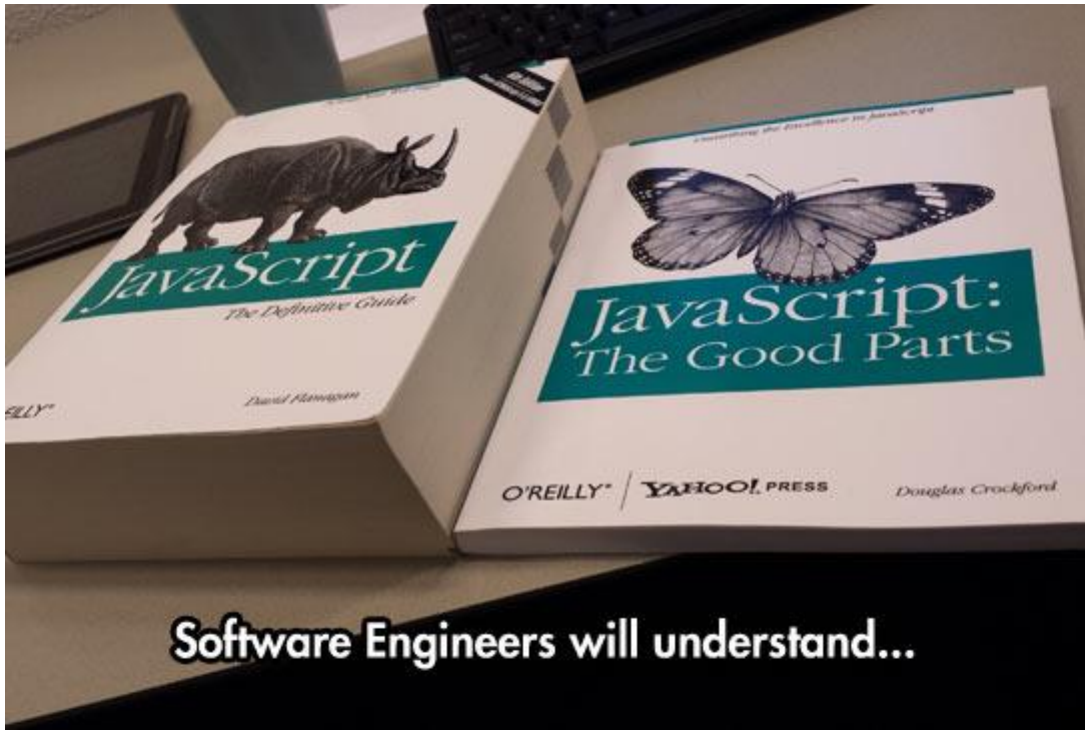
Effective Javascript

JS is Great

But you must be in charge…



Exceptions

- When your javascript makes an error (e.g., referencing an unknown variable or function), an Exception is thrown.
- This gives us a chance to catch and handle the problem.
- Lets look at try/catch

Throwing Exceptions

- You may throw exceptions in your code.
  - Exception can be anything – a string, a number or any object.
  - Best practice is to throw new Error(‘’)

```
function foo(x){
   try  {
     if (x > 100) throw "TooBig";
     if (x < 20)  throw "TooSmall";

   } catch(error)  {
      if (error == "TooBig") alert("Number is too Big");
      if (error == "TooSmall") alert("Number is too Small");
   }
{
```

Handling Exceptions

- Consider the following code:

```
function foo(){
   try  {
     goo(7);
   } catch(error)  {
     txt = "Error: " + error.description + "\n\n";
     alert(txt);
   }
}
```

  - What if goo was ran in setTimeout?

Handling Exceptions

In modern browser we can also use window.onerror as the global handler

```
window.onerror = function (err){
      console.log(err);
}
```

typeof

Lets Have a look at the different types in Javascript

```
//Numbers
typeof 12 === 'number';
typeof 3.14 === 'number';
typeof Infinity === 'number';
typeof NaN === 'number'; // Despite being "Not-A-Number"

// Strings
typeof "" === 'string';
typeof "bla" === 'string';
typeof (typeof 1) === 'string'; // typeof always return a string

// Booleans
typeof true === 'boolean';
typeof false === 'boolean';

// Undefined
typeof undefined === 'undefined';
typeof blabla === 'undefined'; // an undefined variable
```

typeof

More:

```
// Objects
typeof null === 'object';
typeof {a:1} === 'object';
// use Array.isArray to differentiate regular objects from arrays
typeof [1, 2, 4] === 'object';
typeof new Date() === 'object';

// Functions
typeof function(){} === 'function';
typeof Math.sin === 'function';

typeof Boolean(true) === 'boolean';
typeof new Boolean(true) === 'object'; // this is confusing. Don't use!

typeof Number(1) === 'number';
typeof new Number(1) === 'object'; // this is confusing. Don't use!

typeof String("abc") === 'string';
typeof new String("abc") === 'object';  // this is confusing. Don't use!
```

Prefer Primitives to Object Wrappers

- In addition to objects, JavaScript has five types of primitive values:
  - booleans, numbers, strings, null, and undefined.
- Be careful from the primitive wrappers:

```
var s = new String("hello");

typeof "hello"; // "string"
typeof s;  // "object"
```

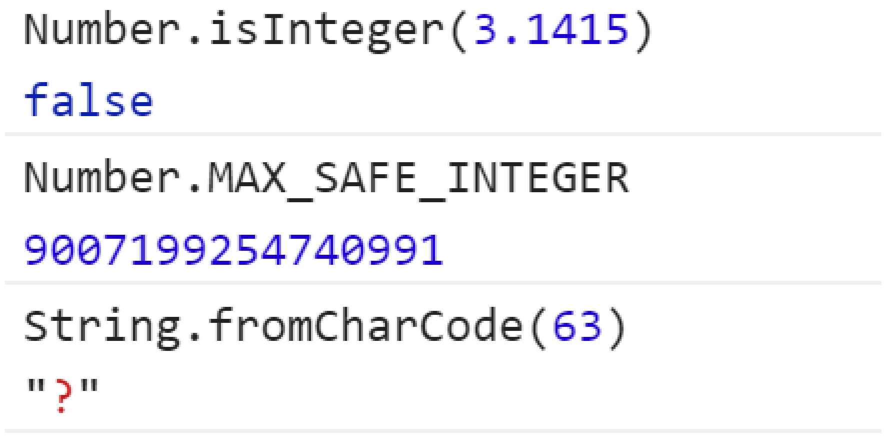
The Primitive Wrappers are tricky

Be careful!

```
var s1 = new String("hello");
var s2 = new String("hello");
s1 == s2;    // false

var n1 = new Number(3);
var n2 = new Number(3);
n1 == n2;    // false
```

