# Τεχνολογίες Εφαρμογών Ιστού

Javascript

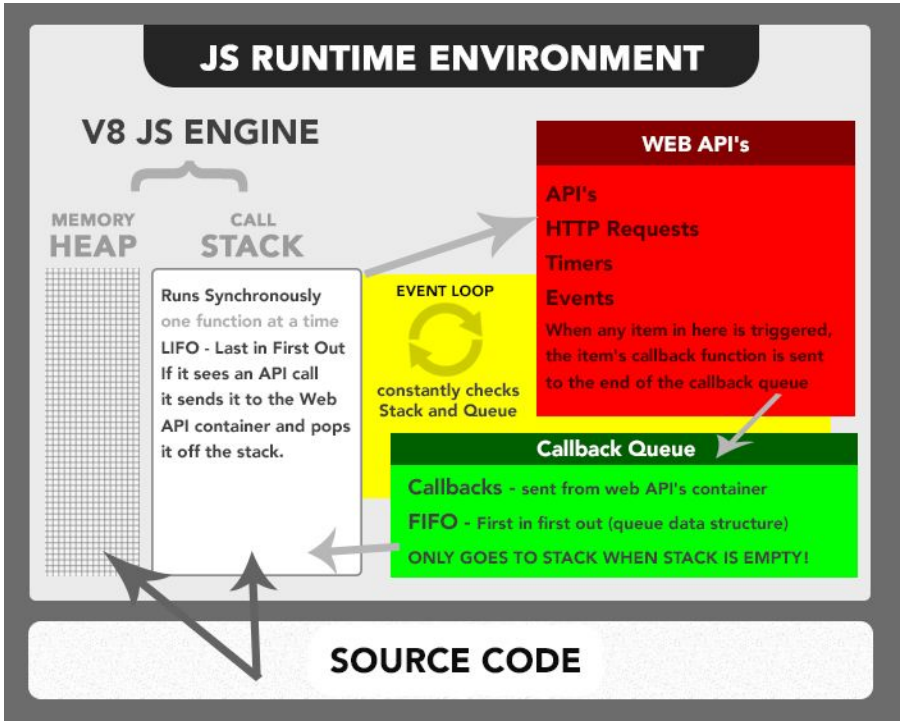
Konstantinos Tserpes

# Introduction

- Scripting language
- Code is parsed in a runtime
  - Google V8 engine
  - Mozilla SpiderMonkey
- Specification: ECMAScript
  - Course emphasis on ECMAScript 6 (ES6)

# JS Runtime: V8

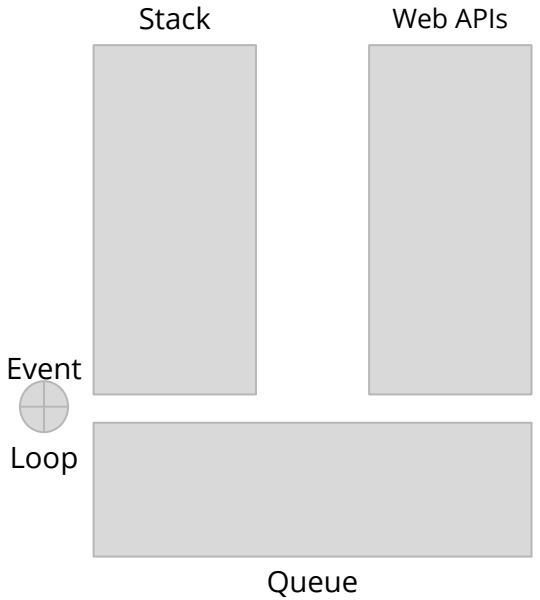
- Event-driven programming
  - Asynchronous
  - Non-blocking I/Os




Source: https://medium.com/@olinations/the-javascript-runtime-environment-d58fa2e60dd0

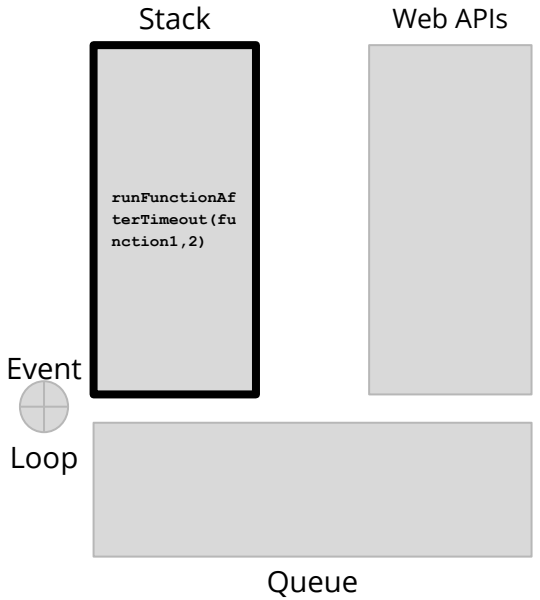
# Example

```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2’’; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code3
}

function2();
function3();
```



# Example

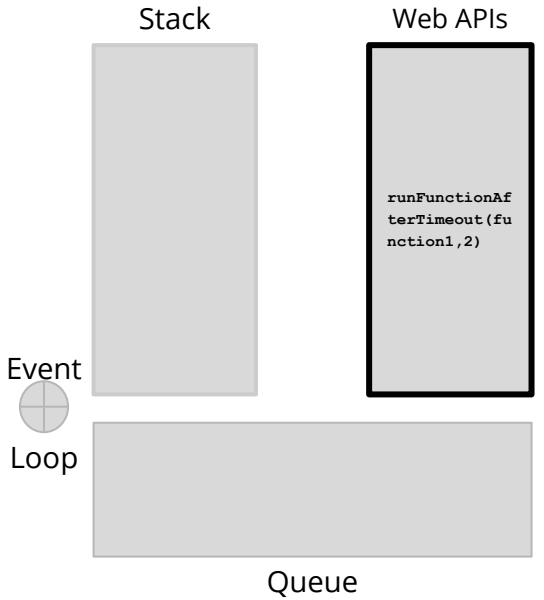
```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2’’; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code3
}

function2();
function3();
```



