

An Introduction to JavaScript

IF3110 – Web-based Application Development
School of Electrical Engineering and Informatics
Institut Teknologi Bandung

Introduction

- Interpreter based, used to be only running on the browser – run on server-side (node.js) or in a JVM (rhino)
- Javascript has similar syntax to Java, but it is **not** based on it
- Dynamic typing and supports duck typing
- Objects as general containers
- Function is the first-class (Lambda/closure)
- Linkage via global variables
- Prototypes over class-inheritance (class keyword is syntactic sugar to prototypes)
- Formalized & standardised as **ECMAScript** (**ECMA-262** and ISO/IEC 16262)

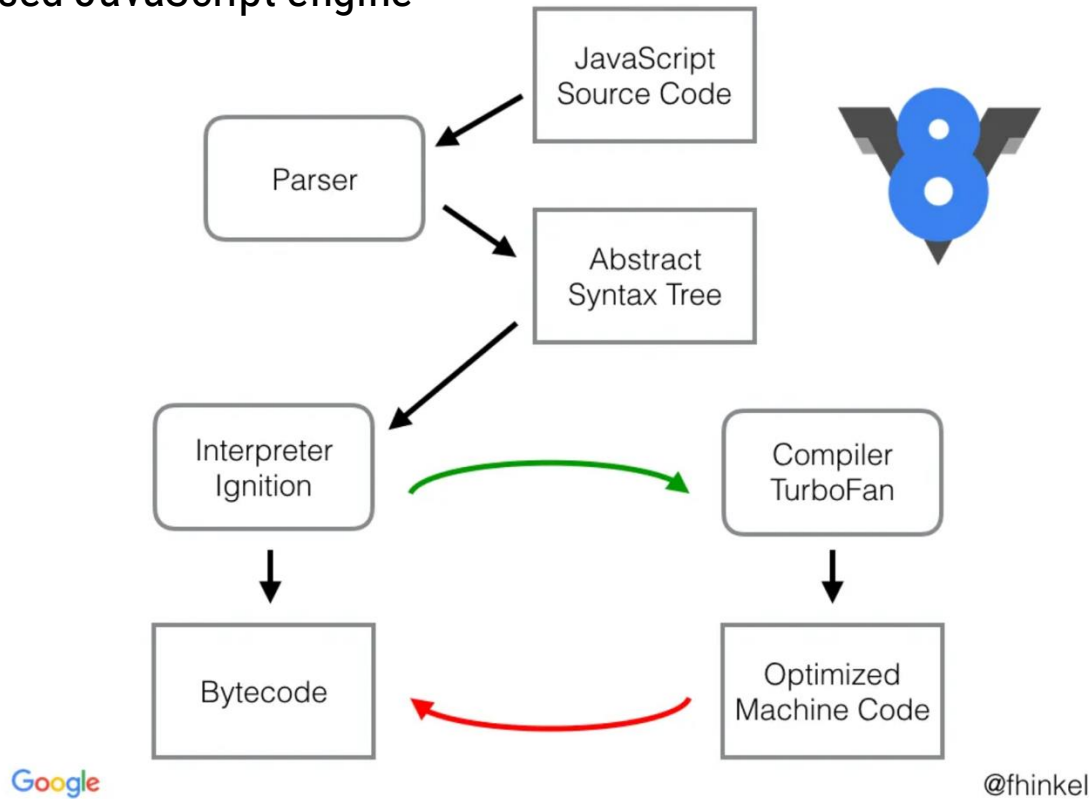
Introduction

- Javascript runs in browser:
 - Core syntax + DOM
- Has various implementation variant (the JavaScript Engine)
 - JavaScript Engine -> a special program embeded in a browser to execute JavaScript code. This program implements ECMAScript standard.
 - V8 -> Chrome, Edge, Opera, **Node.js (not a browser)**
 - SpiderMonkey -> Mozilla Firefox
 - JavaScriptCore -> WebKit/Safari
- support provided for DOM
 - The DOM is not, however, typically provided by the JavaScript engine but instead by a browser

Inside the JavaScript Engine

V8 from Google is the most used JavaScript engine

written in C++



Courtesy: Franziska Hinkelmann's talk on V8



<https://www.youtube.com/watch?v=p-iiEDtpy6I>

V8 has had an interpreter as its first execution tier. This interpreter "compiles" JavaScript "just in time/JIT" to bytecode (which is then interpreted).

JavaScript in HTML

- Locating JavaScript

- internal HTML document

- Standar HTML 4.01

- ```
<script type="text/javascript">
```

- ```
...statement...
```

- ```
</script>
```

- Old Tag

- ```
<script language="JavaScript">
```

- ```
...statement...
```

- ```
</script>
```

- file external

- Script JavaScript dituliskan pada file tersendiri (ekstensi file .js)

- Pemanggilan file JavaScript:

- ```
<script src="namafile.js"></script>
```

- Tips: place js script at the end of a html, to improve the performance

# Simple Example

```
<HTML>
<HEAD>
<TITLE>Hello javascript</TITLE>
</HEAD>
<BODY onload="createDoc();" >
<H1>Hello javascript...</H1>
<HR>
<SCRIPT type="text/javascript">
 function createDoc() {
 document.write("Hello...")
 }
</SCRIPT>
</BODY>
</HTML>
```

# Basic Syntax

- Alike to Java or C
- Case-sensitive
- One line represents a statement
- “;” is optional to end a statment
- Comment: // and /\* \*/

# Basic Syntax: Type

- Data Type
  - Number
  - String
  - Boolean
  - Object: e.g.:
    - Function
    - Array
    - Date
    - RegExp
  - Null
  - undefined



# Number

All numbers are represented in floating-point 64-bit IEEE-754 (double), but can be operated in integer

`0, 3, 100000000, 0xF3, 031`

`3.14, 1.23e10, 3.14E-14`

Be careful with **rounding errors**

Special values

`NaN, Infinity`

`Number.MAX_VALUE, Number.MIN_VALUE`

`Number.POSITIVE_INFINITY`

`Number.NEGATIVE_INFINITY`

# Number

NaN: Not a Number, result from a wrong operation

Every operation that has NaN, will result in NaN

NaN != NaN

```
>>> var a = 0/0;
```

```
>>> a
```

```
NaN
```

```
>>> typeof a
```

```
"number"
```

# Number function

```
Number (value)
```

Cast some value into number

Result in NaN if there is some problem,

Alternative: + prefix & parseInt

```
+value
```

```
parseInt (value, radix)
```

# Math

Math object provides a collection of standard function in arithmetic

<code>abs</code>		absolute value
<code>floor</code>	integer	
<code>log</code>		logarithm
<code>max</code>		maximum
<code>pow</code>		raise to a power
<code>random</code>	random number	
<code>round</code>	nearest integer	
<code>sin</code>		sine
<code>sqrt</code>	square root	

# String

- Series of character written with some delimiter “ or ‘

‘this is string’, “this also a string”, “ini bisa ‘kan”, “nama=‘amir’”, ‘This string\n consists of 2 lines’

- Immutable
- Some basic operations:

```
s = "hello"
```

```
s.length, s.charAt(2), sub = s.substring(2, 3)
```

```
i = s.indexOf('e')
```

```
"hello, world".replace("hello", "goodbye")
```

```
"hello".toUpperCase()
```

# String Methods

`charAt`

`concat`

`indexOf`

`lastIndexOf`

`match`

`replace`

`search`

`slice`

`split`

`substring`

`toLowerCase`

`toUpperCase`

# boolean

- Value: true atau false
- Boolean(value): cast some value into boolean. Alternative: use !!
- Falsy: 0, **false**, **null**, **undefined**, "" (empty string), NaN
- Truthy: other than falsy (including "null", "0", "false")

# 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

# Loosely and Dynamically Typed

## **Loosely typed**

Any of these types can be stored in a variable, or passed as a parameter to any function.