Prefer Primitives to Object Wrappers

- Since these wrappers don’t behave quite right, they don’t serve much of a purpose.
- The main justification for their existence is their utility methods.



```
Number.isInteger(3.1415)
false
Number.MAX_SAFE_INTEGER
9007199254740991
String.fromCharCode(63)
"?"
```

Strict mode

- It is advised to use the strict mode when developing javascript.
  - Syntax is a bit funny, but it will make you write better code
  - For example, it will not allow use of uninitialized variables
  - Beware of concatenating scripts that differ in their expectations about strict mode.

```
'use strict';
var namo = 'Arik';
alert("Hello " + name);
```

JavaScript’s Floating-Point Numbers

JavaScript numbers are double-precision floating-point numbers

Integers are just a subset of doubles rather than a separate datatype

As in other language, there are side effects in arithmetic:

```
typeof 17; // "number"
typeof 98.6; // "number"
typeof -2.1; // "number"

0.2 + 0.1; // 0.30000000000000004

0.3 + (0.2 + 0.1); // 0.6000000000000001
(0.3 + 0.2) + 0.1; // 0.6
```

Implicit Coercions

- Beware of Implicit Coercions

```
3 + true; // 4
(1 + "2") + 3 // “123”
3 * "17"; // 51
"1" | "8"; // 9
```

- Use === to make it clear to your readers that your comparison does not involve any implicit coercions

Implicit Coercions

Know your NaN..

```
var x = NaN;
x === NaN; // false

isNaN(NaN); // true
isNaN("foo"); // true
isNaN(undefined); // true
isNaN({}); // true
isNaN({ valueOf: "foo" }); // true

function isReallyNaN(x) {
   return x !== x;
}
```

Look how Angular2 Change detection logic needs to overcome this… https://github.com/angular/angular/blob/50548fb5655bca742d1056ea91217a3b8460db08/modules/angular2/src/facade/lang.ts#L367

Implicit Coercions

Objects are coerced to numbers via valueOf and to strings via toString.

```
"J" + { toString: function() { return "S"; } }; // "JS"
2 * { valueOf: function() { return 3; } }; // 6
```

Anonymous functions

- The use of anonymous functions in javascript is very common.
- See the following examples:

```
var foo = function(name) {alert('foo ' + name);};
foo('me');

window.setTimeout(function(){alert("Time's up!")}, 3000);
```

Closures

- Accessing variables outside of your immediate lexical scope creates a closure.
- In other words, the function defined in the closure 'remembers' the environment in which it was created in.

```
function init() {
  var name = “Puki"; // name is a local variable created by init
  function displayName() { // displayName() is the inner function, a closure
     alert (name); // displayName() uses variable declared in the parent function
  }
  displayName();
}
init();
```

Closures are Powerful

Functions that keep track of variables from their containing scopes are known as closures.

```
function sandwichMaker(breadType) {
  function make(filling) {
    return breadType + " and " + filling;
  }
  return make;
}

var shifonAnd = sandwichMaker("Shifon");
shifonAnd("cheese"); // Shifon and cheese"
shifonAnd("mustard"); // Shifon and mustard"
```

Closures

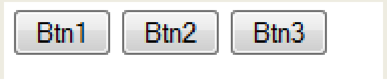
Here's another example:

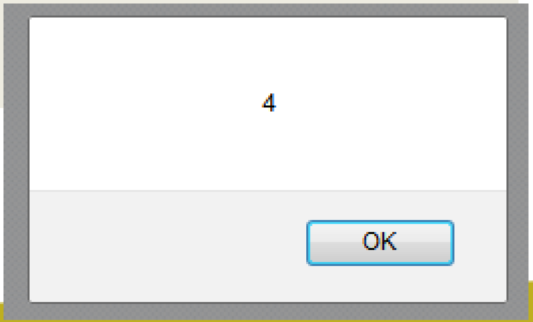
```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2));  // 7
console.log(add10(2)); // 12
```

Common Closures pitfall

a common problem with closures occurred when they were created inside a loop.

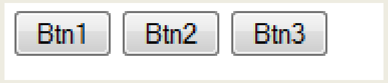
```
<button id="btn1">Btn1</button>
<button id="btn2">Btn2</button>
<button id="btn3">Btn3</button>

<script>
for (var i = 1; i <= 3; i++) {
     document.getElementById("btn"+i).onclick = function() {
                                                    alert(i);
                                                }
}
</script>
```




```
4
```

Common Closures pitfall solution

- a common problem with closures occurred when they were created inside a loop.



```
function alertIt(m) {
   return function() {alert(m)};
}

for (var i = 1; i <= 3; i++) {
    document.getElementById("btn"+i).onclick = alertIt(i);
}
```

Get Comfortable with Closures

- Understanding closures requires learning three facts:
  - Functions can refer to variables defined in outer scopes – these are called closures
  - Closures can outlive the function that creates them.
  - Closures internally store references to their outer variables, and can both read and update their stored variables.

Closure example

```
function Box() {
   var val = undefined;
   return {
      set: function(newVal) { val = newVal; },
      get: function() { return val; },
      type: function() { return typeof val; }
   };
}

var b = Box();
b.type(); // "undefined"
b.set(98.6);
b.get(); // 98.6
b.type(); // "number
```

Closure implications

```
for (var i=0; i<3; i++) {
   var link = document.createElement("a");
   link.innerHTML = "Link " + i;
   link.onclick = function () {
      alert(i);
   };
   document.body.appendChild(link);
}
```

Remember: Closures store their outer variables “by reference”, not by value.

Solution: Use IIFE

```
for (var i=0; i<3; i++) {
   var link = document.createElement("p");
   link.innerHTML = "Link " + i;
   link.onclick = (function (num) {
      return function () {
         alert(num);
      };
   })(i);
   document.body.appendChild(link);
}
```