# Example

*t=0*

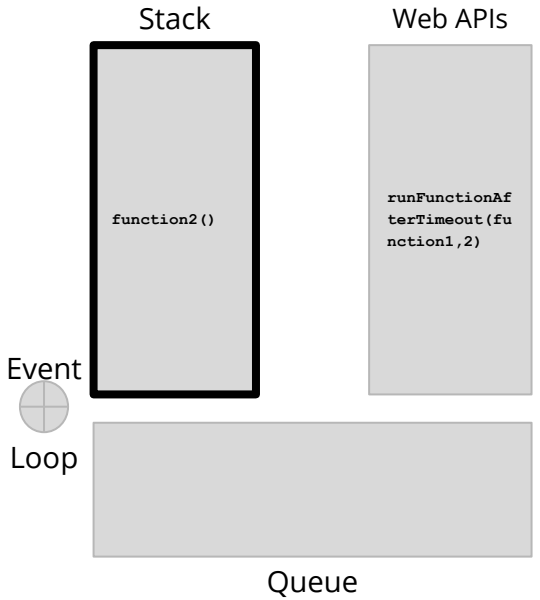
```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2''; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code3
}

function2();
function3();
```



# Example

*t=1*

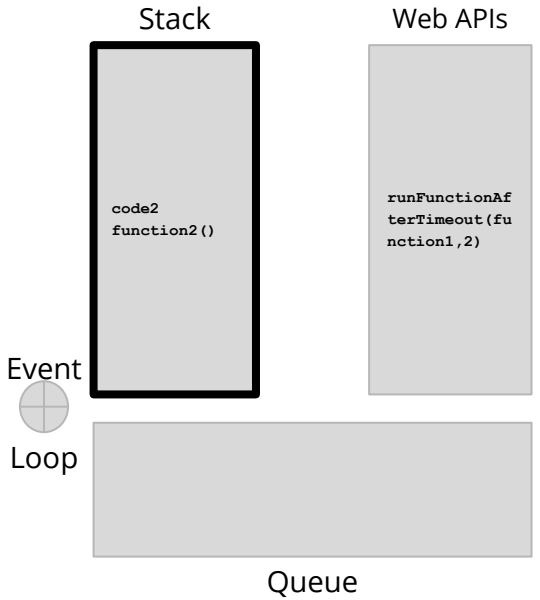
```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2’’; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code3
}

function2();
function3();
```



# Example

*t=1*

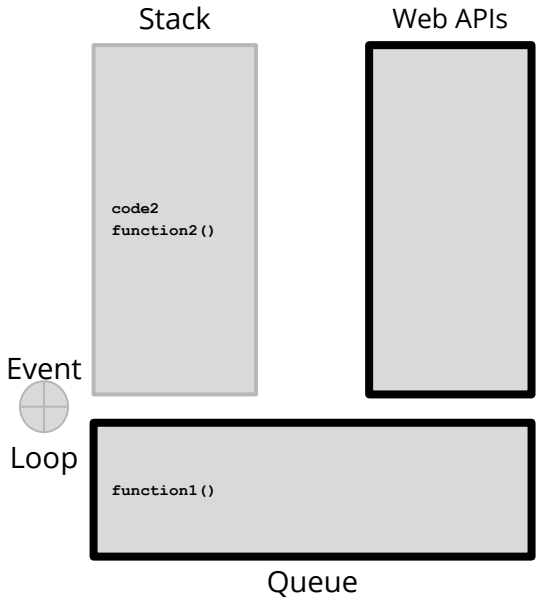
```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2’’; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code3
}

function2();
function3();
```



# Example

*t=2*

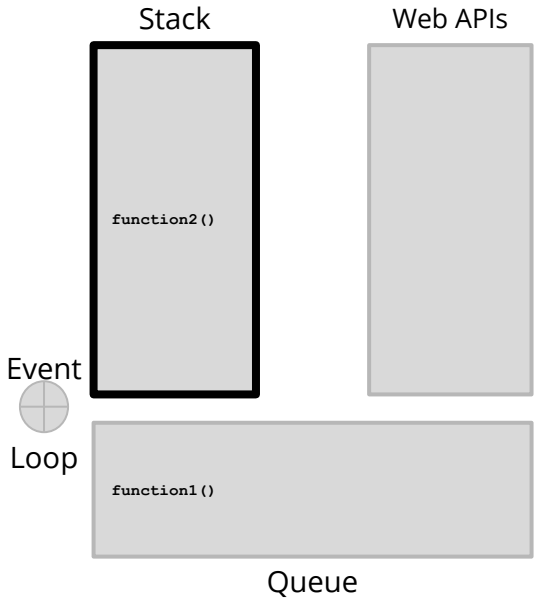
```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2''; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code3
}

function2();
function3();
```



# Example

*t*=-

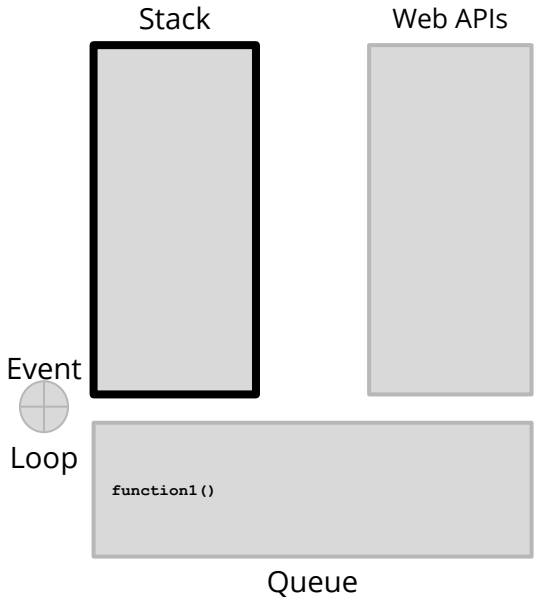
```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2’’; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code3
}

function2();
function3();
```



