Information Visualization

Infovis on the Web

G. Da Lozzo, V. Di Donato, M. Patrignani

Copyright notice

Credits

- Parts of this matherial is inspired by
  - TM, http://www.teaching-materials.org/
- The first version of these slides were produced by

Valentino Di Donato

Giordano Da Lozzo

Web data visualization: motivations

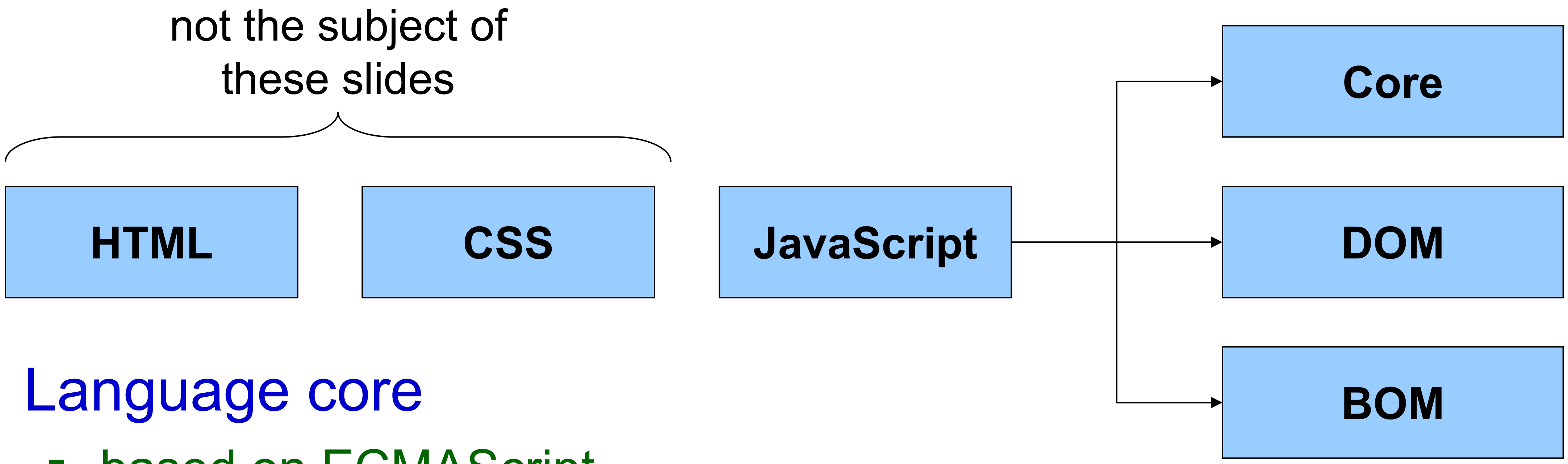
Visualizations aren’t truly visual unless they are seen. Getting your work out there for others to see is critical, and publishing on the Web is the quickest way to reach a global audience

Interactive Data Visualization for the Web,
Scott Murray ’13

Web data visualization: motivations

- Why Web visualization services?
  - quickest diffusion
  - global reach
  - operating system independency
- However we may have browser(s) dependency
  - this problem is more and more marginal

JavaScript basic ingredients



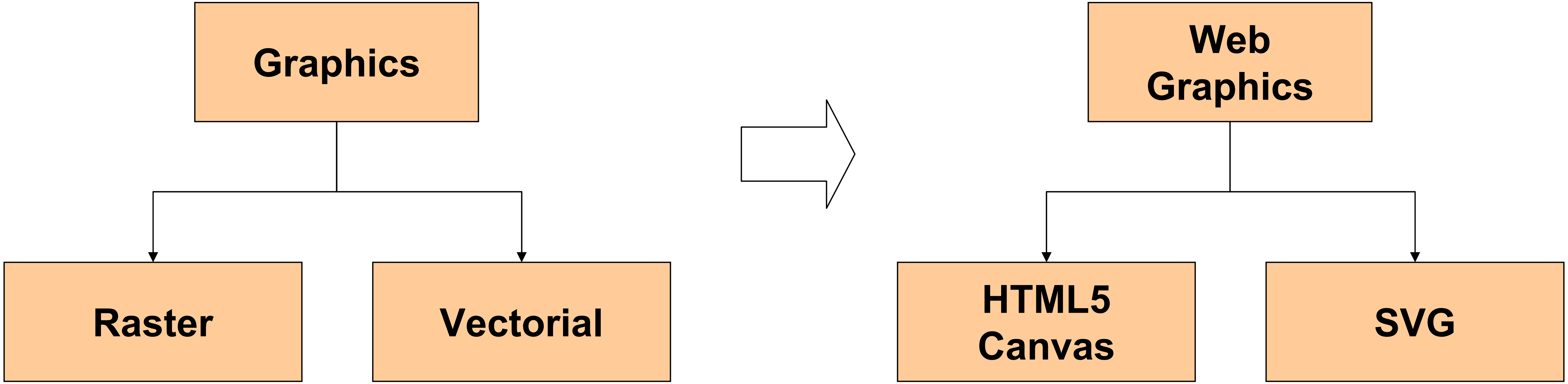
- Language core
  - based on ECMAScript
- Document Object Model (DOM)
  - API for HTML/XML documents
- Browser Object Model (BOM)
  - browser window manipulation

JavaScript basic ingredients

- JavaScript core language
  - based on ECMAScript specification (standard ISO)
    - other dialects of ECMAScript include JavaScript, Microsoft Jscript, Adobe Flash ActionScript
  - provides core scripting capabilities for the browser
- Document Object Model (DOM)
  - data model that is created for each page that is loaded
  - HTML DOM provides also an API to manipulate the model
- Browser Object Model (BOM)
  - allows to perform actions that do not directly relate to the page content (window position, decorations, status bar text, etc)
  - no official standards

Graphics basic ingredients

- By using JavaScript it is possible to produce
  - raster graphic contents
  - vectorial graphic contents



JavaScript

Crash course

JS: What is it?

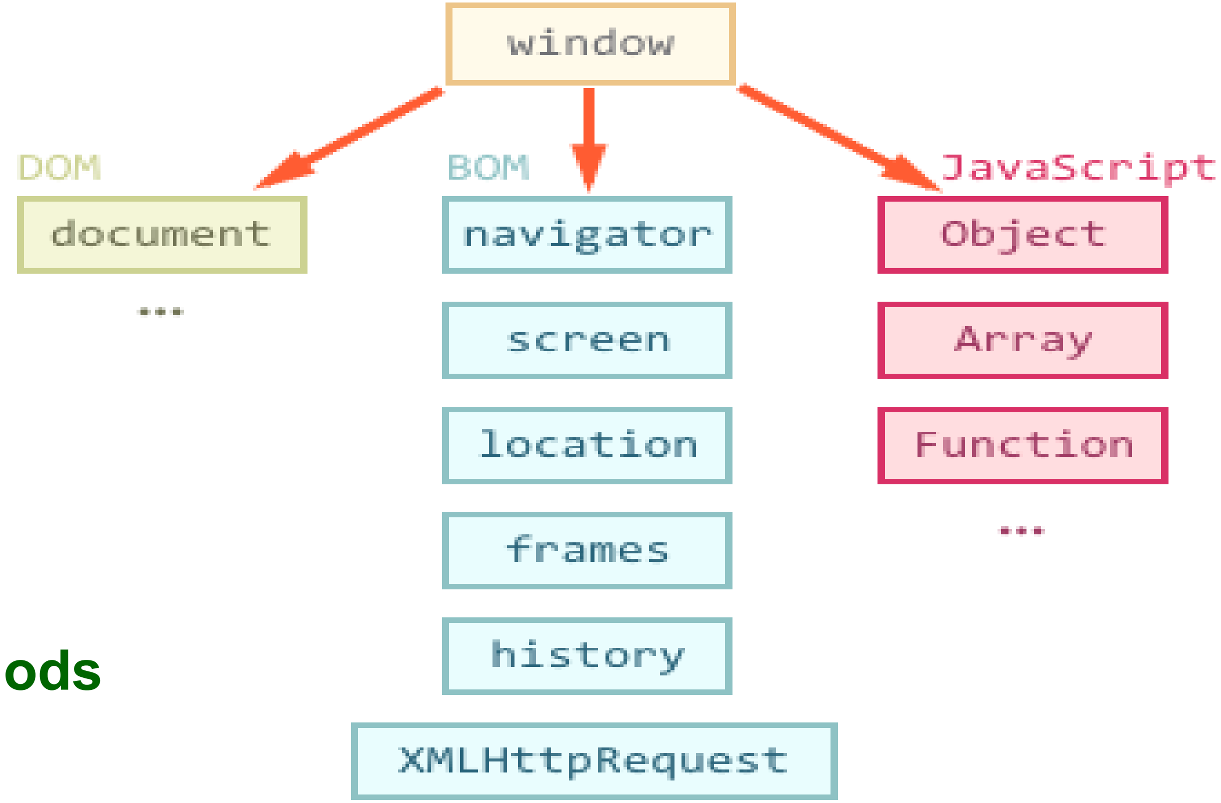
- It is THE scripting language of the Web
- It is not Java ☺
  - names: Mochan → LiveScript → JavaScript
- Running environments
  - a browser (in these slides)
  - a JavaScript runtime environment
    - most renown is Node.js, built on Chrome's V8 JavaScript engine
- Does not need any special software
  - it is just enabled within browser
- It is the building block for very popular libraries such as JQuery and D3.js