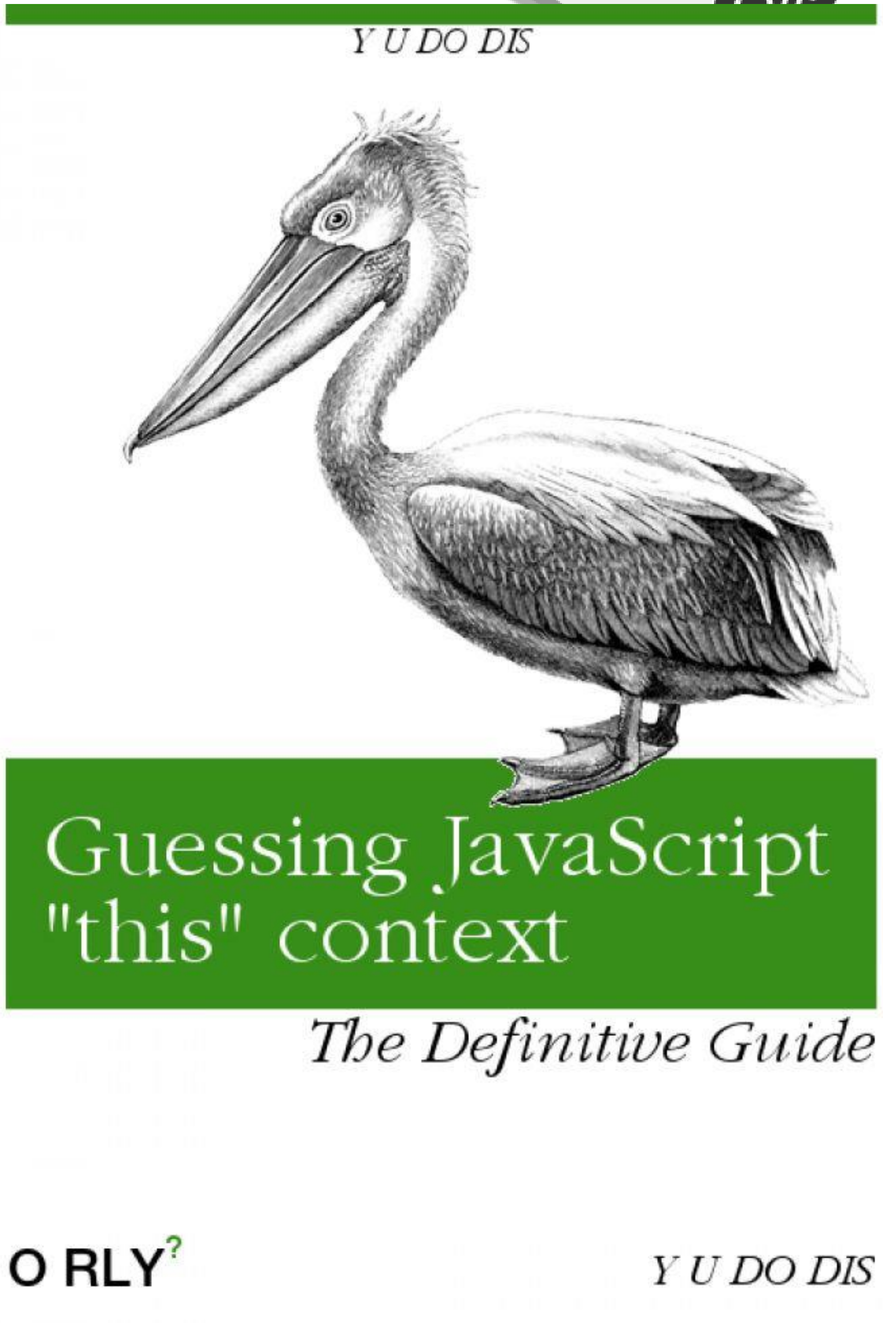
- Use Immediately Invoked Function Expressions to Create Local Scopes
- Beware of the implications of that (using *this* or *arguments*, *break*ing or *continu*ing…)
- Try this challange

Variable Hoisting

- JavaScript (Until ECMA6: *let*) does not support block scoping
- Variable declarations within a block are implicitly hoisted to the top of their enclosing function.
- Re-declarations of a variable are treated as a single variable.
- Consider manually hoisting local variable declarations to avoid confusion

```
var jsFuture = "es6";
(function () {
  if (!jsFuture) { var jsFuture = "es5"; }
  console.log(jsFuture);
}());
```

this?


Y U DO DIS
Guessing JavaScript "this" context
The Definitive Guide
O RLY?
Y U DO DIS

this

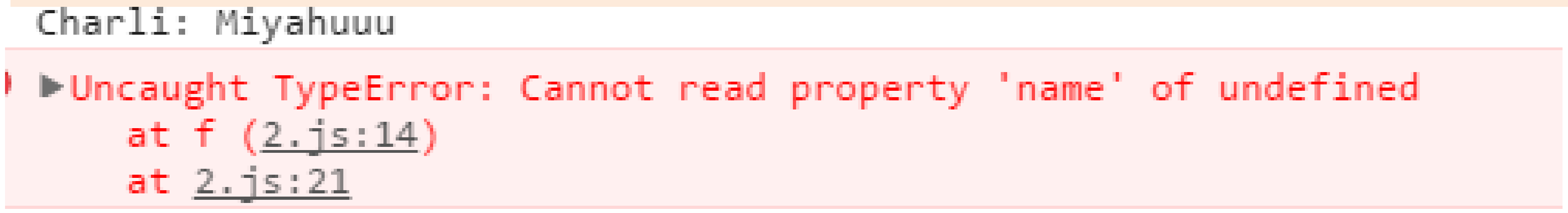
- The *this* keyword in JavaScript confuses new and seasoned JavaScript developers alike. Lets figure it out.

```
console.log(this.document === document); // true

// In web browsers, the window object is also the global object:
console.log(this === window); // true
this.a = 37;
console.log(window.a); // 37
```

- https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/this

this

```javascript
function f(){console.log(this.name + ': Miyahuuu');};
var pet = {
   name: 'Charli',
   speak: f
};

pet.speak();
f();
```



```
Charli: Miyahuuu
▶Uncaught TypeError: Cannot read property 'name' of undefined
    at f (2.js:14)
    at 2.js:21
```

- So, the value of this is determined by how a function is called
- Lets better understand different ways to call functions

Its all about functions

Functions & Methods

```
var obj = {
   // a method
   hello: function() {
      return "Hello " + this.username;
   },
   username: "Puki"
};
obj.hello(); // Hello Puki
```

- Functions are the working bees in javascript
- Methods in JavaScript are nothing more than object properties that happen to be functions

Using function.**call**

- Used for calling a function with a custom receiver.
- calling directly or indirectly, produces the same result:
  - *obj.foo(arg1, arg2, arg3);*
  - *foo.call(obj, arg1, arg2, arg3);*

- It is useful at times for one object to borrow the function of another object.
- meaning that the borrowing object simply executes the lent function as if it were its own.

```
function greet(thing) {
    console.log(this + " says: Hello " + thing);
}
greet.call("Puki", "World")  // Puki says: Hello World
```

Using function.**call**

- Useful Example: – the *arguments* object is not a regular array, never try to shift it or modify it in any way.
- If you need to invoke array functions on it (i.e.: forEach, map, etc) you can borrow those functions from Array:
-
```
[].slice.call(arguments);
```

function.**call** – did you get it?