The language is not "untyped".

Conversion of types happen automatically.

## **Dynamically typed**

Types are not required in advance.

Types are not checked at compile-time.

# Type Conversion

`10 + " objects" // => "10 objects"`. Number 10 converts to a string

`"7" * "4" // => 28`: both strings convert to numbers

`var n = 1 - "x"; // => NaN`: string "x" can't convert to a number

`n + " objects" // => "NaN objects"`: NaN converts to string "NaN"

# Variable

- Case sensitive
- Declare by var, const, or let (ES6, 2015)

`var i; var x = 2;`

`let y = 3; // having a block scope, and cannot be redeclared`

`const z = 4; // having a block scope, and cannot be redeclared & reassigned`

- Variable with out a declaration, will be declared automatically as a global variable

`a = 5;`

- Scope: global dan local. local used when variable declared with in a function with var
- Convention: variable name, parameter, member, function starts with lowercase. Constructor starts with uppercase

# Operator Aritmatika

Operator	Usage	Example	Result
+	addition	3+4	7
-	subtraction	4-3	1
*	multiplication	4*3	12
/	division	4/3	1.33333333
%	modulus	4 % 3	1
++	increment	x=5 x++	x=6
--	decrement	x=5 x--	x=4

# Assignment Operator

operator	example	explanation
=	a = b	assignment b to a
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

# Comparative Operator

operator	example	explanation
==	a == b	a equal to b
!=	a != b	a not equal to b
<	a < b	a lesser than b
>	a > b	a bigger than b
<=	a <= b	a lesser or equal to b
>=	a >= b	a bigger or equal to b
===	'10' === 10	comparation w/o automatic type conversion

# Logic Operator

operator	usage	example
&&	and	x = 6 y = 3 ( x < 7 && y < 4)
	or	x = 6 y = 3 ( x < 7    y < 2)
!	not	x = 6 y = 3 x != y





Equal and not equal operators

These operators can do type correction

It is better to use `===` (strict equality operator) and `!==`, which do not do type correction.

**Examples:**

```
"5" == 5 // true, because of type coercion
```

```
"5" === 5 // false, no type coercion occurs
```

# &&

- 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`.)



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.

# string operator

“ini “+”buku”

“ini buku”

“jumlah adalah “+ 5

“jumlah adalah 5”

“jumlah adalah “ + 5 + 3

“jumlah adalah 53”

5 + 3 + “ adalah bilangan”

“8 adalah bilangan”

# 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 (var 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 statement

Iterate through all of the members of an object:

```
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

 }

}
```

# Switch statement

Multiway branch

The switch value does not need to a number. It can be a string.

The case values can be expressions.

# 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'`

# With statement

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

```
with (o) {

 foo = null;

}

❑ o.foo = null;

❑ foo = null;
```

# Function

- mechanism to structure a program
  - modular
  - reusable
- kind
  - built-in
  - user defined
- function can be
  - have parameters
  - return a value



# Function

## syntax

```
function namaFungsi ([parameter]) {
 ... statements
 [return value]
}
```

function can be defined within another function

# Many ways to define a function

- typical definition

```
function f(x, y) { return x*y; }
```

- constructor Function()

```
var f = new Function("x", "y", "return x*y;");
```

- function literal

```
var f = function(x, y) { return x*y; }
```

- arrow function

```
var f = (x, y) => return x*y;
```

# function as data

function can be treated as data, stored in variable

```
function square(x) { return x*x; }
```

```
var b = square;
```

```
var c = b(5);
```

# function: parameter

function can have parameter/argument

```
function square(x) { return x*x; }
```

argument can be access from the function w/ name or object arguments

```
function square(x) { return arguments[0]*x; }
```

argument checking/verification in Javascript done at runtime

# Return statement

`return expression ;`

or

`return ;`

If there is no *expression*, then the return value is **undefined**.

Except for constructors, whose default return value is **this**.

# Object

Object – entity that composed of states represented as attributes (properties) value, and behavior represented in term of methods



# JavaScript Object

- In JavaScript, almost everything is object
  - Unless the 6 primitive types (string, number, boolean, null, undefined, symbol in ES6, bigint in ES11)
- Object in Javascript is a collection
  - similar to hashtable – can be accessed by a key (member name)
- Member is accessible by dot or subscript ( a.b or a['b'])
- `new Object()` produces an empty container of name/value pairs
- A name can be any string, a value can be any value except **undefined**
- A function is also an object

# Object

- Java Script is prototype-based object language

- Object created by constructor

```
var now = new Date()
```

- Object can be created from literal

```
var circle = { x:0, y:0,
radius:2 }
```

- `class` keyword is introduced in ES6 (2015)

- Attributed & methods access by . (dot)

```
var book = new Object();
```

```
book.title = "Javascript: The
Definitive Guide"
```

```
book.author = "David
Flanagan"
```



# Object

- property (attribute) can be enumerated w/ for loop

```
for(var name in obj) {

 document.write(name + "
");

}
```

- property of an object can be removed

```
delete book.title;
```

# property access

- using dot

```
object.property
```

- as associative array

```
object["property"]
```

# Constructor

- special function to create/initiate an object
- called with “new” command

```
var now = new Date()
```

# prototype

- prototype: mechanism to share properties and methods of objects Javascript
- every object has prototype
- properties & methods, to be share with other objects, place in prototype.

```
Circle.PI = 3.14;
```

```
Circle_area() { return Circle.PI * this.r * this.r; }
```

```
Circle.prototype.area = Circle_area;
```

```
c = new Circle();
```

```
document.write(c.area());
```

# prototype

- properties & method, in object instance, allocated ONLY for that particular instance  
  
'this' refers to the instance
- properties & method, in constructor, allocated ONLY for that constructor  
  
can be used from other object  
  
'this' refers to constructor
- properties & method, in prototype, can be used by other object  
  
'this' refers to the instance

# Circle class prototype

```
function Circle(radius) {
 this.r = radius;
 this.min = function() { }
}

Circle.PI = 3.14159;

Circle.prototype.area = function() {
 return Circle.PI * this.r * this.r;
}

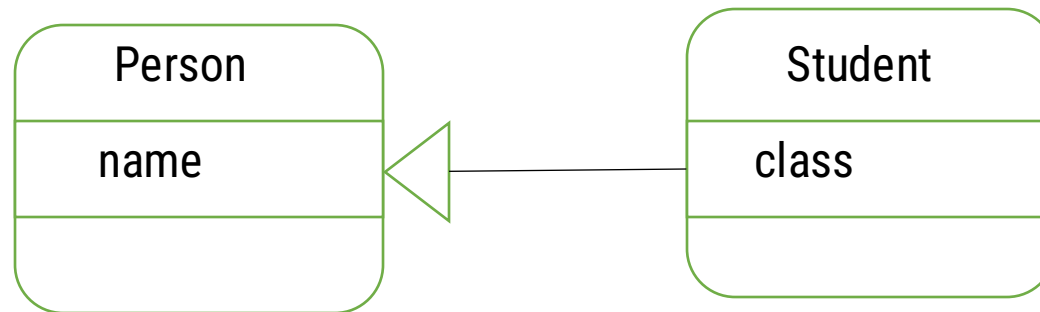
Circle.max = function(a,b) {
 if (a.r > b.r) return a;
 else return b;
}
```

```
var c = new Circle(1.0);
var d = Circle(2.0}
Circle.prototype.name = "a";
c.r = 2.2;
var a = c.area();
var x = Math.exp(Circle.PI);
var d = new Circle(1.2);
var bigger = Circle.max(c,d);
```

# Inheritance

Inheritance: OO concept for reusability

using an existing class to define a new class



# Inheritance through prototype

JavaScript provides inheritance through prototype:

```
function Person(n) { this.name = n; }
function Student(n, c) {
 this.name = n;
 this.class = c;
}
Student.prototype = new Person();
```