JS: History

- **1995**
  - JS was created by Brendan Eich at Netscape
- **1996**
  - Microsoft releases "JScript” for IE3
- **1997**
  - JS was standardized in the "ECMAScript" specifications
- **2010**
  - Node.js was released
- **2017**
  - ECMAScript 8 was released

ECMAScript: Language Core

- **Variables**
  - **declaration, naming conventions**
- **Data types**
  - **primitive types**
  - **special values**
  - **loose/dynamic typing**
- **Expressions**
- **Arrays**
  - **mutator, accessor, and iterator methods**
- **Objects**
- **Functions**
  - **variables scope, “window” object**
- **Call Stack**
  - **blocking calls, asynchronous calls, concurrency and event loop**



JS: Variables (until ECMAScript 5)

- Used to store values
- Declaration syntax
  - keyword “var”+ variable-name
  - no data type required
- Declare, then initialize in 2 statements

```
var x;
x = 5;
```

- Or declare and initialize in one statement

```
var y = 2;
```

- Re-assign the value later

```
var x = 5;
x = 2;
```

JS: Variables (from ECMAScript 6 on)

- Three types of variable declarations
  - **var**: only global or local (function) scope
    - assigned to object **`window`** if in global scope
    - assigned to function block otherwise
  - **let**: block scope “{ … }” and loop scope
  - **const**: for read-only variables

```
var x = 2;
// here x is 2
{
    let x = 1;
    // here x is 1
}
// here x is 2 again
```

```
var x = 5;
for(let x=0; x<10; x++){
    // do something here
}
// here x is 5 again
```

JS: Variable names

- Syntax requirements
  - begin with letters, $ or _
    - a variable named “$” has been adopted by the jQuery library as the shorthand global namespace reference
  - only contain letters, numbers, $ and _
  - case sensitive
  - avoid reserved words
    - e.g.: break, const, if, typeof, etc
- Best practices
  - choose clarity and meaning
  - prefer camelCase for multipleWords (instead of under_score)

JS: Variable scopes

- A variable with “local/function” scope:

```
function addNumbers(num1, num2) {
  var localResult = num1 + num2;
  console.log("Local result is: " + localResult);
}
addNumbers(5, 7);
console.log(localResult); ➔ ReferenceError: localResult is not defined
```

- A variable with “global/program” scope:

```
var globalResult;
function addNumbers(num1, num2) {
  globalResult = num1 + num2;
  console.log("Global result is: " + globalResult);
}
addNumbers(5, 7);
console.log(globalResult);
```

JS: Warning

- In the browser the global scope is the **`window`** object
  - all global variables belong to the **`window`** object (e.g., **`window.globalResult`**)
- If you assign a value to a variable that has not been declared, it will automatically become a global variable
  - the usage of undeclared variables is discouraged

```
// carName not defined
function myFunction() {
  carName = "Punto";
}
myFunction();
```

=

```
var carName;
function myFunction() {
  carName = "Punto";
}
myFunction();
```

JS: Expressions

- Variables can also store the result of any expression

```
var x = 28 + 38;
var hello = "Hello ";
var world = "World”;
var greeting = hello + world;
```

- Variables can even store input from users using the prompt function

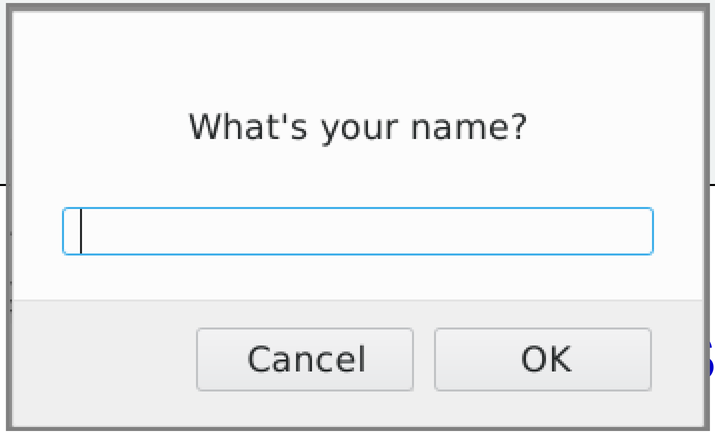
```
var name = prompt("What's your name?");
console.log('Hello ' + name);
```

JS: Expressions

- Variables can also store the result of any expression

```
var x = 28 + 38;
var hello = "Hello ";
var world = "World”;
var greeting = hello +
```

- Variables can even sers using the prompt function


What's your name?

```
var name = prompt("What's your name?");
console.log('Hello ' + name);
```

JS: Primitive data types

- ECMAScript defines six primitive data types
  - boolean
  - number
  - string
  - null
    - the value null represents the intentional absence of any object value
    - null is the unique value of type null
  - undefined
    - a variable that has not been assigned a value has the value undefined of type undefined
  - symbol (new in ECMAScript 6)
    - can be used as a key in an object property

JS: Non-primitive data types

- Two data-types are not primitive
  - objects
  - functions
- They can be created by the user
  - we will se how to manage them in the following

JS: Loose typing

- JavaScript is dynamically typed
  - variables are not directly associated a static type
  - JS figures out the type based on the current value
  - any variable can be assigned (and reassigned) values of any type
    - the type changes as the value changes
  - **`typeof vname`** returns a string containing the current type of the variable **`vname`**
    - **`typeof`** is an operator, not a function

```
var x;
x = 2;
console.log(typeof x); // yields “number”
x = "Hello";
console.log(typeof x); // yields “string”
```

JS: Loose typing

- At any moment a variable has only one type

```
var y = 2 + " cats";
console.log(typeof y); // yields “string”
```

  - ‘2’ has been converted to string (type coercion) to perform a concatentation

JS: Primitive data types

- string
  - any immutable list of chars

```
var greeting = "Hello world!";
var show = "Breaking bad!";
```

- number
  - integer (3,-56) or floating point (5.45)

```
var myAge = 28;
var pi = 3.14;
```

- boolean
  - logical values true or false (which are constants)

```
var trueValue = true;
var falseValue = false;
```