- What will be printed here:

```
var person1 = {
    name: 'Puki',
    greet: function(other){console.log(this.name + ' says hello to: ' + other )}
}
person1.greet('Muki');

var person2 = {name: 'Aria'};
person1.greet.call(person2, 'Hodor')
```

Using function.**apply**

- The apply method takes an array of arguments and calls the function as if each element of the array were an individual argument of the call.
  - This is useful when you need to pass an array to a variadic function or method (unknown number of arguments)
  - Use the first argument of apply to provide a receiver for variadic methods.

```
var grades = getAllGrades();
calcAvg.apply(null, grades);
```

- If grades turns out to have, say, three elements, this will behave the
- same as if we had written:
- `calcAvg(grades[0], grades[1], grades[2]);`

Using function.**apply**

- Here is a useful example.
- In real projects it’s a good idea to use your own logging mechanism, so wrapping around console.log is needed:

```
function log() {
		console.log.apply(console, arguments);
}
```

  - Call it like log(‘info:’, obj);
  - Then that translates to **console.log.apply(console, [‘info‘, obj]);**
  - which is equivalent to **console.log('foo‘, obj)**;
  - which is what you want.
  - See robust solution [here]()

function.**apply** Did you get it?

- You have an array of nums and you need to find maximum.
- You rediscover that Math.max does not work on arrays,
  It’s a variadic function.
- Make it happen!

function.**bind**

- Function objects come with a bind method
  - This function returns a function
  - This function is usually a more specialized (limited) form of the initial function.

- Example:

```
Function greet(from, to, greeting) {
  console.log('from: ', from, 'to:', to, 'greeting:', greeting);
}

var yaronGreet = greet.bind(null, 'Yaron');
var yaronToYuvalGreet = greet.bind(null, 'Yaron', 'Yuval');
```

Using function.**bind** to fix a receiver

Function objects come with a bind method that
takes a receiver object
and produces a wrapper function
that calls the original function as a method of the receiver.

```
var buffer = {
   entries: [],
   add: function(s) {
      this.entries.push(s);
   },
   concat: function() {
      return this.entries.join("");
   }
};

var source = ["puki", "-", "muki"];
source.forEach(buffer.add.bind(buffer));
buffer.concat();
```

Using function.**bind** to curry functions

- The call to simpleURL.bind produces a new function that delegates to simpleURL.
- The first argument to bind provides the receiver value.
  (Since simpleURL does not refer to this, we can pass null)

```
function simpleURL(protocol, domain, path) {
   return protocol + "://" + domain + "/" + path;
}
var paths = ['a.txt', 'b.txt'];
var urls = paths.map(simpleURL.bind(null, "http", 'puki.com'));
```

Variadic functions

- Use the implicit *arguments* object to implement variable-arity functions.
- Note – the *arguments* object is not a regular array, never try to shift it or modify it in any way (you can make a copy if needed: [].slice.call(arguments))

```
function calcAvg() {
   for (var i = 0, sum = 0, n = arguments.length;i < n; i++) {
      sum += arguments[i];
    }
   return sum / n;
}
```

Quiz

- What goes wrong here?

```
function values() {
    var i = 0, n = arguments.length;
    return {
        hasNext: function() {
            return i < n;
        },
        next: function() {
            if (i >= n) {
                throw new Error("end of iteration");
            }
            return arguments[i++];
        }
    };
}
var it = values(3, 4, 2);
it.next(); // undefined
it.next(); // undefined
it.next(); // undefined
```

Quiz Solution

Fixed:

```
function values() {
    var i = 0, n = arguments.length, a = arguments;
    return {
        hasNext: function() {
            return i < n;
        },
        next: function() {
            if (i >= n) {
                throw new Error("end of iteration");
            }
            return a[i++];
        }
    };
}
var it = values(3, 4, 2);
it.next(); // 3
it.next(); // 4
it.next(); // 2
```

Check Point

- What is the output of that?

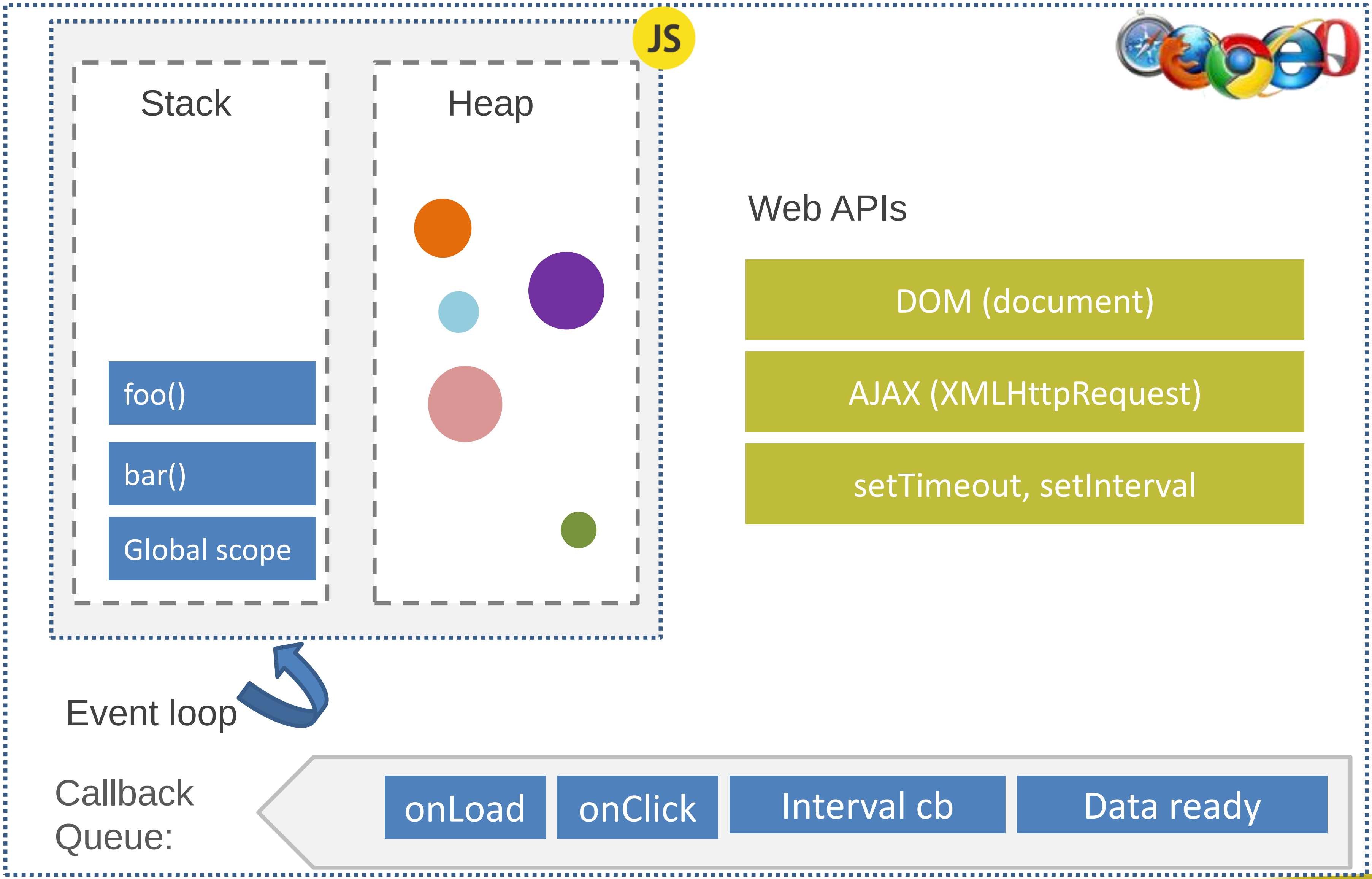
```
function variadicFunc() {
  console.log(‘args:', arguments);
}

var foo = variadicFunc.bind(null, 'Y‘, ‘M’)
foo('C', 'A');
```