# Class and Inheritance

Since ES6 (2015) :

```
class Person {
 constructor(n) {
 this.name = n;
 }
}
class Student extends Person {
 constructor(n, c) {
 super(n);
 this.class = c;
 }
}
```

# Object

## constructor

o.constructor consists of function constructor used to initialize o

## toString()

represents string of an object. Called automatic when an object is casted to string

## valueOf()

represents object other than string. Called automatic when an object is casted to other than string

## isPrototypeOf( x )

verify whether an object is prototype from another object

# Array

- **by Array constructor**

```
var a = new Array();
```

```
var a = new Array(1, 2, "tiga");
```

- **by literal**

```
var a = [1, 2, "tiga"];
```

- **array access**

```
a[0] = 1;
```

```
a[7] = 8;
```

- **length of array**

```
a.length
```

# Array methods

`join()`

convert array values into string

`reverse()`

revers the value of array

`sort()`

sort the array values alphabetically

`concat()`

combing array values with the value of parameter

`slice()`

slice the value of array array

`splice()`

remove and add the value of array

`push()`, `pop()`, `shift()`, `unshift()`

# Array Example

```
<html>
<script type="text/javascript">
//cara 1 pendefinisian array
mhs = new Array();
mhs[0] = "Bevin";
mhs[1] = "Andini";
mhs[2] = "Citra";
//cara 2 pendefinisian array
mhs = new Array("Bevin", "Andini", "Citra");
//cara 3 pendefinisian array
mhs = ["Bevin", "Andini", "Citra"];
//cara pengaksesan array
document.write("Mahasiswa pertama adalah "+mhs[0]+".
"); //Bevin
document.write("Mahasiswa terakhir adalah "+mhs[2]+".
"); //Citra
//cara pengaksesan array menggunakan loop
for (i=0; i<mhs.length; i++) {
 document.write(mhs[i] + "
");
}
//mengurutkan array
mhs.sort();
//menggabungkan array
document.write(mhs.join("-")); //Andini-Bevin-Citra
</script>
</html>
```

# Regular expression

- used to process text, search and convert text with particular pattern

- Literal:

```
var pattern = /s$/
```

- RegExp object

```
var pattern = new RegExp ("s$") ;
```

# Browser object

window

navigator

screen

history

location

Not standard, but provided in every browser

# Browser object

Window object: global execution context,

```
var x = 5;
```

sama dengan

```
window.x = 5;
```

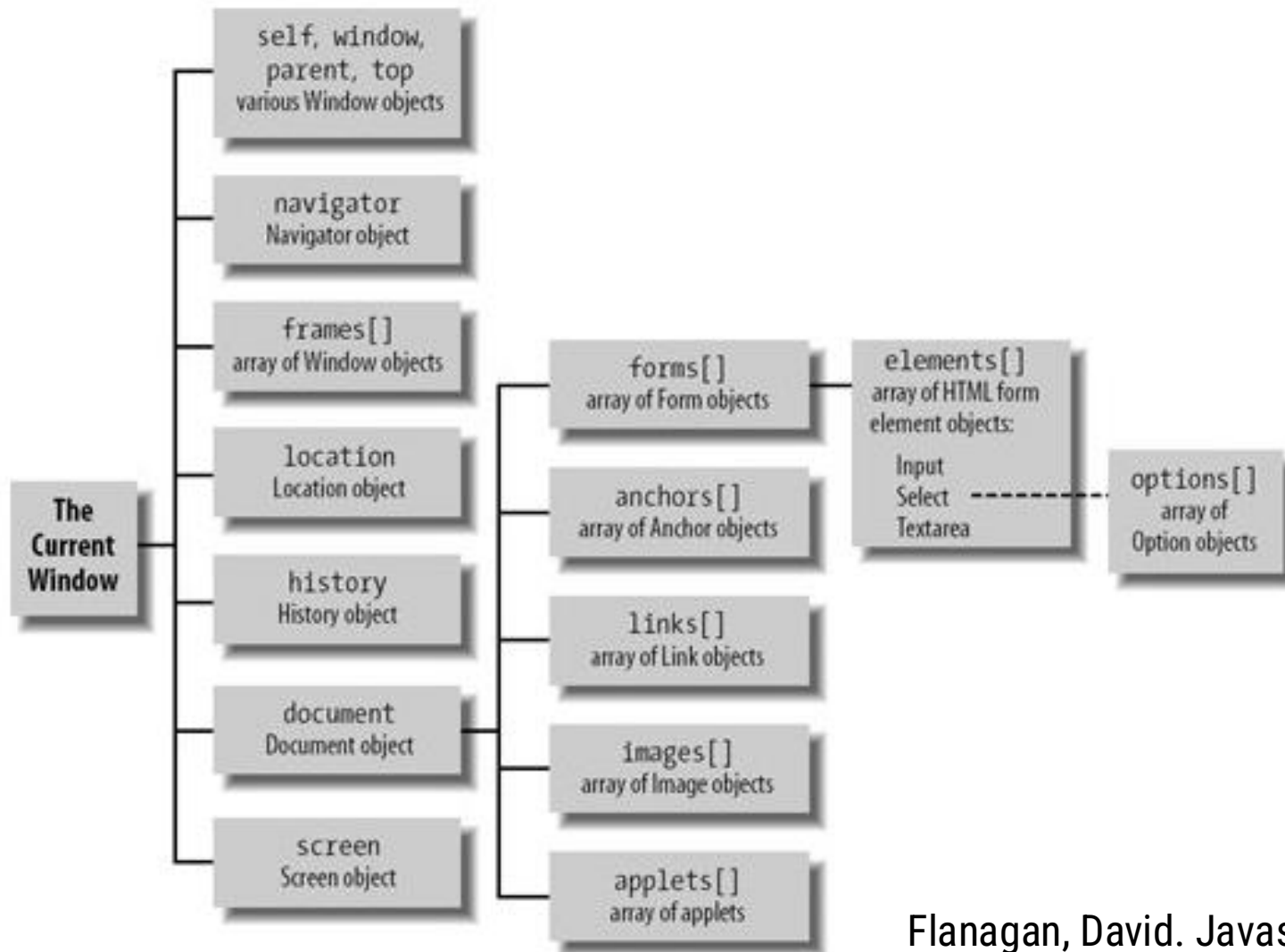
It refers to window browser (or frame) that renders the page

Document object: page that is being rendered

```
window.document
```



# DOM



Flanagan, David. Javascript: The Definitive Guide, 5th Ed.

# Document object

`getElementById( id )`

`getElementsByTagName( tag )`

`createElement( tag )`

HTMLDocument, derived from Document

body, forms, anchors, images, links

title, lastModified, referrers

etc

`close()`, `open()`, `write()`

# Javascript as event handler

HTML Element HTML can define to run script if an event occurs

onclick, onmousedown, onmouseup, onchange, onload

In-line HTML:

```
<input type="checkbox" name="options" value="giftwrap"
onclick="giftwrap = this.checked;" >
```

# Javascript as event handler

Script:

```
<script>
// Define a function to display function displayTime() {
// A script of js code the current time
var elt = document.getElementById("clock");
// Find element with id="clock"
}
window.onload = displayTime;
// Start displaying the time when document loads.
</script>
```

# Javascript as URL and Bookmarklet

Javascript can be executed as a URL link

```
javascript:alert("Hello World!")
```

```
javascript:var now = new Date(); "<h1>The time is:</h1>" + now;
```

URL containing javascript can be saved as a bookmark

```
<a href='javascript:var e="", r="";
do{e=prompt("Expression: "+e+"\n"+r+"\n",e);
try{r="Result: "+eval(e); }catch(ex){r=ex; }}
while(e);void 0;'> JS Evaluator
```

JavaScript URL often used by hackers to run malicious codes, e.g., Cross Site Scripting (XSS)

# Javascript Execution in HTML

Javascript is executed as soon as the element, containing script, has been processed by the browser.