JS: Special values

- undefined
  - a value that hasn’t been defined/declared yet

```
var notDefined;
console.log(typeof notDefinied); // yields “undefined”
```

- null
  - an explicitly empty value
  - the operator typeof returns (erroneously) “object” instead of “null”

```
var nullVariable = null;
console.log(typeof nullVariable); // yields “object”
```

- symbol
  - unique identifier created via factory method

```
    var sym = Symbol(’foo')
```

JS: Type coercion

- Type coercion is the automatic conversion of a value from one type to another type in order to perform an operation (assignment, comparison, etc)
- JavaScript has two comparison operators, one with type coercion (==) and one without it (===)

```
2 == ‘2’   // type coercion -> true
2 === ‘2’ // no type coercion -> false
```

```
var notDefined;                    // type (and value) “undefined”
var nullVariable = null;           // type “null”
notDefined == nullVariable         // type coercion -> true
notDefined === nullVariable        // no type coercion -> false
```

JS: Objects

- Objects are collections of key-value pairs of any type

```
var instructor = {
  firstName: "Giordano",
  lastName: "Da Lozzo",
  age: 34,
  fullname: function(){
        return this.firstName + " " + this.lastName
        }
};
```

- Objects are associative arrays
  - bracket notation

```
var myName = instructor[firstName];
```

  - dot notation

```
var myName = instructor.firstName;
```

JS: Objects

- Objects inherit their prototype from Object
  - loop using the keyword “in”

```
for (var prop in instructor) {
  if (instructor.hasOwnProperty(prop)) {
    console.log("property: ", prop);
    console.log("value: ", instructor[prop]);
  }
}
```

JS: Objects access

```
var aboutMe = {
  hometown: “Rome, IT",
  hair: “brown“
};
```

- Object.keys(objectName)
  - lists all the property names of objectName

```
Object.keys(aboutMe)
["hometown", "hair"]
```

- Object.values(objectName)
  - lists all the property values of objectName

```
Object.values(aboutMe)
["Rome, IT", "brown"]
```

JS: Arrays

- An array can hold many ordered values
  - the property **`length`** gives you the number of such values

```
var arrayName = [item1, item2, item3];
arrayName.length     // yields 3
```

- An array is actually a special object

```
typeof arrayName;  // yields "object"
```

JS: Looping within an array

- We can loop either by using the classical “for”

```
for (var i = 0; i < arrayName.length; i++) {
  console.log(arrayName[i]);
}
```

- Or by using the method “forEach” with a callback function
  - this callback function takes up to three variables

```
[2, 5, 8, 9].forEach(function(element, index, array) {
  console.log("Current el: ", element);
  console.log("Index of current el: ", index);
  console.log("Whole array: ", array);
});
```

JS: Array iterators

- These methods take as arguments functions to be called back while processing the array
  - arrayName.forEach
    - calls a function for each element in the array
  - arrayName.every
    - returns true if every element in this array satisfies the provided testing callback
  - arrayName.filter
    - creates a new array with all of the elements for which the provided filtering callback returns true
- Callback functions have parameters: element, index, array

JS: Heterogeneous arrays

- An array can hold values of different types

```
var arrayName = [true, "ciao", {}, 2, null, (x) => -x];

arrayName.forEach(
      function(el){
            console.log(typeof el)
      }
);
```

  - prints to the console “boolean”, “string”, “object”, “number”, “object”, “function”

JS: Array mutators (1/2)

- These methods modify the array
  - **array.pop**: remove the last element
  - **array.push**: add one or more element to end

```
var fruits = ["Banana", "Orange"];
fruits.push("Lemon");
// fruits contains ["Banana", "Orange”, "Lemon"]
```

  - **array.shift**: remove the first element
  - **array.unshift**: add one or more in front

```
var fruits = ["Banana", "Orange"];
fruits.unshift("Lemon");
// fruits contains ["Lemon", "Banana", "Orange”]
```

JS: Array mutators (2/2)

- These methods modify the array
  - **array.reverse**: reverse the order

```
var fruits = ["Banana", "Orange", "Lemon"];
fruits.reverse();
// fruits contains ["Lemon", "Orange", "Banana"]
```

  - **array.splice**: add/remove elements inside

```
var firstArray = [1,2,3,4,5];
var secondArray = firstArray.splice(1,3,7,8)
// firstArray contains [1,7,8,5]
//     (from pos=1 removed 3 items and inserted 7,8)
// secondArray contains [2,3,4]
//     (all the removed items)
```

JS: Arrays accessors

- These methods do not modify the array
  - **array.concat**: join the array with other arrays
  - **array.join**: join elements into a string

```
var fruits = ["Banana", "Orange", "Lemon"];
fruits.join(); // returns "Banana, Orange, Lemon"
```

- **array.slice**: returns a shallow copy of a portion of an array into a new array

```
[1,2,3,4,5].slice(1,3);
      // yields [2,3], first included, third excluded
```

JS: Arrays accessors

- These methods do not modify the array
  - **array.map:** creates a new array from calling a function for every array element

```
var numbers = [9, 4, 16, 25];
var newArray = numbers.map(Math.sqrt); // [3, 2, 4, 5]
```

  - **array.indexOf**: find first occurrence
  - **array.lastIndexOf**: find last occurrence

JS: Functions

- Functions are re-usable collections of statements
- First declare the function

```
function sayMyName(name) {
        console.log(name);
}
```

- Alternative declaration
  - “arrow functions” were introduced in ECMAScript 6

```
sayMyName = (name) => {
        console.log(name);
}
```

- Then call it

```
sayMyName("Valentino");
```

JS: Function parameters

- Functions can accept any number of named parameters

```
function addNumbers(num1, num2) {
  var result = num1 + num2;
  console.log(result);
}
addNumbers(7, 20); // 22
addNumbers("Hello", " Everybody"); // Hello Everybody
```

- Of course you can also pass variables and generic expressions

```
var number = 5;
addNumbers(number, 12 + 3)
```

JS: Function parameters

- Function parameters are passed by value
  - changes to the parameters are not visible outside the function
- Objects are passed by reference
  - actually, in JS object references are values
  - thus, objects will behave like they are passed by reference
    - changes to object properties are visible (reflected) outside the function

JS: Return values

- The **`return`** keyword returns a value to whoever calls the function (and then exits)

```
function addNumbers(num1, num2) {
  var result = num1 + num2;
  return result;
  // Anything after this line won't be executed
};
```

- You can call functions in expressions or inside function calls:

```
var sum = addNumbers(3, 7) + addNumbers(1, 2);
var sum2 = addNumbers(addNumbers(3, 2), addNumbers(3, 7));
```

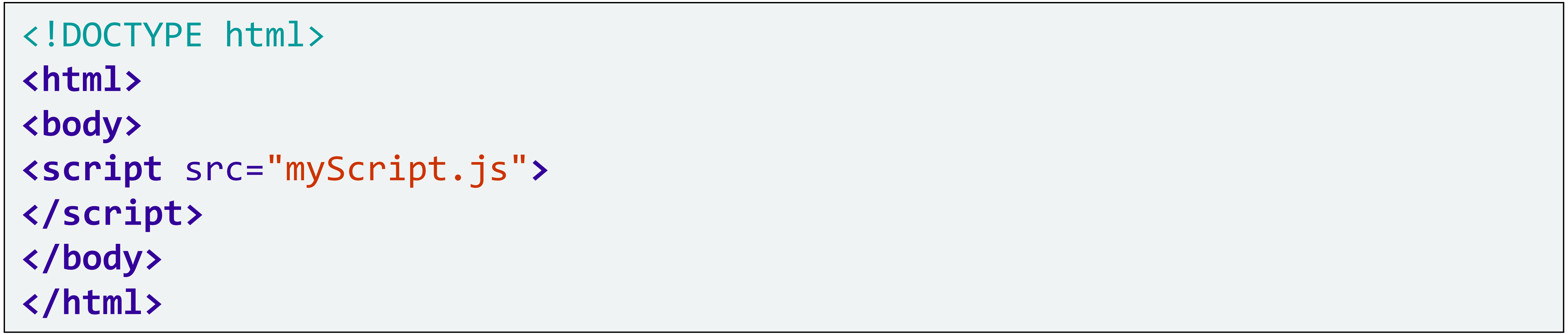
Where to place your code (1/2)