Before Execution Model

Go Learn AJAX

Execution Model

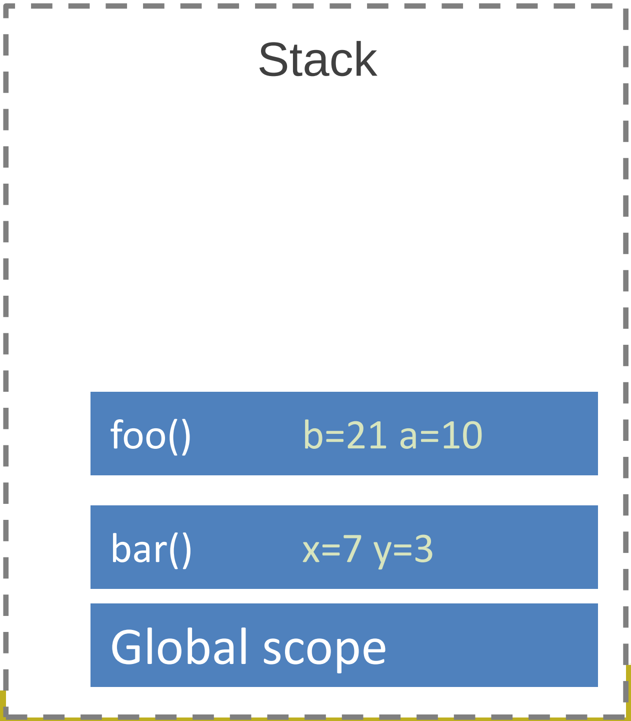
Execution model - Visual Representation



Function calls form a stack of *frames*

- When calling ***bar(7)***, a first frame is created in the stack containing bar's arguments and local variables.
- When calling ***foo()***, a second frame is created and pushed on top of the first one containing foo's arguments and local variables.
- When foo returns, the top frame element is popped out of the stack.
- When the global scope is done, the stack is empty.

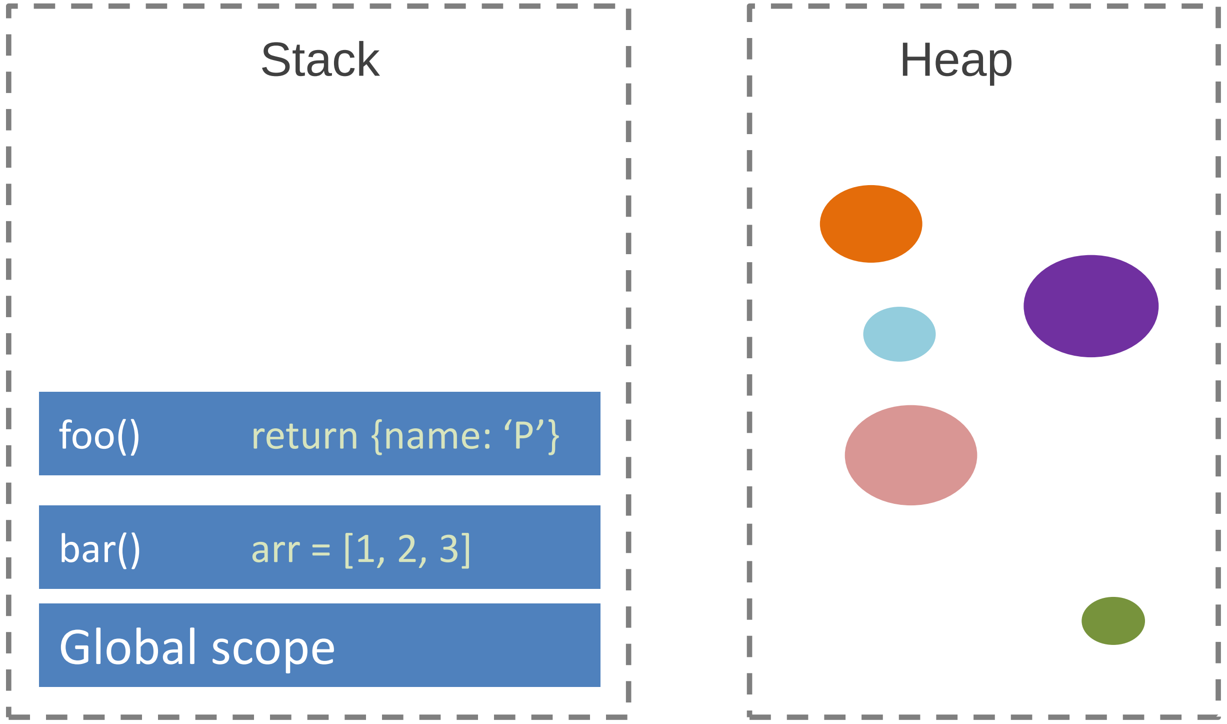
```
function foo(b) {
    var a = 10;
    return a + b + 11;
}

function bar(x) {
    var y = 3;
    return foo(x * y);
}

console.log(bar(7));
```



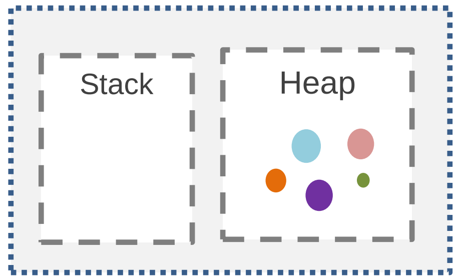
The Heap

Objects are allocated in the **heap**

which is large mostly unstructured region of memory.