To postponed the execution: use `defer` attributed,

OR

write a script as a function that is called by `onload` attributes from the body element

# Javascript threading model

Javascript runs sequentially; with a single thread model.

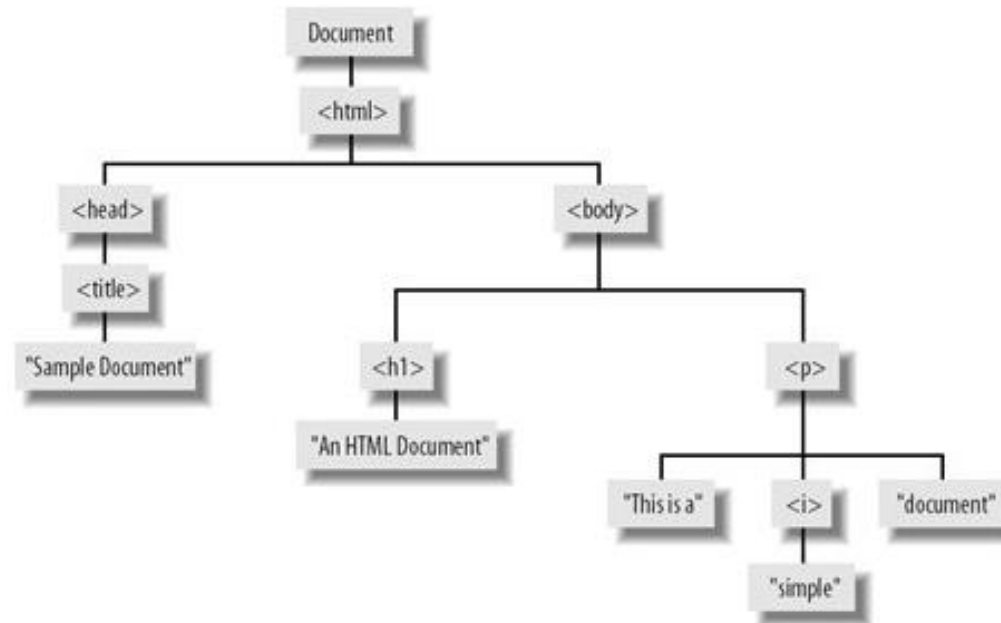
On loading HTML document HTML, executing a script will suspend the loading process until the execution finishes

event-handler javascript SHOULD NOT be long, it creates a browser freeze.

# W3C DOM

HTML Document is accessed by javascript via DOM

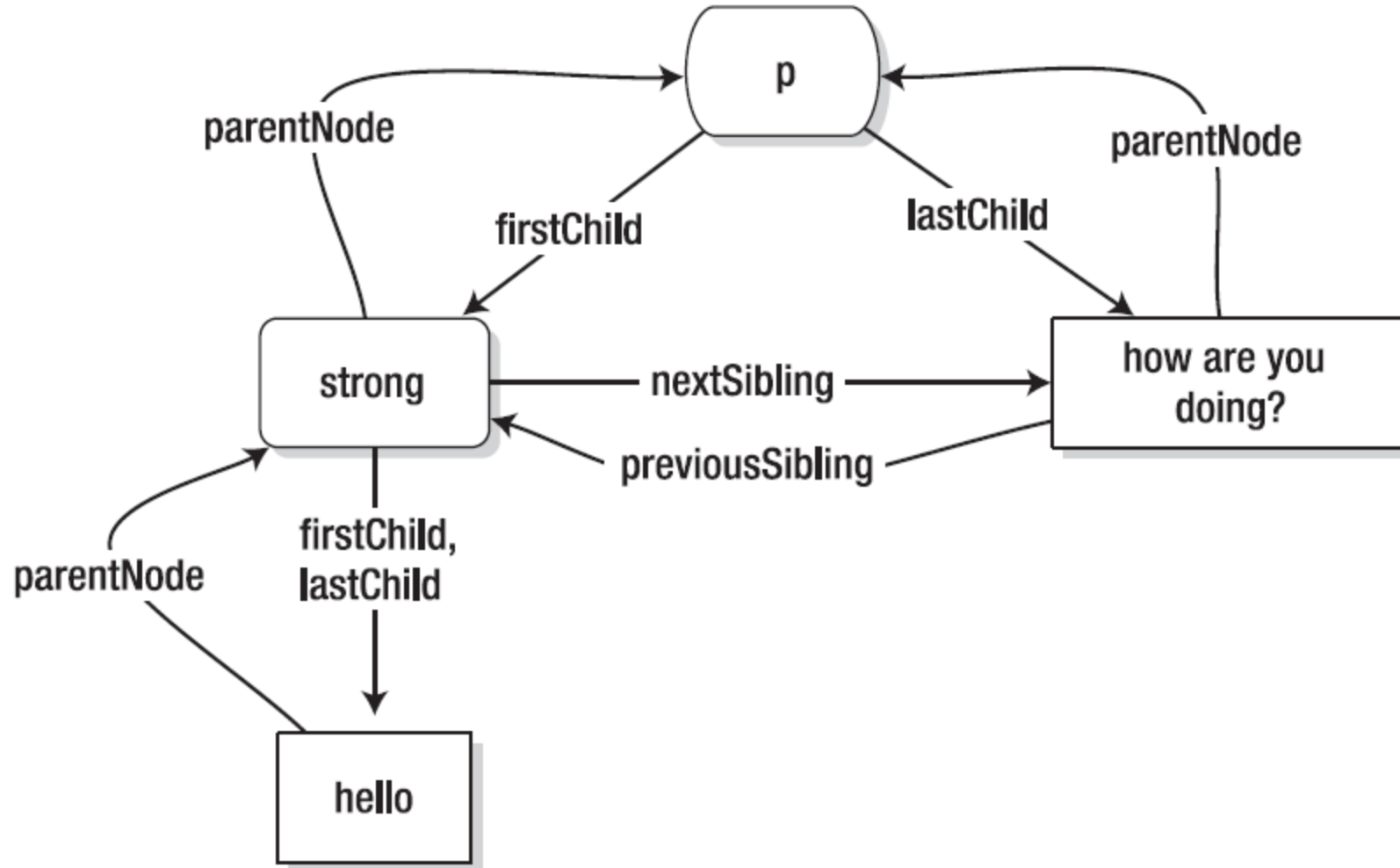
```
<html> <head> <title>Sample Document</title> </head> <body> <h1>An
HTML Document</h1> <p>This is a <i>simple</i> document.</p> </html>
```