- Between the HTML tags <script> and </script>

```
<!DOCTYPE html>
<html>
<head>
<script>
function myFunction() {
  document.getElementById("demo").innerHTML = “Paragraph changed.";
}
</script>
</head>
<body>
<h1>My Web Page</h1>
<p id="demo">A paragraph</p>
<button type="button" onclick="myFunction()">Try it</button>
</body>
</html>
```

Where to place your code (2/2)

- Normally, scripts are placed in external files with .js extension



```
<!DOCTYPE html>
<html>
<body>
<script src="myScript.js">
</script>
</body>
</html>
```

JS: Call stack

- JS is single-threaded
  - i.e.: one call stack = one thing at a time
- The call stack stores the active functions
- The function at the top of the stack is the one that is executed
  - when we step into a function, we push its call on top of the stack
  - when we return from a function (or when it ends), we pop the stack

JS: Call stack

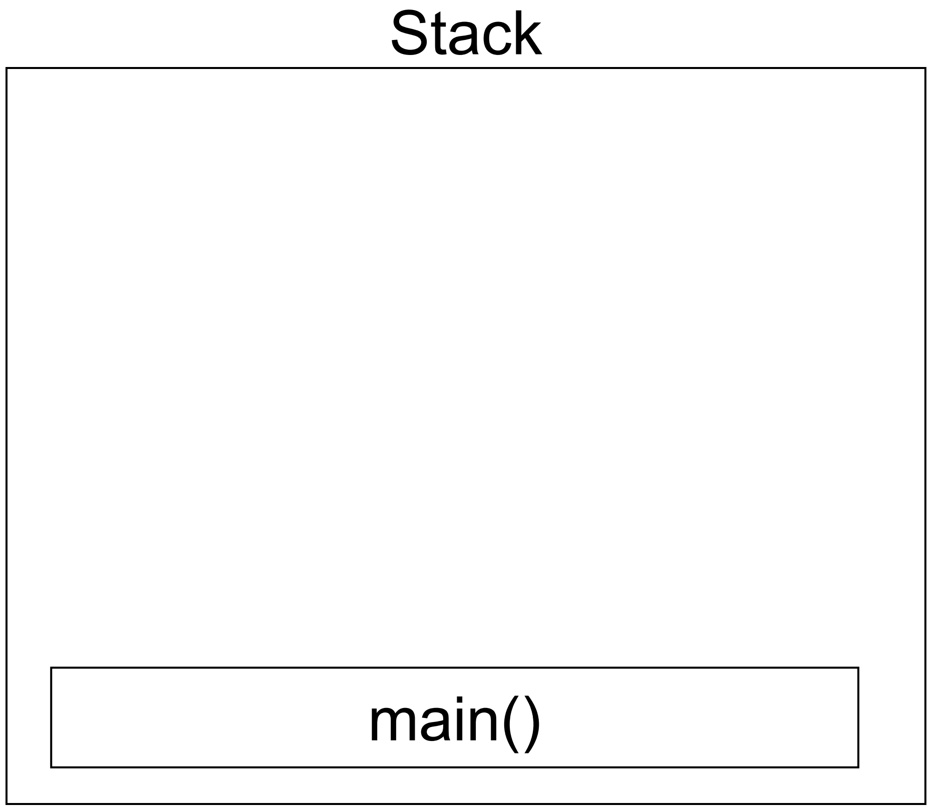
- JS is single-threaded
  - i.e.: one call stack = one thing at a time

```
function multiply(a, b) {
  return a * b;
};

function square(n) {
  return multiply(n, n);
};

function printSquare(n) {
  var squared = square(n);
  console.log(squared);
};

printSquare(4);
```



JS: Call stack

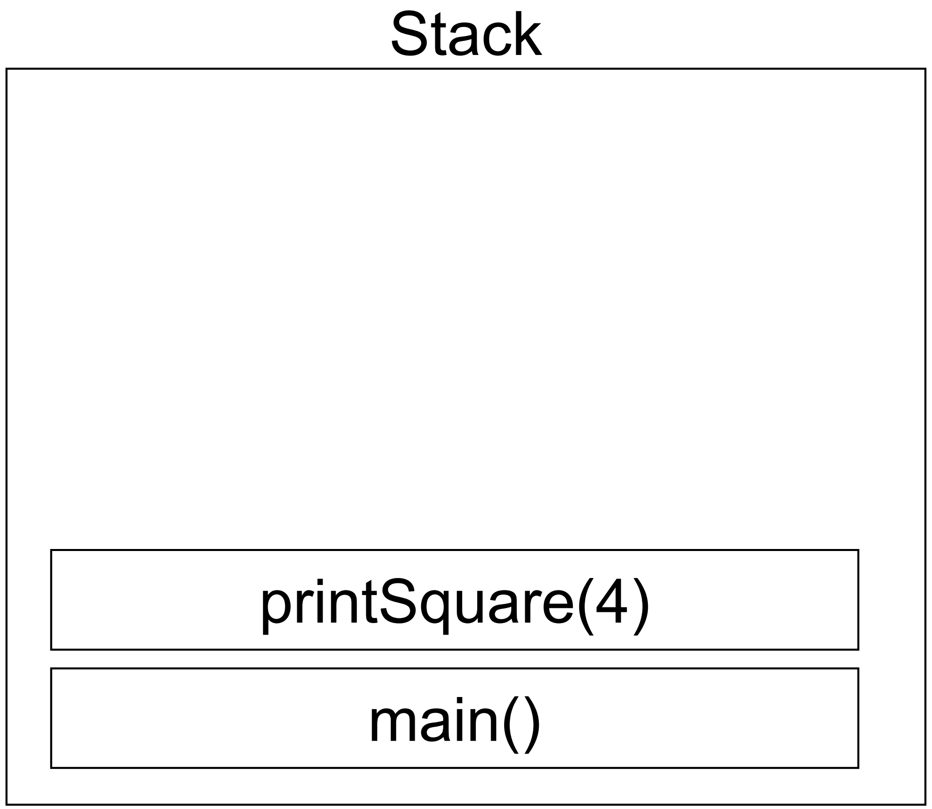
- JS is single-threaded
  - i.e.: one call stack = one thing at a time

```
function multiply(a, b) {
  return a * b;
};

function square(n) {
  return multiply(n, n);
};

function printSquare(n) {
  var squared = square(n);
  console.log(squared);
};

printSquare(4);
```