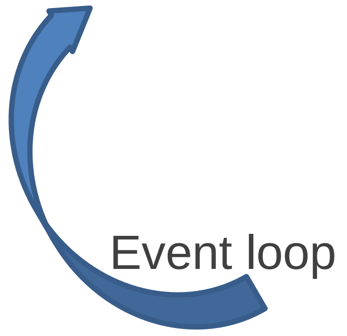
Messages Queue

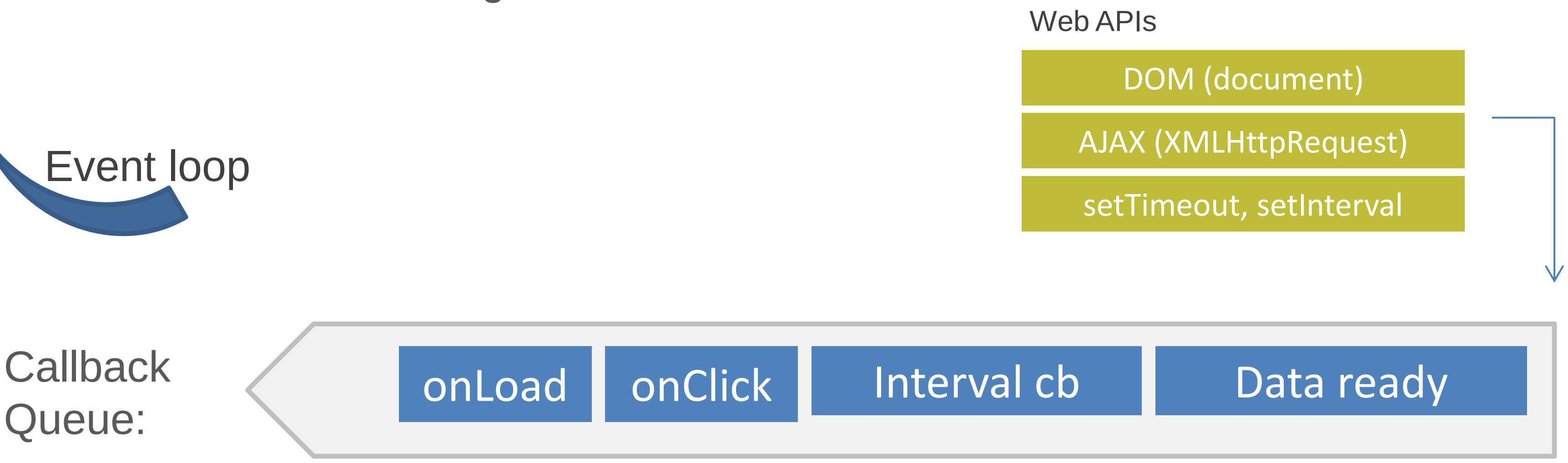


The JS runtime contains a message queue, which is a list of messages to be processed.
A function is associated with each message.

When the stack is (kind of) empty, a message is taken out of the queue and the function is processed.

The processing ends when the stack becomes empty again.





Run to completion & Never blocking

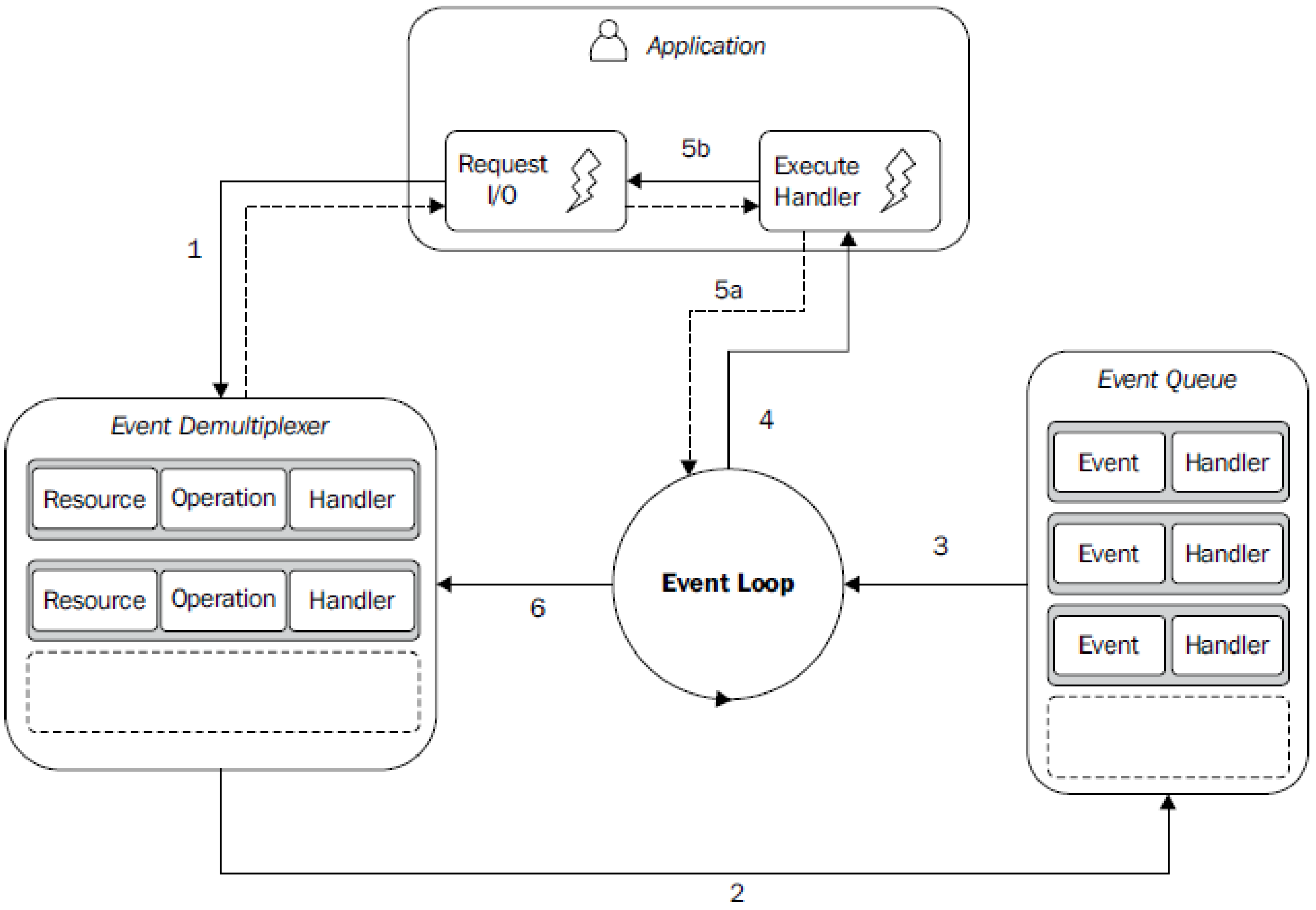
Run to completion
Each message is processed completely before any other message is processed.