# Example

*t*=-

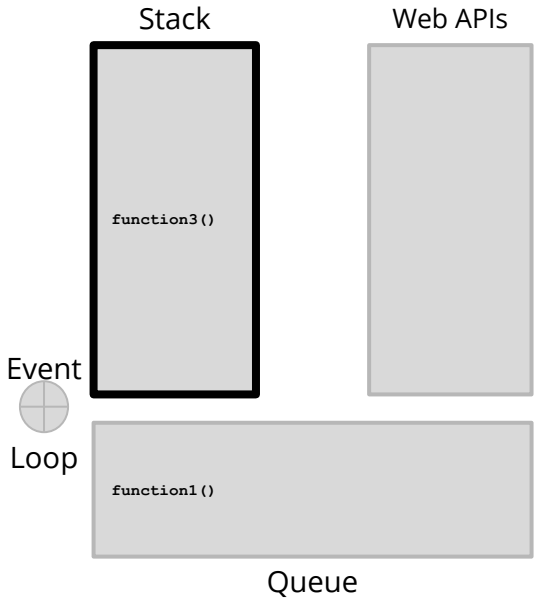
```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2’’; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code3
}

function2();
function3();
```



# Example

*t*=-

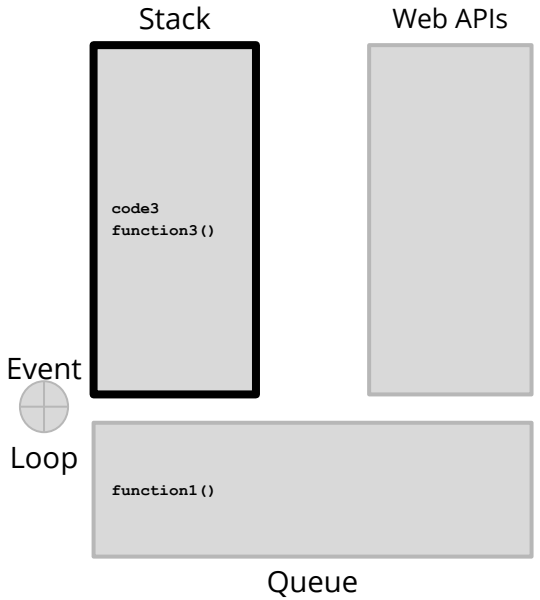
```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2’’; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code3
}

function2();
function3();
```



# Example

*t*=-

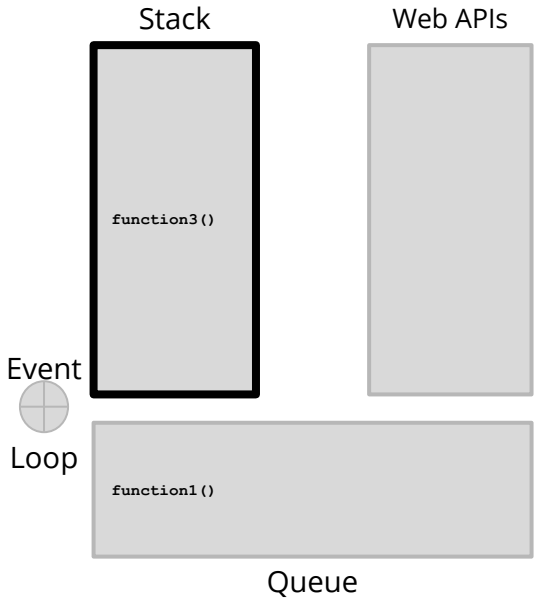
```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2’’; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code3
}

function2();
function3();
```



# Example

*t*=-

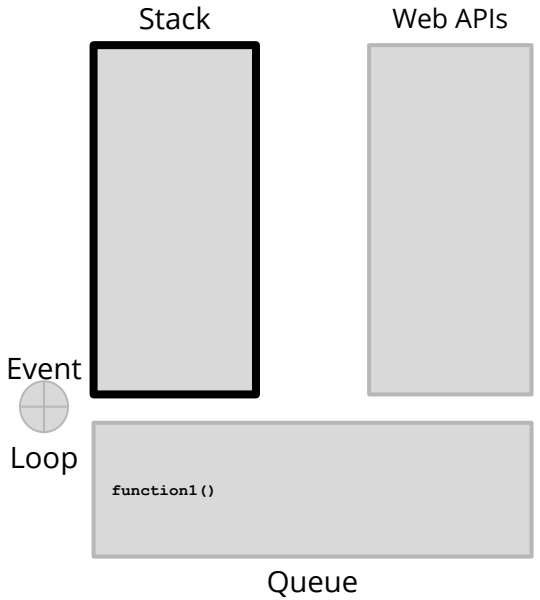
```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2’’; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code2
}

function2();
function3();
```



# Example

*t*=-

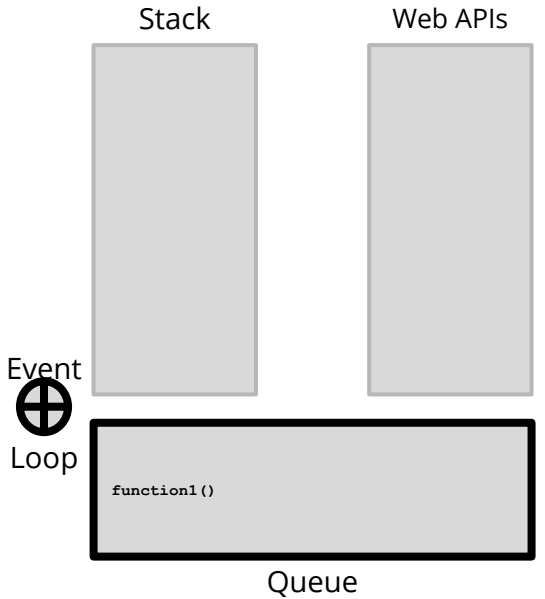
```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2’’; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code2
}

function2();
function3();
```



