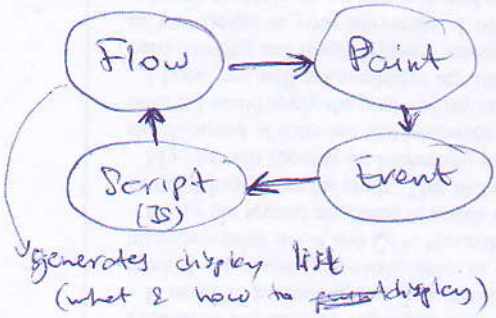
- Dates:
  - support for ISO dates
  - Date.now()
  - Date.toISOString()
- Functions
  - function.bind(this, args)... // Binds context to function
  - function.apply no longer wraps/or replaces its this parameter //
- arguments
  - arguments still isn't an array, but it is more array-like.
  - arguments now inherits from Array.prototype, so these two statements do the same things:

    arguments.slice(2)

    Array.prototype.slice.apply(arguments, [2])
  - arguments is still dangerously linked to parameters
- Regular Expressions:
  - RegExp literals now produce unique objects
  - \s now describes all of the whitespaces chars recognized by the language
  - /[/]/ is now legal.

DOM

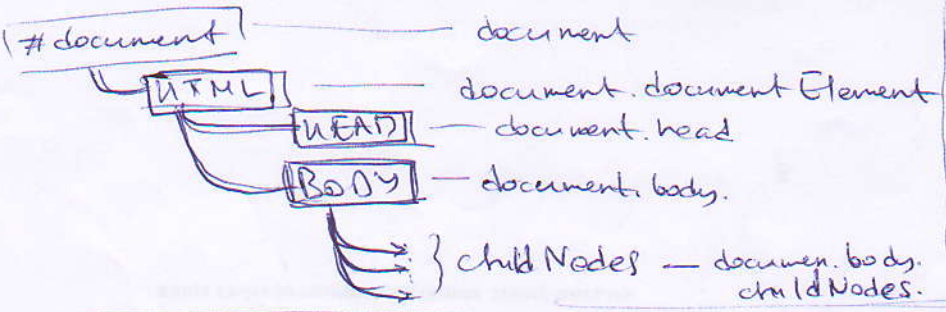
Scripted Browsers works like this:



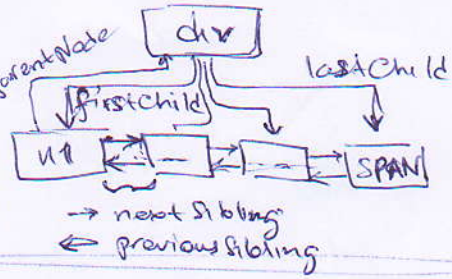
- put script tags in bottom, because it freeze web while its downloaded & executed.
- reduce the number of script files as much as possible (which reduces number of HTTP calls)
- document.write:
  - Allows JS to produce HTML text.
  - Before onload: inserts HTML text into the document.
  - After onload: uses HTML text to replace the current document.
  - Not recommended.
- Collections:

  document.anchors, document.applets, document.embeds, document.forms, document.frames, document.images, document.plugins, document.scripts, document.styleSheets
- document.all — acts as a function or array for accessing elements by position, name, or id
- document.getElementById(id)
  document.getElementsByName(name)
  document.getElementsByTagName(tagName)

- Document Tree:



- Every node has two child pointers



- Walk the DOM:

```
function walkTheDOM(node, func) {
    func(node)
    node = node.firstChild;
    while(node) {
        walkTheDOM(node, func);
        node = node.nextSibling;
    }
}
```

- document.getElementsByClassName(className)

- ~~Body~~
- Manipulating Elements:
  - my_div.getAttribute('src')
  - my_div.setAttribute('src', superurl);
- Style:
  - node.style."stylename"
  - node.className
- Style Names:

CSS	JavaScript
font-size	→ fontSize
z-index	→ zIndex
...	

- Making Elements:

```
document.createElement(tagName)
document.createTextNode(text)
node.cloneNode()    // Clone an individual Element
node.cloneNode(true) // Clone an element and all of it children
```

  - the new nodes are not connected to the document.
- Linking Elements:

```
node.appendChild(new)
node.insertBefore(new, sibling)
node.replaceChild(new, old)
```

- Removing Element ~~Child~~

```
node.removeChild(old)
// it returns the node
```

  - Be sure to remove any event handlers.
- innerHTML – for manipulating inner HTML of tag
  It is faster that above methods for node manipulating.