- This is the single threaded nature of javascript
- This means our processing should be kept short and sweet

Never Blocking
Javascript *never* blocks!

Besides:
1. the native popups
2. There is a synchronous option (hardly used in the browser) for I/O such as in XMLHttpRequest

Execution model



Promises

- When working with async data, we had to work with callbacks
- Not any more!
- Its much cleaner and less error-prone

Promises to the rescue

Comparing between a simple callback and a simple promise does not show much difference

```
// Instead of sending a call back
getVyCallback(“someting", function(x) {
   console.log(“Response: " + x);
});

// using a promise
var promise = getByPromise(“something");
promise.then(function(x) {
   console.log(“Response: " + x);
});
```

Promises are compassable

- The callback passed to *then* can return a value, which will construct a new promise:

```
var prmData = fetch('http://www.filltext.com/?rows=10&city={city}&population={numberRange|1000,7000}')
    .then(function(res) { return res.json() })

prmData.then(function(data) { return data.length })
    .then(function(data){ console.log(data) })
```

Promises are powerful

- Promise give us turbo powers!
  - Cleaning up the *banana code*
  - Providing a single error callback for the entire process
  - Joining other synching asynch actions

```
var p1 = Promise.resolve(4);
var p2 = 1337;
var p3 = new Promise(function(resolve, reject) {
        setTimeout(resolve, 100, ‘puki');
});

Promise.all([p1, p2, p3]).then(values => {
    console.log(values); // [4, 1337, “puki"]
});
```

Dos & Don'ts

Avoid using with statements.

- Avoid using the *with* statement its evil.

```
function status(info) {
   var widget = new Widget();
   with (widget) {
      setBackground("blue");
      setForeground("white");
      setText("Status: " + info); // ambiguous reference
      show();
   }
}
```

dont allow *eval* to interfere with scope

```
var y = "global";
function test(x) {
   if (x) {
      eval("var y = 'local';");
   }
   return y;
}
test(true); // "local"
test(false); // "global"
```

- Those are all bad ideas…

Automatic semicolon insertion

- Javascript has the ability to leave off some statements' terminating semicolons.
- There are specific rules
  (i.e. - Semicolons are only ever inferred before a }, at the end of a line, or at the end of a program.)
- When leaning towards functional programming the semiolon is used even less.

Caching DOM elements

- In some cases, its important to keep references to DOM elements
- i.e. – instead of repeatedly finding the element in a loop

- Avoid Out of DOM references!
  - It is dangerous to hold DOM elements for longer than needed
  - if element is removed from the DOM, but you are still holding a reference to it, it will prevent its garbage collection and will cause memory leaks.

  - reference to a table cell <td> from JavaScript code will cause the whole table to stay in memory!

Consider this carefully when keeping references to DOM elements!

Memory Leaks

The main reasons for memory leaks in the frontend are:

- global variables (also accidental ones)
- setInterval
- Out of DOM references

Document Fragments

Build the entire HTML fragment before appending to the DOM

```
var el  = document.getElementById('ul'); // assuming ul exists
var fragment = document.createDocumentFragment();
var browsers = ['Firefox', 'Chrome', 'Opera',
    'Safari', 'Internet Explorer'];

browsers.forEach(function(browser) {
    var li = document.createElement('li');
    li.textContent = browser;
    fragment.appendChild(li);
});

el.appendChild(fragment);
```

One time function

- Here is a function with self-destroy mechanism (can be used only once)

```
function oneTimer() {
   // Do your thing
   // self destroy
   oneTimer = function(){};
}
```

Arguments.caller and Arguments.callee

- Even though it works to some extent, it is recommended to avoid the nonstandard arguments.caller and arguments.callee, because they are not reliably portable.

```
function revealCaller() {
   return revealCaller.caller;
}
function start() {
   return revealCaller();
}
start() === start; // true
```

Function trace

- You can also figure out information about the function and stack trace:

```
function foo(x, y) {
   console.log(foo.length); // print '2‘ – arguments count
   console.log(foo.caller); // prints 'start(shouldCallFoo)'
   foo.caller(false);       // prints 'Called by foo'
}
function start(shouldCallFoo) {
   if (shouldCallFoo) foo(8, 3);
   else console.log("Called by foo");
}
start(true);
```

Handling Errors

- Asynchronous APIs cannot throw exceptions
  - Instead, asynchronous APIs tend to represent errors as special arguments to callbacks, or take additional error-handling callbacks
  - in Node.js it is popular to have the first argument hold the error if occurred

```
downloadAsync("http://example.com/file.txt", function(text) {
   console.log("File contents: " + text);
}, function(error) {
   console.log("Error: " + error);
});

// OR in node.js style:
downloadAsync("a.txt", function(error, data) {
   if (error) {
      console.log("Error: " + error);
      return;
   }
   console.log("Data:", data);
}
```

Don’t Block the Event Queue on Computation

- Asynchronous APIs help to prevent a program from clogging up an application’s event queue
- But a long computation such as: *while (true) { }*
  is enough to stall an application

- One technique to handle CPU intensive operations is the Worker API:

The worker API

- The Worker API makes it possible to spawn concurrent computations.
- Unlike conventional threads, workers are executed in a completely isolated state, with no access to the global scope or the DOM
  - i.e. In a worker, you may use the synchronous variant of XMLHttpRequest as it does not prevent the page from rendering or the event queue from responding to events

```
// Main Script
var worker = new Worker('doWork.js');
worker.addEventListener('message', function(e) {
   console.log('Worker said: ', e.data);
}, false);

worker.postMessage('Hello Worker');

// doWork.js (the worker):
self.addEventListener('message', function(e) {
   self.postMessage(e.data);
}, false);
```

Transferables

- Objects that can be transferred between different execution contexts, like the main thread and Web workers.
- The ArrayBuffer, MessagePort and ImageBitmap types implement this interface.
- Read more:
- https://developer.mozilla.org/en-US/docs/Web/API/Transferable

Break work to smaller chunks

- Another technique to avoid blocking the Event Queue on Computation is breaking the work into smaller chunks
- Then using *setTimeout* to give the event loop a chance to handle other events that arrived
  - Is that a recursion?

```
function doit(cb) {
   var result = {};
   var isDone = false;
   function doSomeWork() {
      // ... handle 10 items and update result ....
      isDone = checkIsDone();
      if (!isDone) setTimeout(doSomeWork, 0);
      else return cb(result) )
   }

   setTimeout(doSomeWork, 0); // schedule the first iteration
};
```

Avoid calling an asynchronous callback synchronously

- Calling an asynchronous callback synchronously disrupts the expected sequence of operations and can lead to unexpected interleaving of code or mishandled exceptions.
- So excuting the callback in async way even if the data is immediately available.