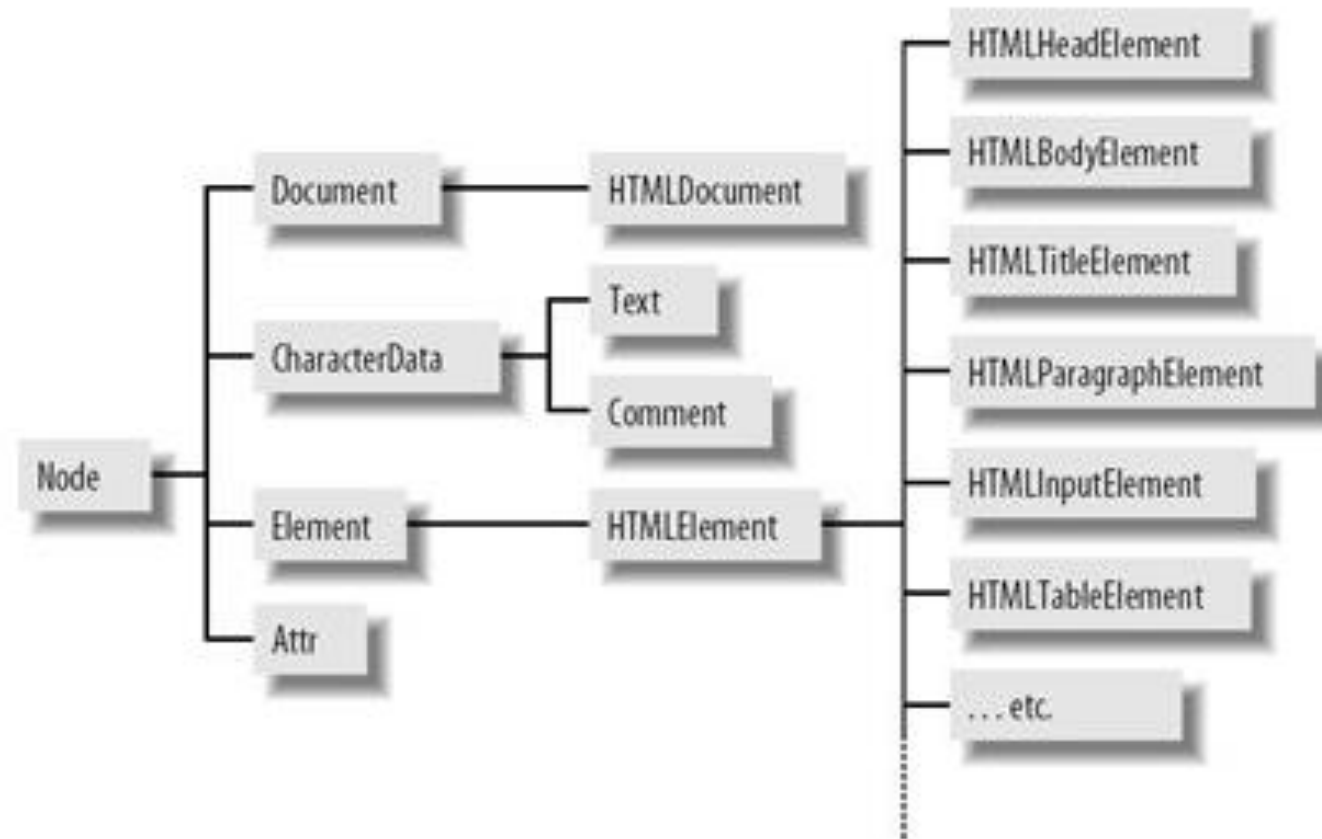
# W3C DOM

`<p><strong>Hello</strong> how are you doing?</p>`



# W3C DOM

HTML element in DOM is descendant of Node



# W3C DOM

DOM Level: standardisation DOM by W3C in 2 version (level)

DOM Level 1, 1998

Definition of Node, Element, Attr, Document etc

DOM Level 2, 2000

Modular. DOM Level 1 becomes Core module, there are optional modules, such as Event, CSS

DOM Level 3

# W3C DOM

Accessing element:

```
document.getElementById(id_name);
```

return object Node object, if existed

```
Document.getElementsByTagName(tag_name);
```

return NodeList, can be accessed as array

```
document.getElementsByTagName("h1")[0]
var li = document.getElementsByTagName("li");
for (var j = 0; j < li.length; j++) {
 li[j].style.border = "1px solid #000";
}
```

# Encode/Decode URIs

- In a URL (Uniform Resource Locator) or a URI (Uniform Resource Identifier), some characters have special meanings.
  - If you want to "escape" those characters, you can use the functions `encodeURIComponent()` or `encodeURIComponent()`.
    - The first one will return a usable URL.
    - The second one assumes you're only passing a part of the URL, like a query string for example, and will encode all applicable characters.
- ```
>>> var url = 'http://www.packtpub.com/script.php?q=this and that';
>>> encodeURI(url);
"http://www.packtpub.com/script.php?q=this%20and%20that"

>>> encodeURIComponent(url);
"http%3A%2F%2Fwww.packtpub.com%2Fscript.php%3Fq%3Dthis%20and%20that"
```
- The opposites of `encodeURIComponent()` and `encodeURIComponent()` are `decodeURI()` and `decodeURIComponent()` respectively.
 - Sometimes, in older code, you might see the similar functions `escape()` and `unescape()` but these functions have been deprecated and should not be used.

Timers

Timer berguna untuk membuat animasi atau efek tampilan

```
<script>
  var WastedTime = {
    start: new Date( ),
    displayElapsedTime: function( ) {
      var now = new Date( );
      var elapsed = Math.round((now - WastedTime.start)/60000);
      window.defaultStatus = "You have wasted " + elapsed + "
        minutes."; }
  }
  setInterval(WastedTime.displayElapsedTime, 60000);
</script>
```

Forms

- Form can be accessed via javascript using `document.forms[]`
- Form object has `submit()` and `reset()` method
- Javascript can be called from a form via event:
 - `onsubmit`, `onreset`
 - `onchange`, `onblur`, `onfocus`

Form Element

| Object | HTML tag | type property | Description and events |
|----------|---|-------------------|--|
| Button | <code><input type="button"></code> or <code><button type="button"></code> | "button" | A push button; <code>onclick</code> . |
| Checkbox | <code><input type="checkbox"></code> | "checkbox" | A toggle button without radio-button behavior; <code>onclick</code> . |
| File | <code><input type="file"></code> | "file" | An input field for entering the name of a file to upload to the web server; <code>onchange</code> . |
| Hidden | <code><input type="hidden"></code> | "hidden" | Data submitted with the form but not visible to the user; no event handlers. |
| Option | <code><option></code> | none | A single item within a Select object; event handlers are on the Select object, not on individual Option objects. |
| Password | <code><input type="password"></code> | "password" | An input field for password entrytyped characters are not visible; <code>onchange</code> . |
| Radio | <code><input type="radio"></code> | "radio" | A toggle button with radio-button behavioronly one selected at a time; <code>onclick</code> . |
| Reset | <code><input type="reset"></code> or <code><button type="reset"></code> | "reset" | A push button that resets a form; <code>onclick</code> . |
| Select | <code><select></code> | "select-one" | A list or drop-down menu from which one item may be selected; <code>onchange</code> . (See also Option object.) |
| Select | <code><select multiple></code> | "select-multiple" | A list from which multiple items may be selected; <code>onchange</code> . (See also Option object.) |
| Submit | <code><input type="submit"></code> or <code><button type="submit"></code> | "submit" | A push button that submits a form; <code>onclick</code> . |
| Text | <code><input type="text"></code> | "text" | A single-line text entry field; <code>onchange</code> . |
| Textarea | <code><textarea></code> | "textarea" | A multiline text entry field; <code>onchange</code> . |

TypeScript

- What is TypeScript?:
 - Programming Language: A language that includes all the existing JavaScript syntax, plus new TypeScript-specific syntax for defining and using types
 - Type Checker: A program that takes in a set of files written in JavaScript and/or TypeScript, checks for incorrect syntaxes
 - Compiler (Transpiler) : A program that runs the type checker, reports any issues, then outputs the equivalent JavaScript code.
 - Language Service: A program that uses the type checker to tell editors such as VS Code how to provide helpful utilities to developers

TypeScript

TS

1. TypeScript source -> TypeScript AST
2. AST is checked by typechecker
3. TypeScript AST -> JavaScript source

JS

4. JavaScript source -> JavaScript AST
5. AST -> bytecode
6. Bytecode is evaluated by runtime

TypeScript

- JS
 - Unconstrained
 - Checked at runtime
 - Errors surfaced at runtime (mostly)

```
function squareOf(n) {  
    return n * n  
}  
squareOf(2)           // evaluates to 4  
squareOf('z')         // evaluates to NaN
```

- TS
 - Constrained or bound statically
 - Checked at compile time
 - Errors surfaced at compile time (mostly)

```
function squareOf(n: number) {  
    return n * n  
}  
squareOf(2)           // evaluates to 4  
squareOf('z')         // Error TS2345: Argument of type '"z"' is not  
                        // parameter of type 'number'.
```