# Example

*t*=-

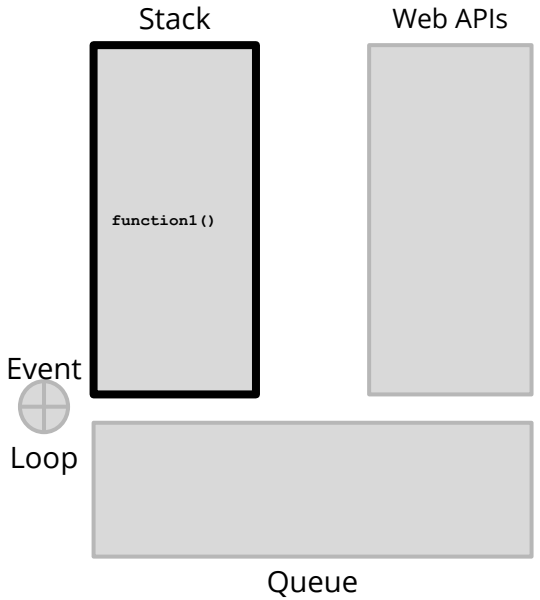
```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2’’; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code2
}

function2();
function3();
```



# Example

*t*=-

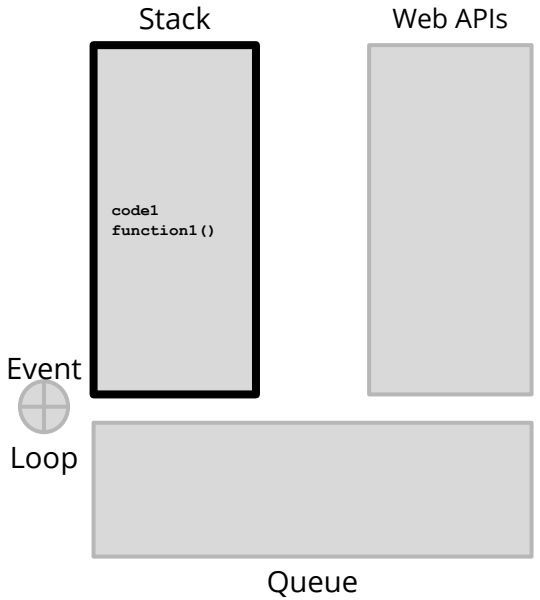
```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2’’; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code2
}

function2();
function3();
```



# Example

*t*=-

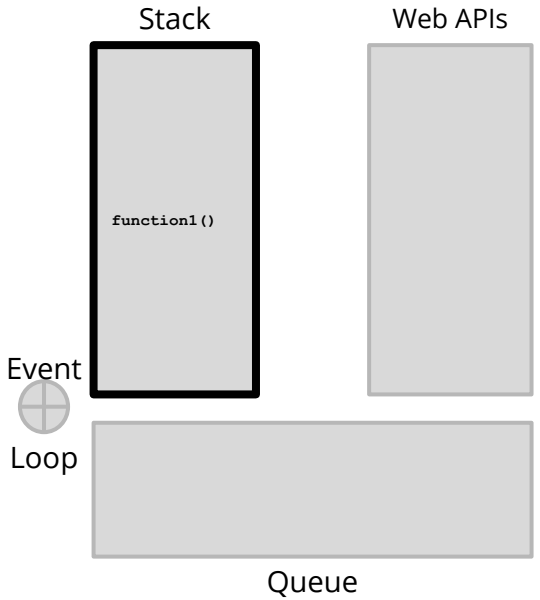
```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2’’; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code2
}

function2();
function3();
```



# Example

*t*=-

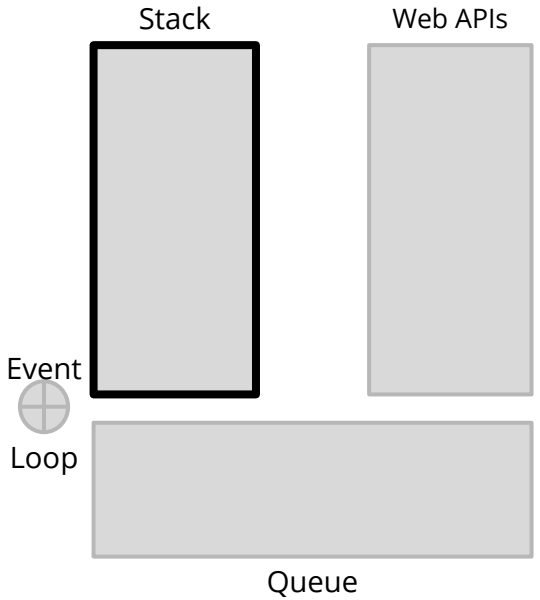
```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2’’; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code2
}

function2();
function3();
```



# Example

*t*=-

```
function1{
      //code1
}
runFunctionAfterTimeout(function1,2); //timeout = 2’’; assume a really slow machine
function2(){
      //code2
}
function3(){
      //code2
}

function2();
function3();
```



# Characteristics

- curly-bracket syntax
  - Like Java, C
- dynamic typing
  - Unlike Java, like Python
- prototype-based object-orientation
  - Like Java
- first-class functions (treats functions as first-class citizens, e.g. can assign them to variables)
  - Like Java Lambda Expressions, Unlike C
- high-level
  - Unlike Assembly, Like Python
- Interpreted with just-in-time (JIT) compilation
  - Profilers check the code for warm/hot chunks and create and compile stubs