- This is a common way to handle it:

  setTimeout(onDataReady.bind(null, cachedData), 0);

Here is an example of why this important

Avoid calling an asynchronous callback synchronously

```
function readData(onDataReady) {
    var cachedData = getFromCache();
    if (cachedData) {
        //onDataReady(cachedData);
        setTimeout(onDataReady.bind(null, cachedData), 0);

    } else {
        $.getJSON('someUrl', onDataReady)
    }
}

readData(function (data) {
    console.log(data);
})
```

Avoid data race

- Remember that events occur in unpredictable order:
  - If I issue 2 calls for the server, they will return on unpredictable order
  - Based on (Server cost, Network cost, etc)
- Promises are a great help here.

Compatibility

Handle cross browser issues

- The Least-Common-Denominator Approach – you only use features supported everywhere – this is not a great idea if you want to be a leader of your market.
- Defensive Coding – write and include code (Shiv) to fix the behavior in specific browsers
- Feature testing:

```
newwin = window.open("", "new", "width=500, height=300");
if (!newwin.opener) newwin.opener = self;
```

- Implementing this strategy, Modernizr is a JavaScript library that detects HTML5 and CSS3 features in the user's browser.

Handle cross browser issues

- Platform-Specific Workarounds

```
var browserVersion = parseInt(navigator.appVersion);
var isNetscape = navigator.appName.indexOf("Netscape") != -1;
var isIE = navigator.appName.indexOf("Microsoft") != -1;
var agent = navigator.userAgent.toLowerCase( );
var isWindows = agent.indexOf("win") != -1;
var isMac = agent.indexOf("mac") != -1;
var isUnix = agent.indexOf("X11") != -1;


if (isNetscape && browserVersion < 4 && isUnix) {
    // Work around a bug in Netscape 3 on Unix platforms here
}
```

Handle cross browser issues

- Compatibility Through Server-Side Scripts
  - Server side can inspect the User-Agent field of the HTTP request header, which allows it to determine exactly what browser the user is running.
  - With this information, the program can generate customized JavaScript code that is known to work correctly on that browser.
- Ignore the problem
  - Supporting older browsers is a hard work, if you can avoid it and refer user to upgrade their software you will probably live longer and happier life.
- Fail gracefully
  - Don’t crash, expect the unexpected and may the force be with you.
  - try-catch sometimes help here

Real project example

- Here is a snippet from a grunt file used for build a player-studio project
  - see how much extra libraries were needed for old browsers support.

```
vendorModern: {
                src: [
                    'app/bower_components/jquery/jquery2.0.3.min.js',
                ],
                dest: 'dist/vendor/vendor.min.js'
            },
vendorOld: {
                src: [
                    'app/lib/html5shiv.js',
                    'app/lib/respond.min.js',
                    'app/lib/es5-shim.min.js',
                    'bower_components/selectivizr/selectivizr.js',
                    'app/lib/jquery-1.10.2.min.js',
                    'app/lib/jquery.xdomainrequest.min.js',
                ],
                dest: 'dist/vendor/vendorOld.min.js'
            }
```

Which browser to use?

- Don’t use IE as your development environment, as it is usually the one behaving differently.
- Chrome is a better choice, but sometimes it is “too advanced” and may again understand things that the rest of the browsers wont.
- Chrome and Firefox are leading choices due to their large list of extensions for developers

IE is a *different* boy..

- Here are some examples:

Action	IE	Others
XHR	Older versions via ActiveX	XMLHttpRequest
Set or read the text value of an element	innerText	textContent
Read/Set the float property of an element	style.cssFloat	style.styleFloat
Get the event object in a function	window.event	must be passed to the function
Get the target of an event	event.srcElement	event.target
Get window size	document.body.clientHeight or document.documentElement .clientHeight (depending on the doctype)	window.innerHeight and window.innerWidth

IE things

- IE detection comments
-
```
<!--[if IE]>
   <link rel="stylesheet" type="text/css" href=“fixIE.css"/>
<![endif]-->
```

Designing your APIs

Stateful vs Stateless APIs

- stateless APIs tend to be easier to learn and use, more self-documenting, and less error-prone.
  - A sample stateful API is the HTML5 Canvas
- Prefer stateless APIs where possible.

Accept Options Objects instead of many config Arguments

```
// Instead of many parmas:
var alert = new Alert("Error", message,
"blue", "white", "black",
"error", true);

// use opts object instead
function Alert(msg, opts) {
   // extend from some 3rd party
   opts = extend({
      title: "Alert",
      titleColor: "gray",
      bgColor: "white",
      textColor: "black",
      icon: "info",
      modal: false
   }, opts);
   extend(this, opts);
}
```

Note: The arguments provided by an options object should all be treated as optional.

Support Method Chaining

- It helps building compound operations out of smaller ones:

```
function escapeBasicHTML(str) {
   return   str.replace(/&/g, "&amp;")
                .replace(/</g, "&lt;")
                .replace(/>/g, "&gt;")
                .replace(/"/g, "&quot;")
                .replace(/'/g, "&apos;");
}
```

```
var userNames = users.map(function(user) {
   return user.username;
})
.filter(function(username) {
   return !!username;
})
.map(function(username) {
   return username.toLowerCase();
});
```

Method Chaining for stateful APIs

- Method chaining for stateful APIs is sometimes known as the fluent style.
- JQuery API is a good example:

```
$(".baloon").html("Try again")
            .removeClass("info")
            .addClass("error");
```

Functional Programming

Unless you're building a reality like game or simulator (where OOP is a good design pattern)

When building crud apps, prefer using the functional style: (more here)

```
var transactions = "P 130.56, C, P 12.37 , P 6.00, R 75.53, P 1.32";
transactions
   // Break the string into an array on commas
   .split(",")

   // Keep just the purchase transactions ('P')
   .filter(function(s) { return s.trim()[0] === 'P' })

   // Get the price associated with the purchase
   .map   (function(s) { return +(s.trim().substring(1).trim()) })

   // Sum up the quantities of purchases
   .reduce(function(acc, v) { return acc + v; });
```

MisterBit
misterBIT.co.il
Keep on diving