Advance JavaScript

Closure

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives a function access to its outer scope. In JavaScript, closures are created every time a function is created, at function creation time.

```
function makeFunc() {  
  const name = "Mozilla";  
  function displayName() {  
    console.log(name);  
  }  
  return displayName;  
}  
  
const myFunc = makeFunc();  
myFunc();
```

```
function makeAdder(x) {  
  return function (y) {  
    return x + y;  
  };  
}  
  
const add5 = makeAdder(5);  
const add10 = makeAdder(10);  
  
console.log(add5(2)); // 7  
console.log(add10(2)); // 12
```

Memory Management

Regardless of the programming language, the memory life cycle is pretty much always the same:

1. Allocate the memory you need
2. Use the allocated memory (read, write)
3. Release the allocated memory when it is not needed anymore

JavaScript automatically allocates memory when objects are created and frees it when they are not used anymore (garbage collection)

Memory Management: Allocation

Value initialization

```
const n = 123; // allocates memory for a number
const s = "azerty"; // allocates memory for a string

const o = {
  a: 1,
  b: null,
}; // allocates memory for an object and contained values

// (like object) allocates memory for the array and
// contained values
const a = [1, null, "abra"];

function f(a) {
  return a + 2;
} // allocates a function (which is a callable object)

// function expressions also allocate an object
someElement.addEventListener(
  "click",
  () => {
    someElement.style.backgroundColor = "blue";
  },
  false,
);
```

Function Call

```
const d = new Date(); // allocates a Date object

const e = document.createElement("div"); // allocates a DOM element
```

Memory Management: Using Values

Using values basically means reading and writing in allocated memory. This can be done by reading or writing the value of a variable or an object property or even passing an argument to a function.

Memory Management: Garbage collection

The majority of memory management issues occur at this phase. The most difficult aspect of this stage is determining when the allocated memory is no longer needed.

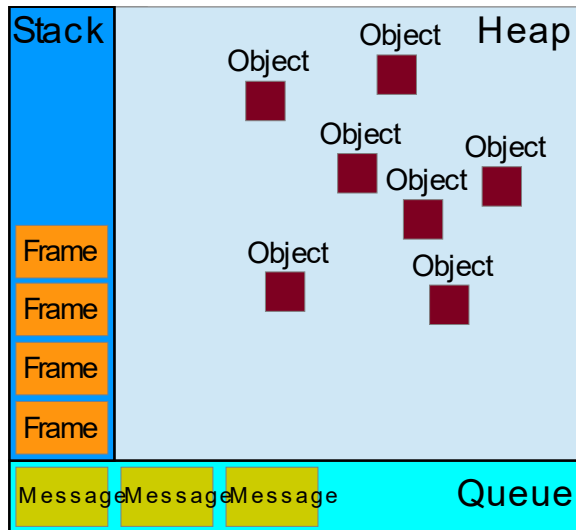
The main concept that garbage collection algorithms rely on is the concept of reference. Within the context of memory management, an object is said to reference another object if the former has access to the latter (either implicitly or explicitly).

Reference-counting Algorithm -> determining whether or not an object is still needed to determining if an object still has any other objects referencing it (not used anymore in modern JavaScript engine)

Mark-and-sweep Algorithm -> reduces the definition of "an object is no longer needed" to "an object is unreachable" (all modern engines ship a mark-and-sweep garbage collector)

Concurrency model and Event Loop

JavaScript has a runtime model based on an **event loop**, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks. This model is quite different from models in other languages like C and Java.



JavaScript runtime concepts (only a theoretical model)

Modern JavaScript engines implement and heavily optimize this.

JavaScript program is *single-threaded*. A thread is a sequence of instructions that a program follows. Because the program consists of a single thread, it can only do one thing at a time: so if it is waiting for our long-running synchronous call to return, it can't do anything else.

Runtime Concept: Stack

Function calls form a stack of frames.

```
function foo(b) {  
  const a = 10;  
  return a + b + 11;  
}  
  
function bar(x) {  
  const y = 3;  
  return foo(x * y);  
}  
  
const baz = bar(7); // assigns 42 to baz
```

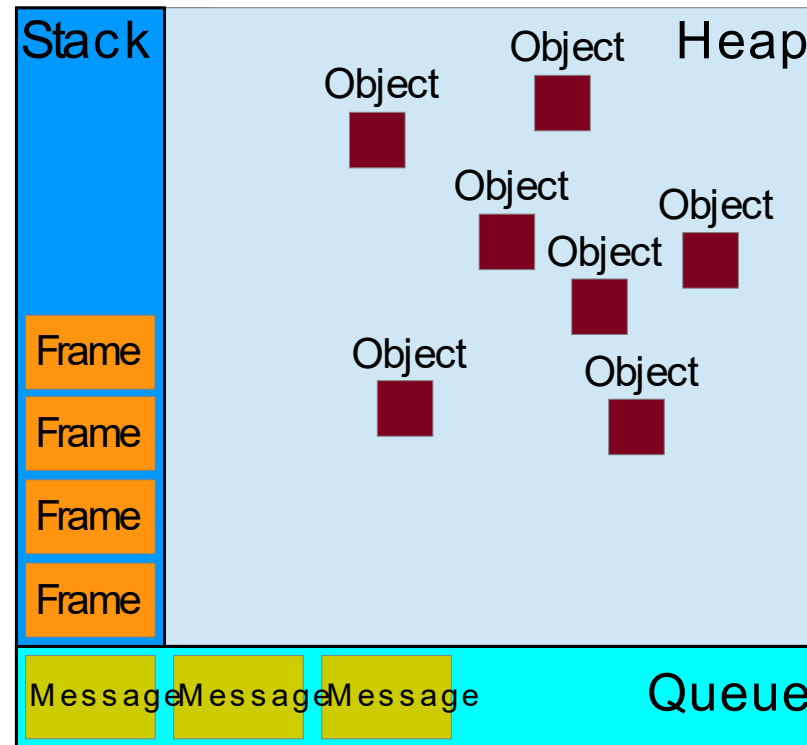
Order of operations:

1. When calling **bar**, a first frame is created containing references to bar's arguments and local variables.
2. When bar calls **foo**, a second frame is created and pushed on top of the first one, containing references to foo's arguments and local variables.
3. When **foo** returns, the top frame element is popped out of the stack (leaving only bar's call frame).
4. When **bar** returns, the stack is empty.

Note that the arguments and local variables may continue to exist, as they are stored outside the stack

Runtime Concept: Heap

Objects are allocated in a heap which is just a name to denote a large (mostly unstructured) region of memory.



Runtime Concept: Queue

A JavaScript runtime uses a message queue, which is a list of messages to be processed. Each message has an associated function that gets called to handle the message.

At some point during the event loop, the runtime starts handling the messages on the queue, starting with the oldest one. To do so, the message is removed from the queue and its corresponding function is called with the message as an input parameter. As always, calling a function creates a new stack frame for that function's use.

The processing of functions continues until the stack is once again empty. Then, the event loop will process the next message in the queue (if there is one).