# Variables

- Case-sensitive names
- Types are not explicitly declared
- Types are calculated during assignments at run-time
- Primitive types:
  - string, number, bigint, boolean, undefined, and symbol
  - pass by value
- The value of unassigned variables is “undefined”
- Objects can also be assigned to variables
  - pass by reference
- A special object: “null”
  - It’s value is “undefined”
  - Any structured type is derived from “null”

# Scope

- Declarations using the keyword “var”
  - function-scoped variables
- Declarations using the keywords “let” and “const” (ECMAScript 6)
  - block-scoped variables
- Declarations with no keyword
  - globally-scoped variables

# Function-scoped variables

```
function f() {
   var x=0;
   console.log(x); //logs "0"
   console.log(y); //logs "6"
}
var y=6;
f();
```

# Block-scoped variables

```
   {
      let y=6;
   }
   console.log(y); //logs "ReferenceError: Can't
find variable: y"
```

```
function func() {
      let foo = 5;
      if (···) {
            let foo = 10; // shadows outer `foo`
            console.log(foo); // 10
      }
      console.log(foo); // 5
}
```

# Let Vs Const

- Variables created by let are mutable

```
let foo = 'abc';
foo = 'def';
console.log(foo); // def
```

- Variables created by const, constants, are immutable

```
const foo = 'abc';
foo = 'def'; // TypeError
```

# Globally-scoped variables

```
function f(){
    console.log(x); //logs ‘3’
    x=6;
}
x=3;
f();
console.log(x); //logs ‘6’
```

# Idea

- You can think of variables as a structure with 5 fields
  - Name
  - Address
  - Value (always as text)
  - Type
  - Scope

# Quiz

```
var a = "1200";
var b = 1200;
alert(a == b); //alerts ???
alert(a === b); //alerts ???
```

# Hoisting

```
var foo = 1;
function bar() {
   if (!foo) {
      var foo = 10;
   }
   alert(foo);
}
bar();
```

alerts?

# Hoisting

- Function and variable declarations are always moved (“hoisted”) invisibly to the top of their containing scope by the JavaScript interpreter.

# Hoisting

```
var foo = 1;
function bar() {
   if (!foo) {
      var foo = 10;
   }
   alert(foo);
}
bar();
```


```
var foo = 1;
function bar() {
   var foo;
   if (!foo) {
      foo = 10;
   }
   alert(foo);
}
bar();
```

# Quiz

- alert(typeof(null));
- alert(typeof(undefined));
- alert(null !== undefined)
- alert(null == undefined)

# Quiz

- alert(typeof(null));
  - object
- alert(typeof(undefined));
- alert(null !== undefined)
- alert(null == undefined)

# Quiz

- alert(typeof(null));
  - object
- alert(typeof(undefined));
  - undefined
- alert(null !== undefined)

- alert(null == undefined)

# Quiz

- alert(typeof(null));
  - object
- alert(typeof(undefined));
  - undefined
- alert(null !== undefined)
  - true
- alert(null == undefined)

# Quiz

- alert(typeof(null));
  - object
- alert(typeof(undefined));
  - undefined
- alert(null !== undefined)
  - true
- alert(null == undefined)
  - true

# Syntax (control structures)

- IF

```
if(condition1){ … }
else if(condition2){  … }
else{ … }
```

- SWITCH

```
switch(condition){
     case value1:
          ...
          break;
     case value2:
          ...
          break;
     default:
          …
}
```

# Syntax (loops)

- For

```
for(initialization;condition;step){
 code
}
```

- While

```
while(condition){
 code
}
```

- Do… While

```
do {
    code
while (condition)
```

# Objects

- JavaScript is designed on a simple object-based paradigm
- An object is a collection of properties
  - a property is an association between a name (or key) and a value
  - a property's value can be a function, in which case the property is known as a method.
- Predefined and custom objects

# Instantiating Objects

- Create a new object with the keyword “new”

```
var myCar = new Object();
myCar.make = 'Ford';
myCar.model = 'Mustang';
myCar.year = 1969;
```

# Object initialization

```
var myCar = {
    make: 'Ford',
    model: 'Mustang',
    year: 1969
};
```

Powerful because:

```
let m = eval({make: 'Ford',model:'Mustang',year: 1969})
};
console.log(m.model)    //prints ‘Ford’
```

# Objects == Associative Arrays

- Access or set objects using a bracket notation

```
myCar['make'] = 'Ford';
myCar['model'] = 'Mustang';
myCar['year'] = 1969;
```

# Object as execution context

- In OO languages, everything runs in an object
  - When a statement or a function is executed it does so in an instantiated object
- The currently instantiated object defines the ‘execution context’
- JS passes at runtime a reference of the execution context to its property ‘this’
- As the program runs, the execution context may change, and so is the value of ‘this’
- Special case of execution context: Global object
  - A global object is an object that always exists in the global scope. In a browser it is the ‘window’ object

# Example

```
function f1() {
  return this;
}
// In a browser:
f1() === window; // true
```

# Creating objects using constructor functions

- Define the object type by writing a constructor function.
  - convention: use a capital initial letter
- Create an instance of the object with “new”

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
var kenscar = new Car('Nissan', '300ZX', 1992);
var vpgscar = new Car('Mazda', 'Miata', 1990);
```

# Methods