JS: Call stack

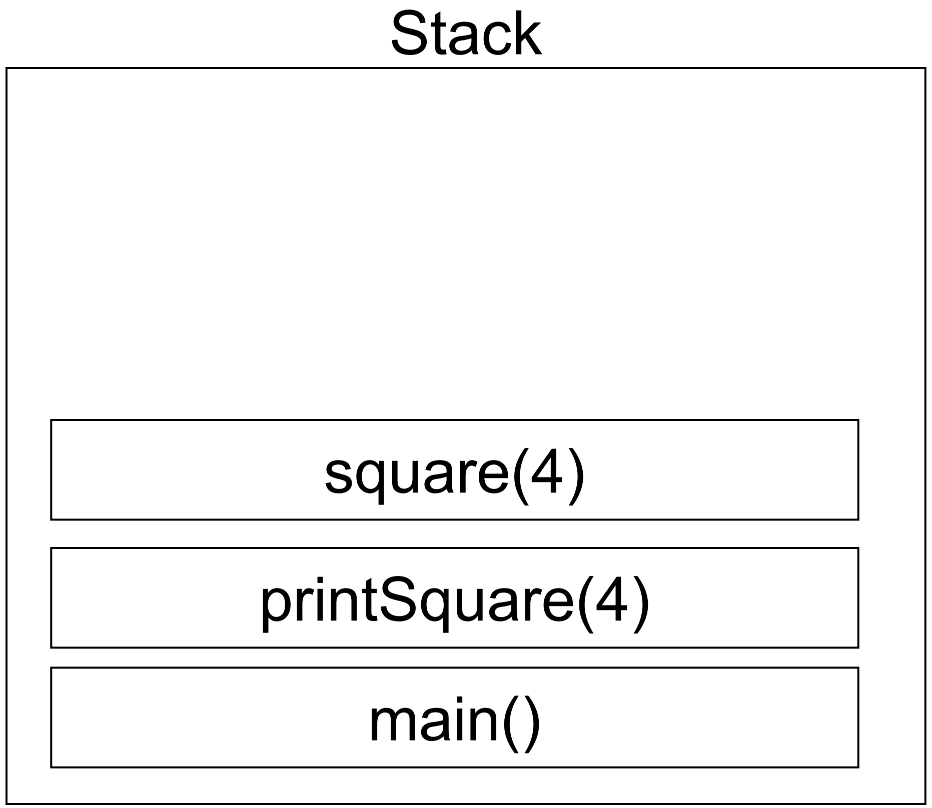
- JS is single-threaded
  - i.e.: one call stack = one thing at a time

```
function multiply(a, b) {
  return a * b;
};

function square(n) {
  return multiply(n, n);
};

function printSquare(n) {
  var squared = square(n);
  console.log(squared);
};

printSquare(4);
```



JS: Call stack

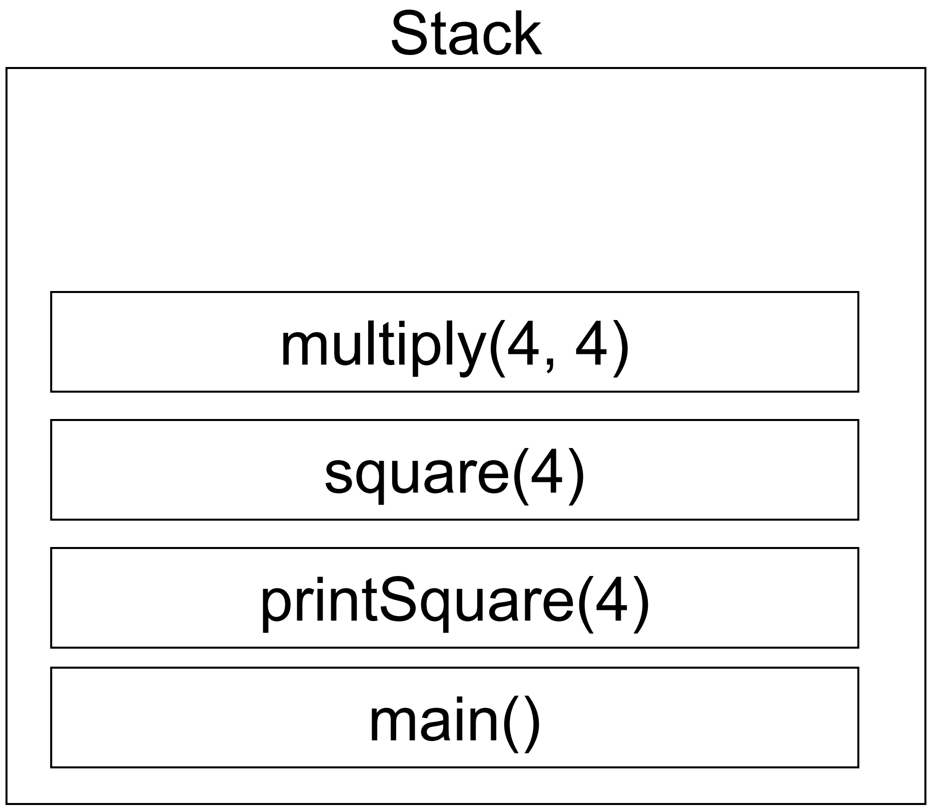
- JS is single-threaded
  - i.e.: one call stack = one thing at a time

```
function multiply(a, b) {
  return a * b;
};

function square(n) {
  return multiply(n, n);
};

function printSquare(n) {
  var squared = square(n);
  console.log(squared);
};

printSquare(4);
```



JS: Call stack

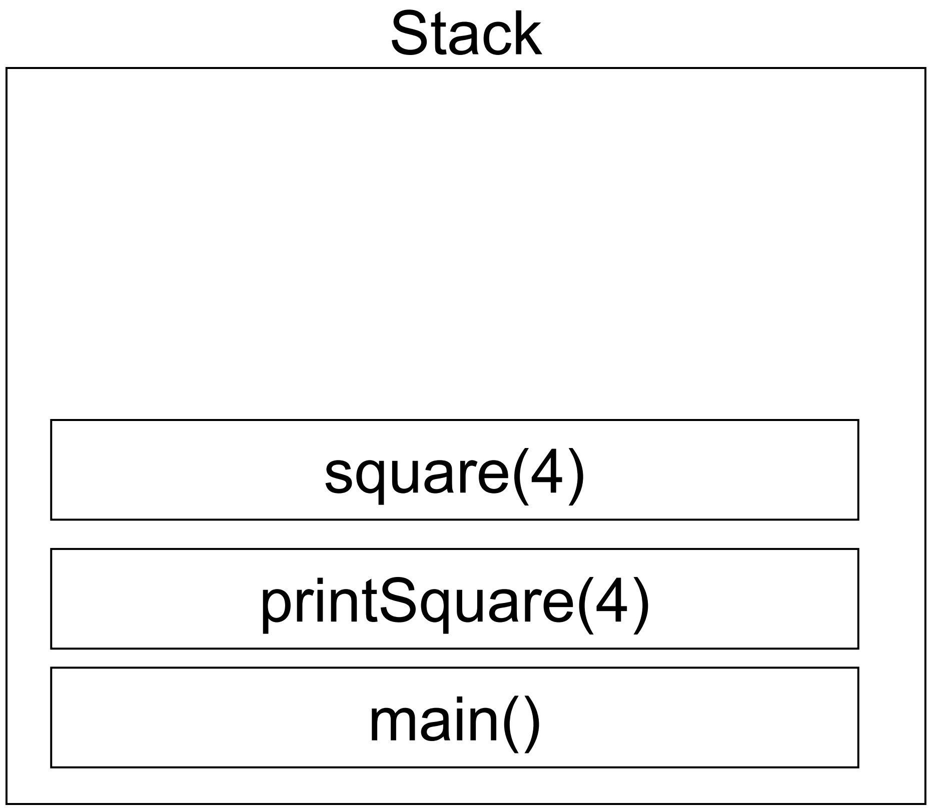
- JS is single-threaded
  - i.e.: one call stack = one thing at a time

```
function multiply(a, b) {
  return a * b;
};

function square(n) {
  return multiply(n, n);
};

function printSquare(n) {
  var squared = square(n);
  console.log(squared);
};

printSquare(4);
```