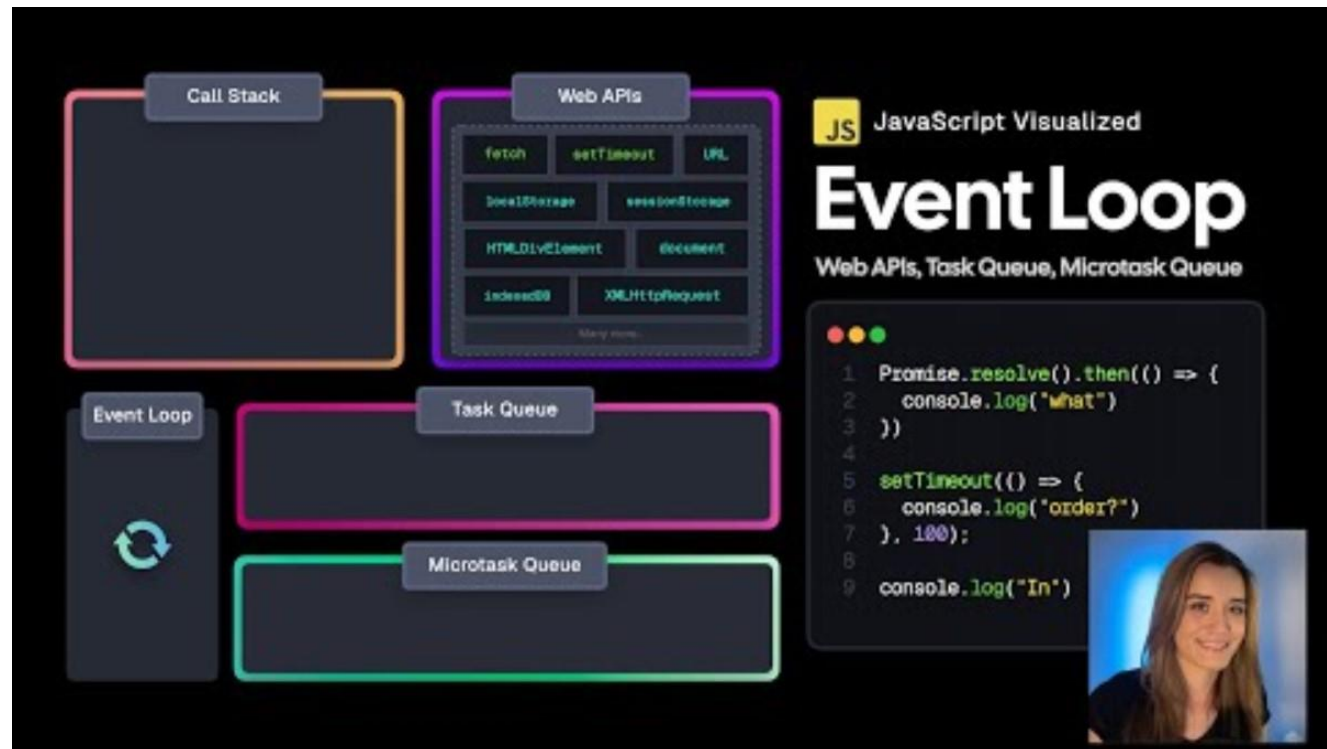
The Event Loop

The event loop got its name because of how it's usually implemented, which usually resembles:

```
while (queue.waitForMessage()) {  
  queue.processNextMessage();  
}
```

A very interesting property of the event loop model is that JavaScript, unlike a lot of other languages, never blocks.

Legacy exceptions exist like alert or synchronous XHR, but it is considered good practice to avoid them



Good Visualization of the Event Loop:

<https://www.youtube.com/watch?v=eiC58R16hb8>

Asynchronous JavaScript

When multiple related things happen without any being dependent on the completion of previous happenings, they are asynchronous.

```
const MAX_PRIME = 1000000;

function isPrime(n) {
  for (let i = 2; i <= Math.sqrt(n); i++) {
    if (n % i === 0) {
      return false;
    }
  }
  return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generatePrimes(quota) {
  const primes = [];
  while (primes.length < quota) {
    const candidate = random(MAX_PRIME);
    if (isPrime(candidate)) {
      primes.push(candidate);
    }
  }
  return primes;
}
```

We have synchronous program that a very inefficient algorithm to generate multiple large prime numbers.

What if the synchronous function takes a long time?

A downside of the JavaScript runtime model is that if a function takes too long to complete, the web application is unable to process user interactions like click or scroll. The browser mitigates this with the "a script is taking too long to run" dialog.

Asynchronous JavaScript: Event Handler

Event handlers are really a form of asynchronous programming: you provide a function (the event handler) that will be called, not right away, but whenever the event happens. If "the event" is "the asynchronous operation has completed", then that event could be used to notify the caller about the result of an asynchronous function call.

Example using
asynchronous API:
XMLHttpRequest

```
const log = document.querySelector(".event-log");

document.querySelector("#xhr").addEventListener("click", () => {
  log.textContent = "";

  const xhr = new XMLHttpRequest();

  xhr.addEventListener("loadend", () => {
    log.textContent = `${log.textContent}Finished with status: ${xhr.status}`;
  });

  xhr.open(
    "GET",
    "https://raw.githubusercontent.com/mdn/content/main/files/en-us/_wikihistory.json",
  );
  xhr.send();
  log.textContent = `${log.textContent}Started XHR request\n`;
});
```


About Event Listeners in JavaScript

- Event Handler concept

Event Handler is function that handles when particular Event happened at specified target. Object representing the event is passed as the first argument to the event handler. This event object either implements or is derived from the [Event](#) interface.

Used not only on UI, but also in workers, websocket, etc.

- Two approaches:
 - onevent: simplicity and one to one.
 - addEventListener and removeEventListener:

Very Flexible and Specific, Allows multiple event listener attached to a target. Also enable Event Bubbling/Capture phase to be specified (for DOM event)

```
const btn = document.querySelector('button');

function greet(event) {
  console.log('greet:', event)
}

btn.onclick = greet;
```

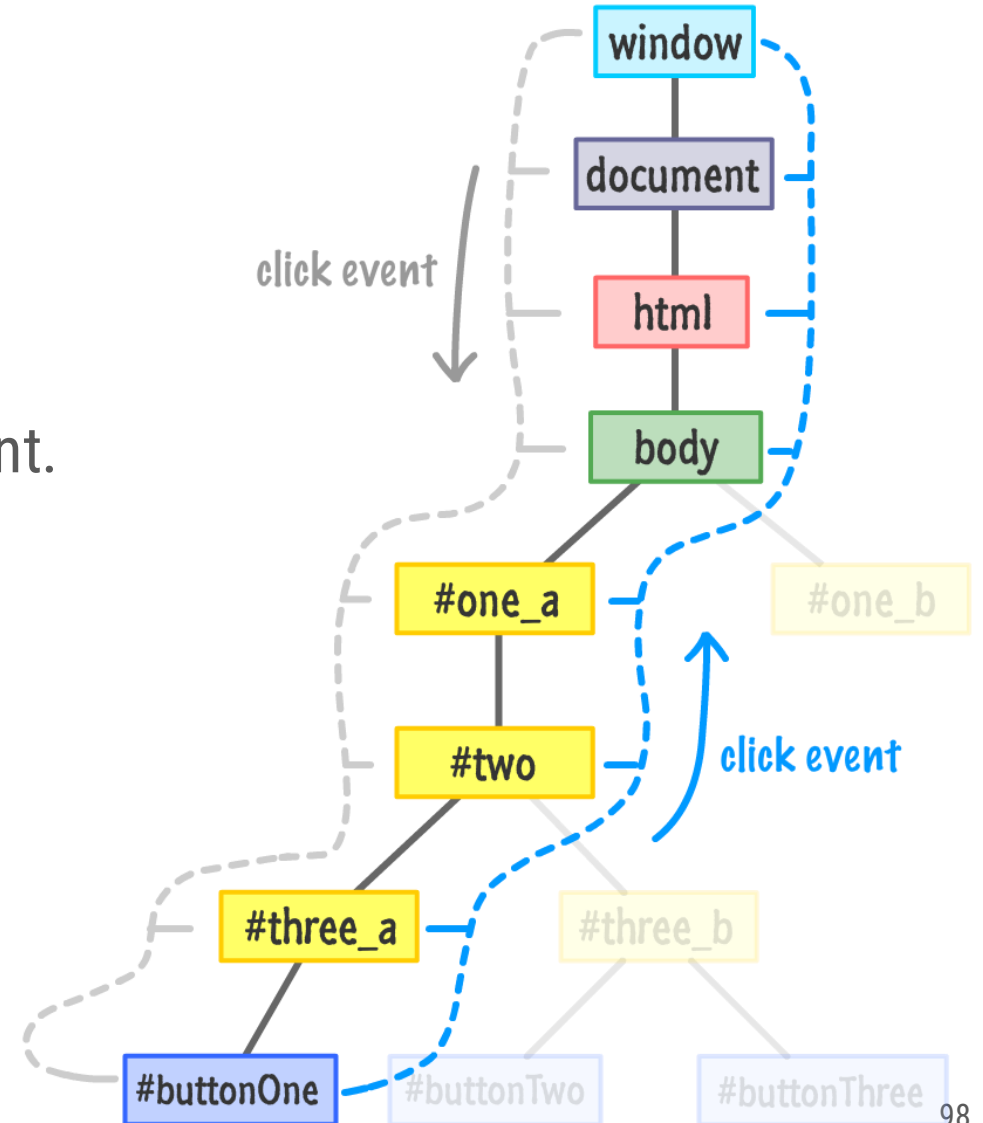
```
const btn = document.querySelector('button');

function greet(event) {
  console.log('greet:', event)
}

btn.addEventListener('click', greet);
```