Events:

- the browser has an event-driven, single-threaded, async programming model
- events are targeted to particular nodes.
- Events cause the invocation of event handler functions.

- Event Handlers:
  - node.addEventListener(type, func, false);
- Bubbling – means that the event is given to the target, and then its parent, and then its parent, and so on until the event is cancelled
- Cancel Bubbling
  event.cancelBubble = false;
  if (event.preventDefault) e.preventDefault();
  return false;
- Prevent Default Actions:
  an event handler can prevent a browser action associated with the event (such as submitting a form)
  event.returnValue = false;
  if (event.preventDefault) event.preventDefault();
  return false;
- Memory Leaks
  - memory mangement is automatic
  - it is possible to hang on to too much state, preventing it from being garbage collected.
- alert(text), confirm(text), prompt(text, default)
  These functions break the async model
  Avoid these in Ajax apps.
- Every window, frame and iframe has its own unique window object
- Global object, self or parent or top, is also called
- Inter-Window communication
  frames[] – Child frames and iframes
  name – Text name of window
  opener – Reference to opener
  parent – Reference to parent
  self – Reference to this window
  top – Reference to outermost
  window – Reference to this window
- open() – Open new window
- a script can access another window if
  it can get a reference to it, if
  document.domain == otherwindow.document.domain.
- Browser Detection
  navigator.userAgent – It lies, because browsers are not honest with users
- It is faster to manipulate new nodes before they are attached to the tree.
- Always optimize the part which takes most time of all.
- A trick that is faster on Browser A might be slower on Browser B.
- Avoid short-time optimization
- Don't optimize without measuring:
  start_time = new Date().valueOf();
  code_to_measure();
  end_time = new Date().valueOf();
  elapsed_time = end_time - start_time.
- measure multiple times

- An operation that is performed only once (O(1)) is not worth optimizate
- An operation that is performed many times (O(n)) may be worth optimizate
- Bad algorithms: O(n log(n))
  O(n²)
- The most effective way to make programs faster is to make it smaller

Security

Bad parts:
- Global variables
- + adds & concatenates
- semicolon insertion (;)
  if compiler get an error (syntax) while parsing your program, it will rewind change the nearby line to ";" & try again
- ~~typeof~~
- with & eval
- phoney arrays // ~~ugly~~ slow
- == & !=
- false, null, undefined, NaN // sometimes they are interchangable, because of ==
- for_in statement mixes intended functions with the desired data members ~~Style~~

Simple optimization with closure:

```
var names = [ ... ]; // global array = bad!
var someFunc(n) {
    return names[n];
};
```

⇓ Optimization

```
var someFunc(n) {
    var names = [ ... ]; // every time we execute function we redeclare "names"
    return names[n];                    = inefficient!
};
```

⇓ Optimize

```
var someFunc = (function() {
    var names = [ ... ]; // we declare "names" only once
    return function(n) {
        return names[n];
    }
});
}());
```

JSON

- JSON.parse(text, reviver)
  - text: text to be parsed
  - reviver: function which will be called for each of the values of in the new object, giving opportunity to modify the result

```
function reviver(key, value) {
    return new-value;
}
```

- JSON.stringify(value, replacer, space)
  - value need to be stringified
  - replacer is an optional function that will be called for each of the values, giving an opportunity to modify the result
  - space: is an optional pretty printing param.

Instead of replacer you can pass an array with keys of future JSON.

~~XML~~

- JSON is very stable;
- JSON has no version number
- Superset of JSON: YAML, JavaScript
- JSONT — JSON Transform

```
var rules {
    self: "— {color} — {closed} — {points} —"
    closed: function(v) { return v? "X" : "-"; }
    'points[*][*]': '{$}'
}

var data = {
  "color": "blue",
  "closed": true,
  "points": [ [10, 10], [20, 20], — ]

jsonT(data, rules); // => you get => "— blue — X — 10 10 20 20 —"
// it applies data to rules. & returns a result
```

- Don't wrap JSON text in comment