JS: Call stack

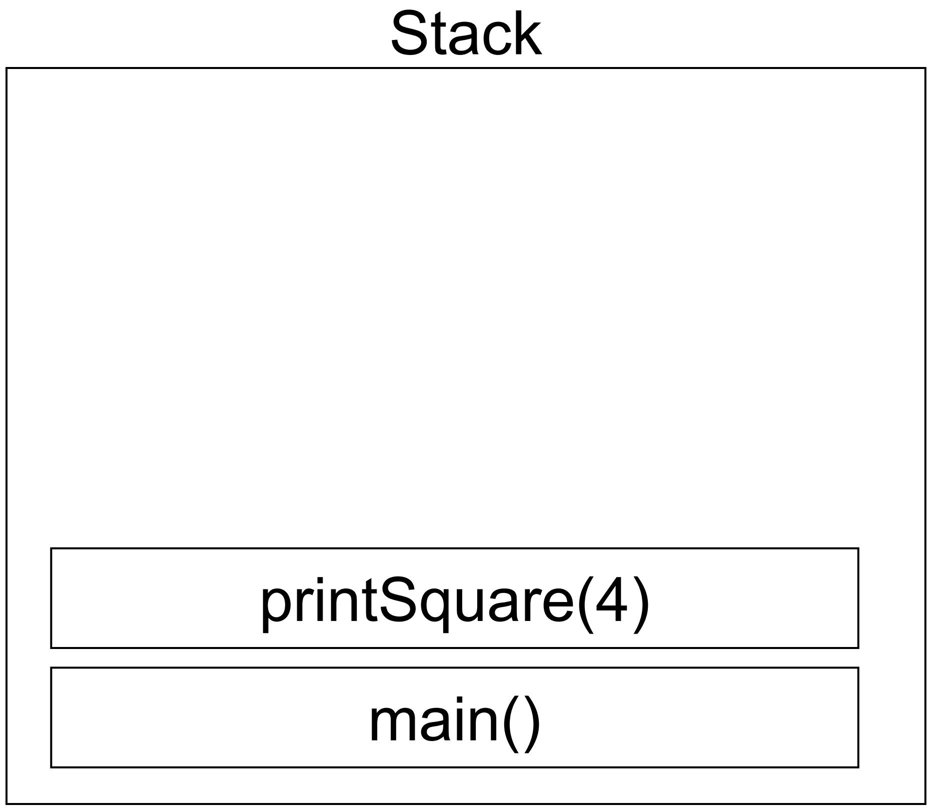
- JS is single-threaded
  - i.e.: one call stack = one thing at a time

```
function multiply(a, b) {
  return a * b;
};

function square(n) {
  return multiply(n, n);
};

function printSquare(n) {
  var squared = square(n);
  console.log(squared);
};

printSquare(4);
```



Stack

printSquare(4)

main()

JS: Call stack

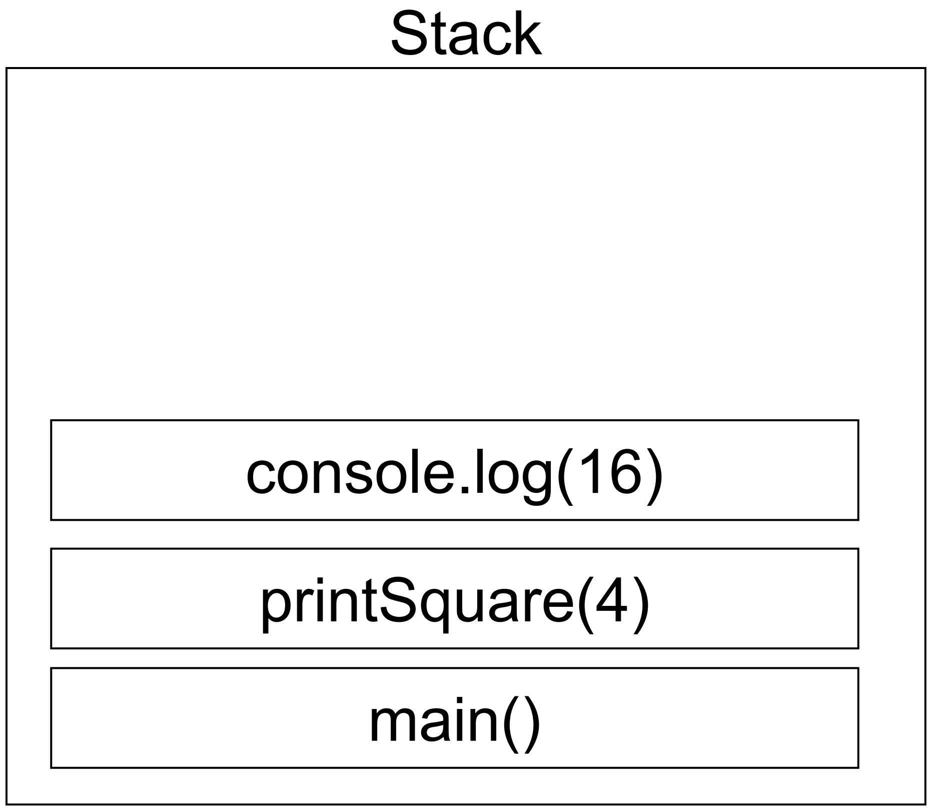
- JS is single-threaded
  - i.e.: one call stack = one thing at a time

```
function multiply(a, b) {
  return a * b;
};

function square(n) {
  return multiply(n, n);
};

function printSquare(n) {
  var squared = square(n);
  console.log(squared);
};

printSquare(4);
```



JS: Call stack

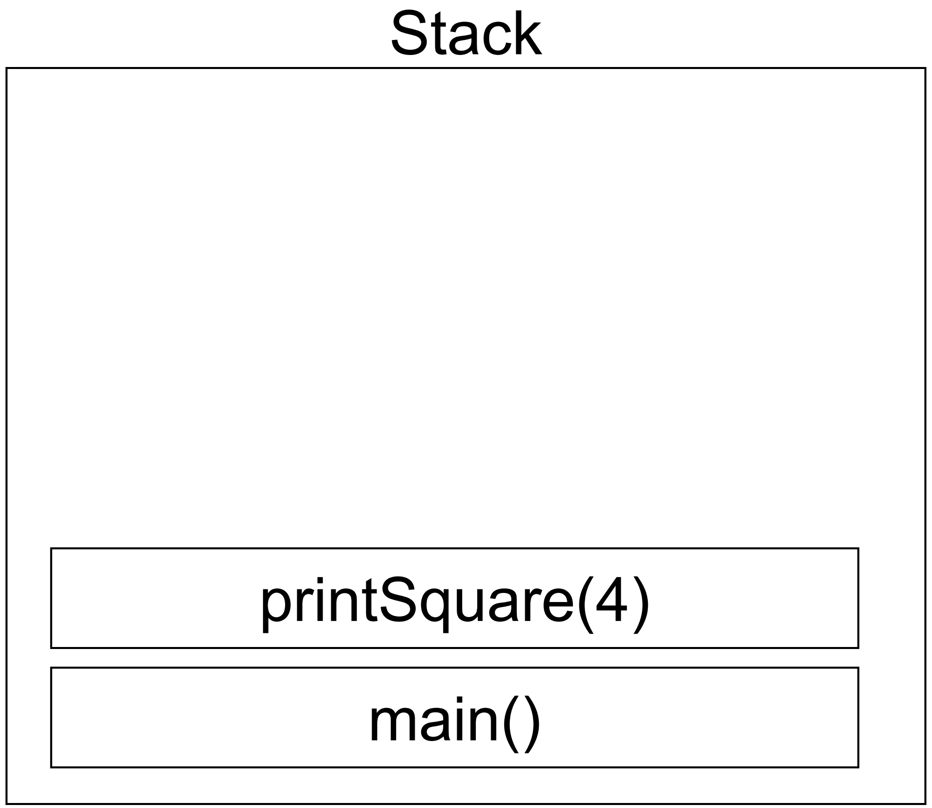
- JS is single-threaded
  - i.e.: one call stack = one thing at a time

```
function multiply(a, b) {
  return a * b;
};

function square(n) {
  return multiply(n, n);
};

function printSquare(n) {
  var squared = square(n);
  console.log(squared);
};

printSquare(4);
```