Event Bubble/Capture

- The standard DOM Events describes 3 phases of event propagation:
 - Capturing phase – the event goes down to the element.
 - Target phase – the event reached the target element.
 - Bubbling phase – the event bubbles up from the element.
- Event.target is the deepest nested element being the target.
- "this" refer to the current element.



Asynchronous JavaScript: Callback

A callback is just a function that's passed into another function, with the expectation that the callback will be called at the appropriate time.

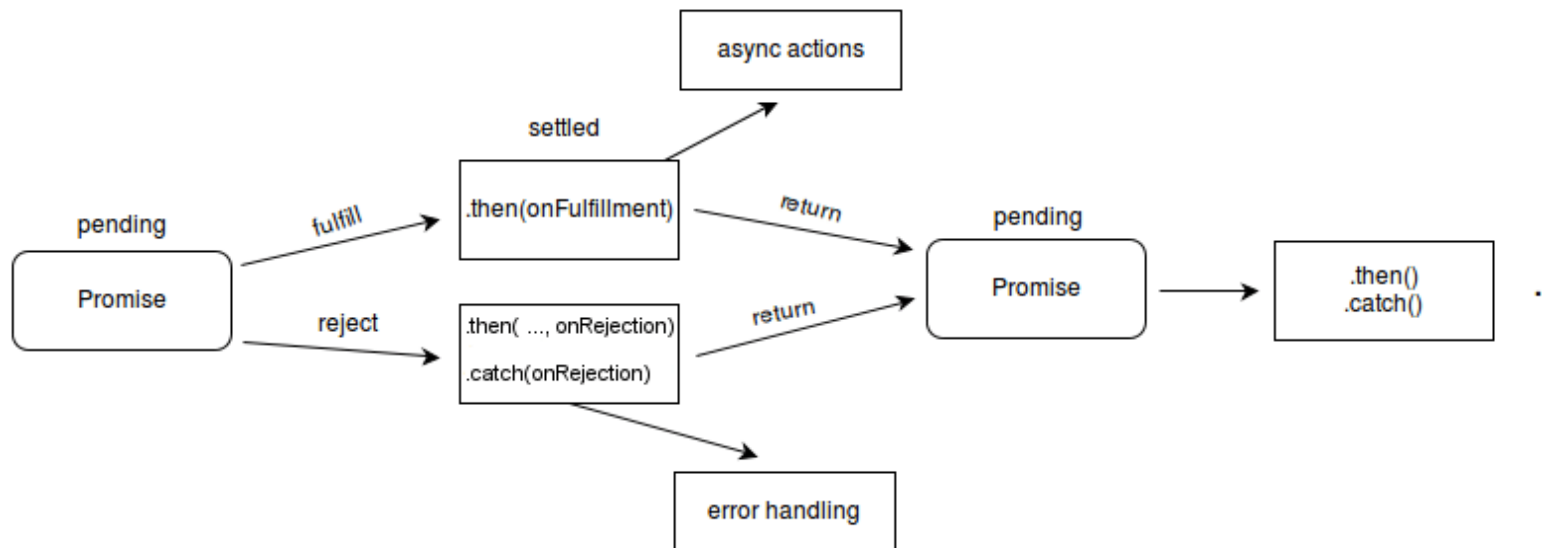
```
function doStep1(init, callback) {  
  const result = init + 1;  
  callback(result);  
}  
  
function doStep2(init, callback) {  
  const result = init + 2;  
  callback(result);  
}  
  
function doStep3(init, callback) {  
  const result = init + 3;  
  callback(result);  
}  
  
function doOperation() {  
  doStep1(0, (result1) => {  
    doStep2(result1, (result2) => {  
      doStep3(result2, (result3) => {  
        console.log(`result: ${result3}`);  
      });  
    });  
  });  
}  
  
doOperation();
```

However, callback-based code can get hard to understand when the callback itself has to call functions that accept a callback. This is a common situation if you need to perform some operation that breaks down into a series of asynchronous functions.

This is sometimes called "callback hell" or the "pyramid of doom".

Asynchronous JavaScript: Promise

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason



pending: initial state, neither fulfilled nor rejected.

fulfilled: meaning that the operation was completed successfully.

rejected: meaning that the operation failed.

Asynchronous JavaScript: Promise creation and chaining

Example below create and return new Promise Object. Then we handle or reject using then interface.

```
const myPromise = new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve("foo");  
  }, 300);  
});
```

```
myPromise  
  .then(handleFulfilledA, handleRejectedA)  
  .then(handleFulfilledB, handleRejectedB)  
  .then(handleFulfilledC, handleRejectedC);
```

The promise chain is what you need when your operation consists of several asynchronous functions, and you need each one to complete before starting the next one.

This special then() function returns another promise p. Depends on the return of value of the handler which may:

- returns a value: p gets **fulfilled** with the returned value as its value.
- doesn't return anything: p gets **fulfilled** with undefined as its value.
- throws an error: p gets **rejected** with the thrown error as its value.
- returns another pending promise: p is **pending** and becomes fulfilled/rejected with that promise's value as its value immediately after that promise becomes fulfilled/rejected.

Asynchronous JavaScript: Fetch example

```
// Make a request for user.json
fetch('/article/promise-chaining/user.json')
  // Load it as json
  .then(response => response.json())
  // Make a request to GitHub
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  // Load the response as json
  .then(response => response.json())
  // Show the avatar image (githubUser.avatar_url) for 3 seconds (maybe)
  .then(githubUser => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => img.remove(), 3000); // (*)
  });
```

Fetch is the modern replacement for XMLHttpRequest: unlike XMLHttpRequest (XHR), which uses callbacks, Fetch is promise-based.

The `fetch()` function returns a Promise which is fulfilled with a Response object representing the server's response. You can then check the request status and extract the body of the response in various formats, including text and JSON, by calling the appropriate method on the response.

Asynchronous JavaScript: Async Await

The `async` keyword gives you a simpler way to work with asynchronous promise-based code. Adding `async` at the start of a function makes it an async function:

```
async function fetchProducts() {
  try {
    // after this line, our function will wait for the `fetch()` call to be settled
    // the `fetch()` call will either return a Response or throw an error
    const response = await fetch(
      "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-
store/products.json",
    );
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    // after this line, our function will wait for the `response.json()` call to be
    settled
    // the `response.json()` call will either return the parsed JSON object or throw
    an error
    const data = await response.json();
    console.log(data[0].name);
  } catch (error) {
    console.error(`Could not get products: ${error}`);
  }
}

fetchProducts();
```

Inside an async function, you can use the `await` keyword before a call to a function that returns a promise.

The `async` and `await` keywords make it easier to build an operation from a series of consecutive asynchronous function calls, avoiding the need to create explicit promise chains, and allowing you to write code that looks just like synchronous code (a cleaner style).