```
function displayCar() {
  var result = `A Beautiful ${this.year} ${this.make} ${this.model}`;
  return result;
}
function Car(make, model, year, owner) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
  this.displayCar = displayCar;
}
var car1 = new Car(...);
var car2 = new Car(...);
console.log(car1.displayCar());
console.log(car2.displayCar());
```

# Functions

- Functions are declared with the keyword ‘function’
- They inherit from the Object prototype and they can be assigned key/value pairs
- Functions are first-class citizens in JS (examples)
  - Assign function to a variable
  - Pass function as argument
  - Return a function
    - Using a variable
    - Using double parentheses
- They always return something
  - Whatever the ‘return’ statement defines
  - ‘undefined’ in the absence of the ‘return’ statement
  - Reference to ‘this’ in the case of constructor functions used with the ‘new’ keyword

# Expressions for defining functions

- function expression
- Immediately Invoked Function Expression
- function* expression
- Anonymous
- Arrow expression

# function expression

```
var myFunction = function namedFunction(){
    statements
}
```

# Immediately Invoked Function Expression

- Useful before ECMAScript6 (function-scoping)
  - avoids variable hoisting from within blocks
  - protects against polluting the global environment
  - allows public access to methods while retaining privacy for variables defined within the function

```
(function() {
    statements
})();
```

# function* expression

- Defines a ‘generator’ function, which returns a Generator object
- Generator object
  - Conforms to both the iterable protocol and the iterator protocol
  - Instance methods
    - Generator.prototype.next()
      - Returns a value yielded by the yield expression.
    - Generator.prototype.return()
      - Returns the given value and finishes the generator.
    - Generator.prototype.throw()
      - Throws an error to a generator (also finishes the generator, unless caught from within that generator).

# function* example

```
function* generator(i) {
  yield i;
  yield i + 10;
}
const gen = generator(10);
console.log(gen.next().value);
// expected output: 10
console.log(gen.next().value);
// expected output: 20
```

# Anonymous function expression

```
var myFunction = function() {
    statements
}
```

# Object methods using anonymous functions

```
function Car(make, model, year, owner) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
  this.displayCar = function(){
    var result = `A Beautiful ${this.year} ${this.make} ${this.model}`;
    return result;
  };
}
var car1 = Car(...);
var car2 = Car(...);
console.log(car1.displayCar());
console.log(car2.displayCar());
```

# Initialization with anonymous functions as methods

```
let myCar = {
    make:'Nissan',
    model: '300ZX',
    year: 1992,
    owner: 'John',
    displayCar: function(){
        console.log(`A Beautiful ${this.year} ${this.make}
${this.model}`);
    }
};
myCar.displayCar();
```

# Arrow function expression

- Compact alternative to traditional function expression
- With limitations and extra properties

```
// Traditional Function
function (a){
  return a + 100;
}
// Arrow Function Break Down
(a) => {
  return a + 100;
}
(a) => a + 100;
a => a + 100;
```

# ‘this’ and Arrow functions: Traditional functions

```
window.age = 10;
function Person() {
  this.age = 42;
  setTimeout(function () {
    console.log("this.age", this.age); // yields "10" because the
function executes on the window object scope
  }, 100);
}

var p = new Person();
```

# ‘this’ and Arrow functions Vs Traditional functions

```
window.age = 10;
function Person() {
  this.age = 42;
  setTimeout(() => {
    console.log("this.age", this.age); // yields "42" because the
function executes on the Person scope
  }, 100);
}

var p = new Person();
```

# Quiz

```
for(var i = 0; i < 5; i++) {
    setTimeout(function() {
        console.log(i);
    },1000);
}
```

# Quiz

```
for(var i = 0; i < 5; i++) {
    setTimeout(function(i) {
        console.log(i);
    },1000);
}
```

# Awkward phenomenon

```
for(let i = 0; i < 5; i++) {
    setTimeout(function() {
        console.log(i);
    },1000);
}

//logs 0,1,2,3,4
```

```
for(var i = 0; i < 5; i++) {
    setTimeout(function() {
        console.log(i);
    },1000);
}

//logs 5,5,5,5,5
```

# Lexical scoping

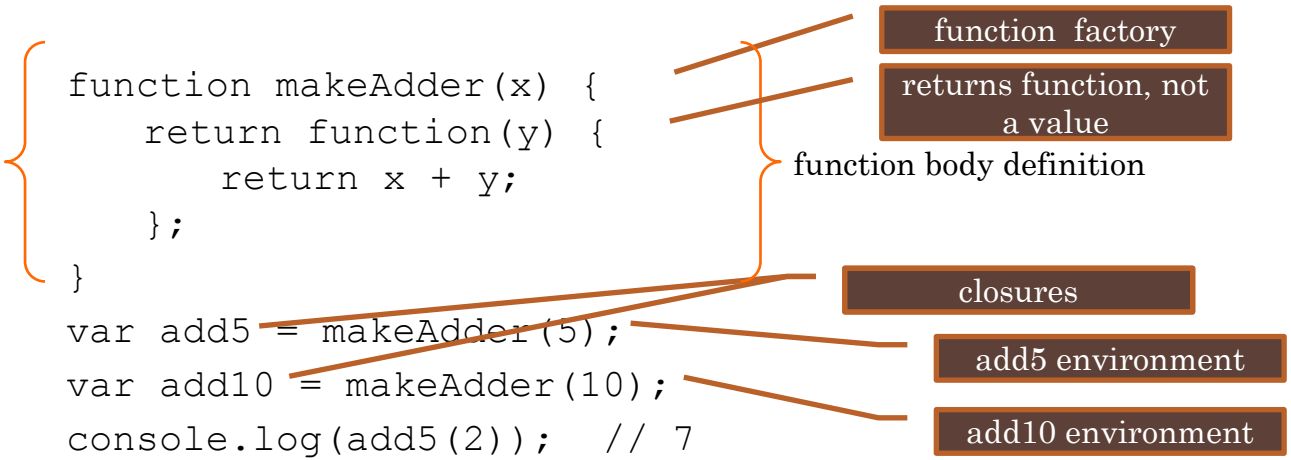
```
function init() {
  var name = 'Mozilla'; // name is a local variable created by init
  function displayName() { // displayName() is the inner function,
a closure
    alert(name); // use variable declared in the parent function
  }
  displayName();
}
init();  // displays 'Mozilla'
```

# Closures