Stack

printSquare(4)

main()

JS: Call stack

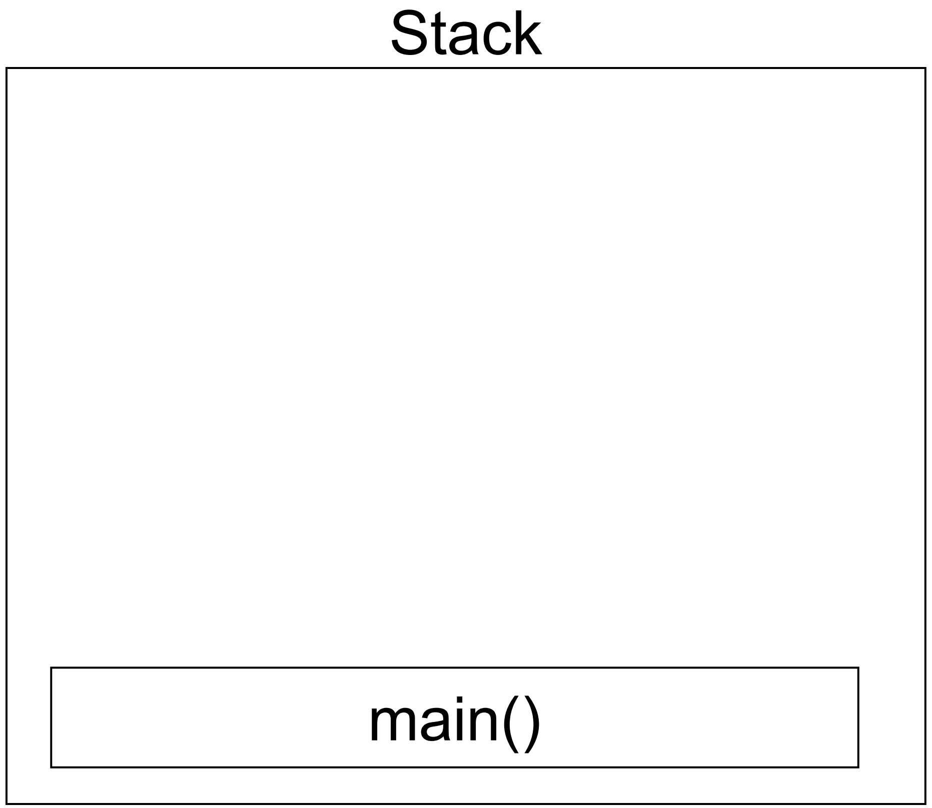
- JS is single-threaded
  - i.e.: one call stack = one thing at a time

```
function multiply(a, b) {
  return a * b;
};

function square(n) {
  return multiply(n, n);
};

function printSquare(n) {
  var squared = square(n);
  console.log(squared);
};

printSquare(4);
```



JS: Call stack

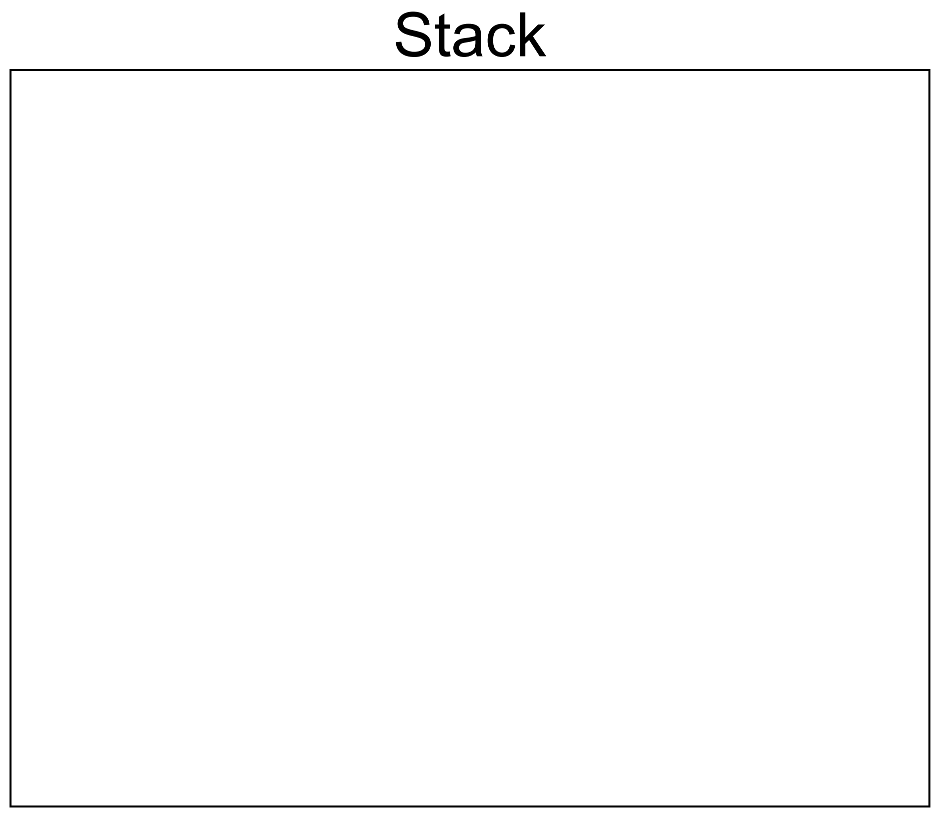
- JS is single-threaded
  - i.e.: one call stack = one thing at a time

```
function multiply(a, b) {
  return a * b;
};

function square(n) {
  return multiply(n, n);
};

function printSquare(n) {
  var squared = square(n);
  console.log(squared);
};

printSquare(4);
```



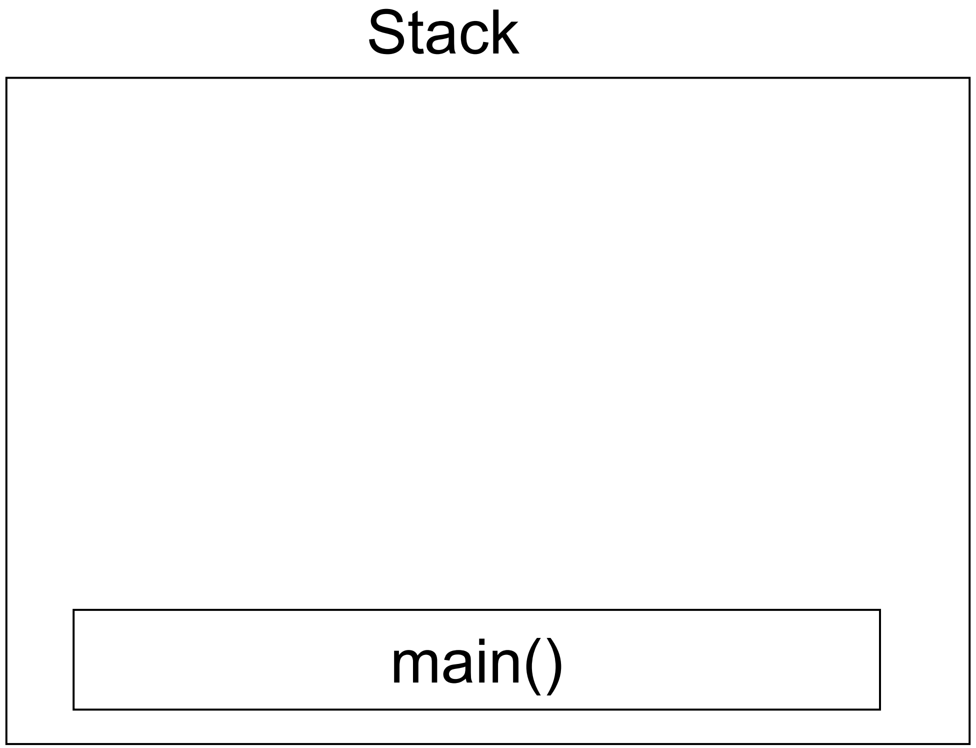
JS: Asynchronous callbacks

- JavaScript is asynchronous
  - events can happen outside of the main flow of your program