```
function makeFunc() {
  var name = 'Mozilla';
  function displayName() {
    alert(name);
  }
  return displayName;
}

var myFunc = makeFunc();
myFunc();  // displays 'Mozilla'
```

# JavaScript Closures

- A closure is a special kind of object that combines two things:
  - a function
  - the lexical environment in which that function was created
    - The environment consists of any local variables that were in-scope at the time that the closure was created.



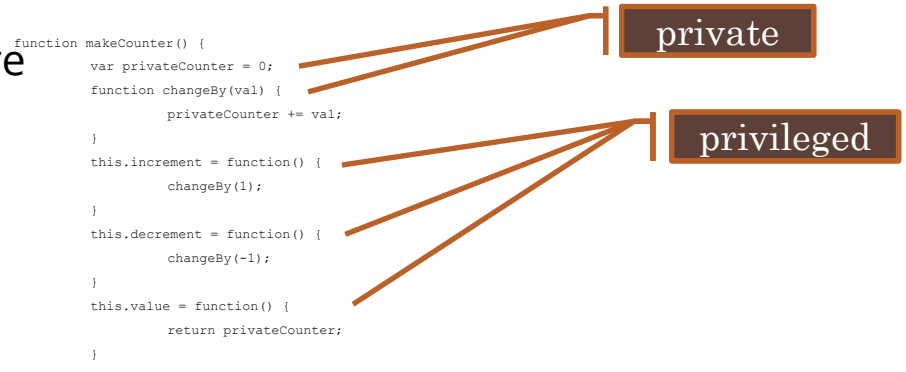

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

# Closures

- Definition
  - a closure is a stack frame which is allocated when a function starts its execution, and not freed after the function returns (as if a 'stack frame' was allocated on the heap rather than the stack!).
- Why are they important?
  - Scope is (by default) at the function level, not the block level
  - Much of what you do in practice in JavaScript is asynchronous/event driven

# Emulating private accessors with Closures

- No standard way to declare private methods




```
function makeCounter() {
        var privateCounter = 0;
        function changeBy(val) {
                privateCounter += val;
        }
        this.increment = function() {
                changeBy(1);
        }
        this.decrement = function() {
                changeBy(-1);
        }
        this.value = function() {
                return privateCounter;
        }
}
var Counter1 = new makeCounter();
var Counter2 = new makeCounter();
alert(Counter1.value()); /* Alerts 0 */
Counter1.increment();
Counter1.increment();
alert(Counter1.value()); /* Alerts 2 */
Counter1.decrement();
alert(Counter1.value()); /* Alerts 1 */
alert(Counter2.value()); /* Alerts 0 */
alert(Counter1.privateCounter); /* Alerts undefined */
Counter1.changeBy(3); /* Alerts TypeError: Counter1.changeBy is not a function */
```

# Inheritance

- All JS objects inherit from Object
- All objects include a ‘prototype’ property
    - a bucket for storing properties and methods when we are extending a class

```
function Person(first, last, age, gender, interests) {
  this.name = {
    first,
    last
  };
  this.age = age;
  this.gender = gender;
  this.interests = interests;
};
Person.prototype.greeting = function() {
  alert('Hi! I\'m ' + this.name.first + '.');
};
```

# Inheriting objects with ‘call’

```
function Person(first, last, age, gender, interests) {
  this.name = {
    first,
    last
  };
  this.age = age;
  this.gender = gender;
  this.interests = interests;
  this.greeting = function() {
    alert('Hi! I\'m ' + this.name.first + '.');
  };
};
function Teacher(first, last, age, gender, interests, subject) {
  Person.call(this, first, last, age, gender, interests);
  this.subject = subject;
}
let teacher = new Teacher('John','Doe',30,'male','history','art history');
teacher.greeting();
```

# Iterating through object properties

- for-in loop

```
for (const variable_name in object_name){
//statement (object_name[variable_name])
}
```

- Example

```
for (const prop_name in teacher)
    console.log(teacher[prop_name]);
```

# Quiz

```
var a = 5;
(function() {
  var a = 12;
  alert(a);
})();
```

```
var a = 10;
var x = (function() {
  var y = function() {
    var a = 12;
  };
  return function() {
    alert(a);
  }
})();
x();
```

```
var a = 10;
(function() {
  var a = 15;
  window.x = function() {
    alert(a);
  }
})();
x();
```

# JS for client-side programming

# JS in client-side programming

- Integrated in HTML

```
<html>
    <body>
    <script type="text/javascript">
        document.write("Hello World!");
    </script>
    </body>
</html>
```

# Executing JS in HTML

- Time of execution depends on their position in the page
- Strong event-driven programming element
- Declaring functions in <head> is a good practice
- You can always refer to an external JS file

```
<script src="code.js"></script>
```

# Quiz

- Open a text editor and save the file as “test.html”
- Write a script within the HTML code, that will print the numbers 1-99, omitting one of them
  - In the browser’s console
  - In the web page that it will load
- Run the test.html in your browser

# Events

- User-activity-related events
  - Mouse/keyboard actions
- Time-related events
  - `setTimeout(function, timeout)`
  - `setInterval(function, interval)`
- System-related events
  - Page loads, form submitted, I/O-related events

# Quiz

- Can I tell that an event occured by just looking the runtime structures as they evolve?

# Managing Events in the client

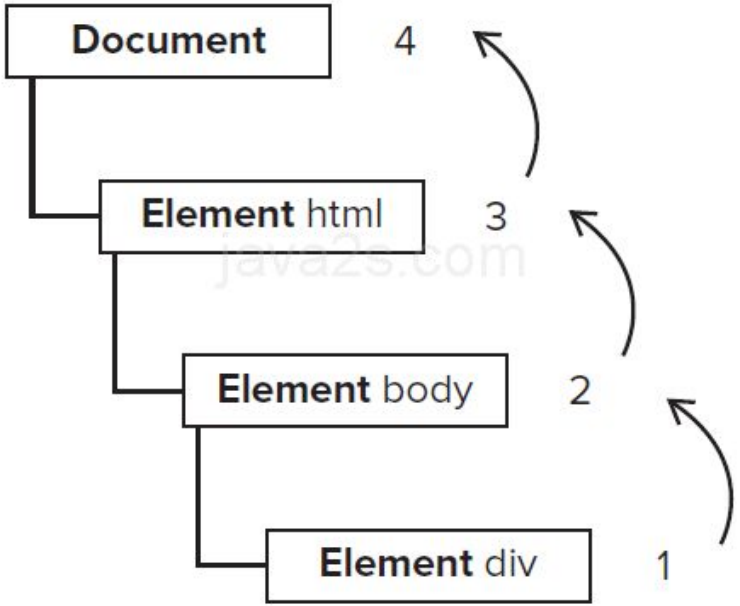
- Create *onevent handlers* in the HTML elements
  - `<input type=”button” value=”OK” onclick=”return function1();”/>`
  - Event handlers are HTML attributes
  - Mixing HTML and JS and without `<script>`
- Use `addEventListener` and a callback
  - target.addEventListener(type, listener [, options]);

# Quiz

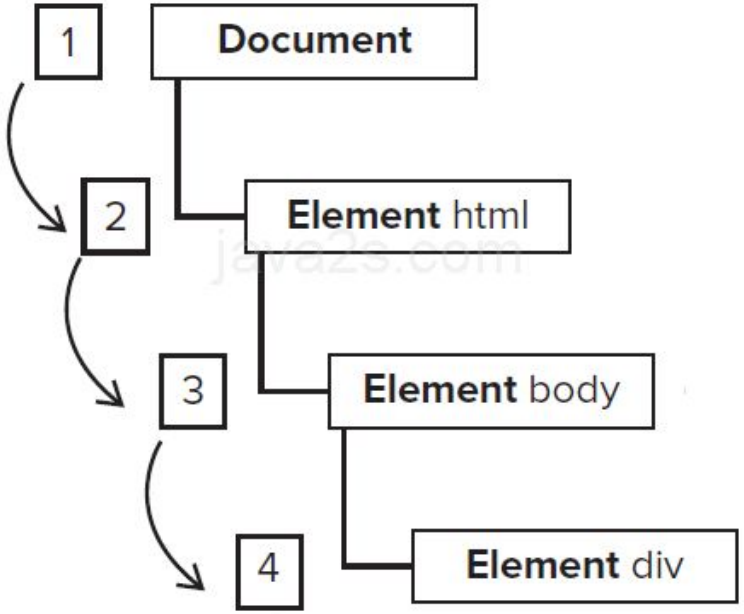
- Any question about the ordering of fired events?

# Event order

**Bubbling**




**Capturing**



# Example events

| Attribute | The event occurs when... |
|---|---|
| onblur | An element loses focus |
| onchange | The content of a field changes |
| onclick | Mouse clicks an object |
| ondblclick | Mouse double-clicks an object |
| onerror | An error occurs when loading a document or an image |
| onfocus | An element gets focus |
| onkeydown | A keyboard key is pressed |
| onkeypress | A keyboard key is pressed or held down |
| onkeyup | A keyboard key is released |
| onmousedown | A mouse button is pressed |
| onmousemove | The mouse is moved |
| onmouseout | The mouse is moved off an element |
| onmouseover | The mouse is moved over an element |
| onmouseup | A mouse button is released |
| onresize | A window or frame is resized |
| onselect | Text is selected |
| onunload | The user exits the page |
| **Property** | **Description** |
| altKey | Returns whether or not the "ALT" key was pressed when an event was triggered |
| button | Returns which mouse button was clicked when an event was triggered |
| clientX | Returns the horizontal coordinate of the mouse pointer when an event was triggered |
| clientY | Returns the vertical coordinate of the mouse pointer when an event was triggered |
| ctrlKey | Returns whether or not the "CTRL" key was pressed when an event was triggered |
| metaKey | Returns whether or not the "meta" key was pressed when an event was triggered |
| relatedTarget | Returns the element related to the element that triggered the event |
| screenX | Returns the horizontal coordinate of the mouse pointer when an event was triggered |
| screenY | Returns the vertical coordinate of the mouse pointer when an event was triggered |
| shiftKey | Returns whether or not the "SHIFT" key was pressed when an event was triggered |
| **Property** | **Description** |
| bubbles | Returns a Boolean value that indicates whether or not an event is a bubbling event |
| cancelable | Returns a Boolean value that indicates whether or not an event can have its default action prevented |
| currentTarget | Returns the element whose event listeners triggered the event |
| eventPhase | Returns which phase of the event flow is currently being evaluated |
| target | Returns the element that triggered the event |
| timeStamp | Returns the time stamp, in milliseconds, from the epoch (system start or event trigger) |
| type | Returns the name of the event |

# Quiz

- Create a webpage with a textbox (input type=”text”) that when clicked it will show a popup (alert()) stating something
- Create a link (<a href=”http://...”) that will do the same when the mouse cursor hovers over it (event: mouseover)

# JS Reference Objects

- JS Reference
  - Array
  - Boolean
  - Date
  - Error
  - JSON
  - Math
  - Number
  - Operators
  - RegExp
  - Statements
  - String
- HTML Element Objects Reference
  - ...

- **Web APIs**
  - Console
  - Geolocation
  - History
  - Storage
  - DOM
  - ...

# Quiz

- Write a script that will create an object with three properties (numbers) and a method that once invoked it will print in the console the  sum of the numbers
  - Using initialization
  - Using a constructor function

# Sources

- https://developer.mozilla.org/en-US/docs/Web/
- https://www.w3schools.com/