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```



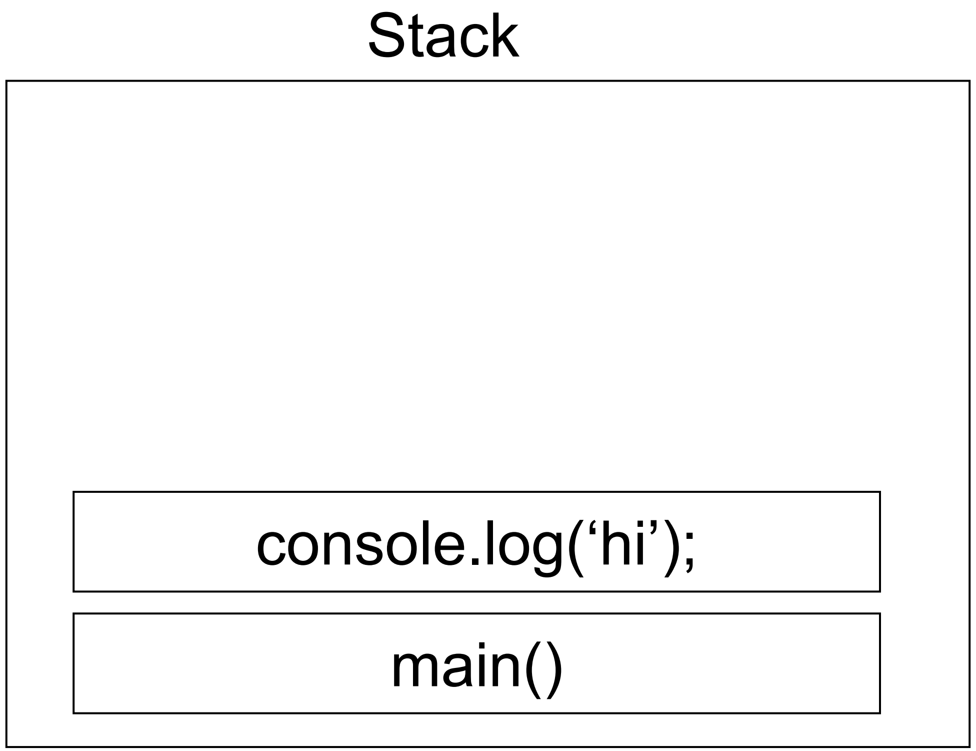
JS: Asynchronous callbacks

- JavaScript is asynchronous
  - events can happen outside of the main flow of your program

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```



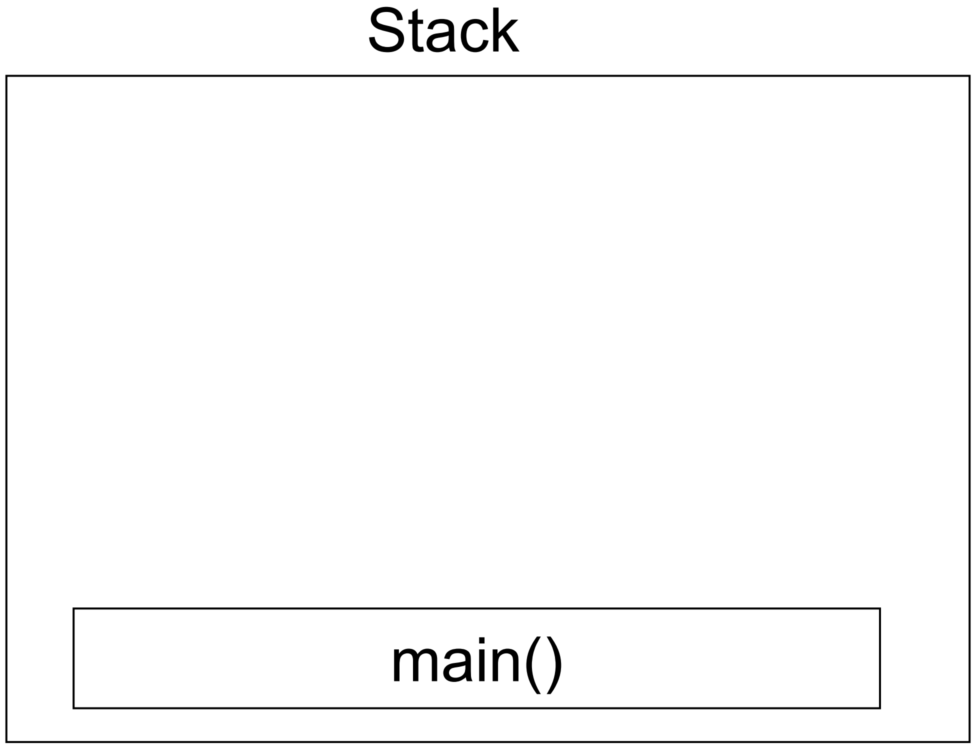
JS: Asynchronous callbacks

- JavaScript is asynchronous
  - events can happen outside of the main flow of your program

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```



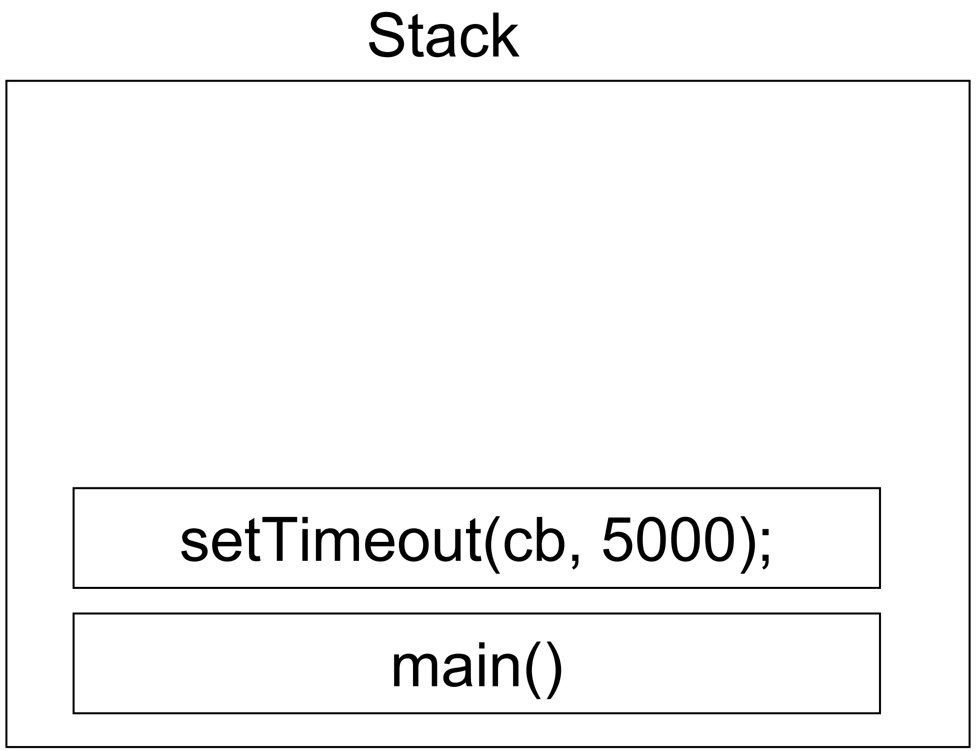
JS: Asynchronous callbacks

- JavaScript is asynchronous
  - events can happen outside of the main flow of your program

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```



JS: Asynchronous callbacks

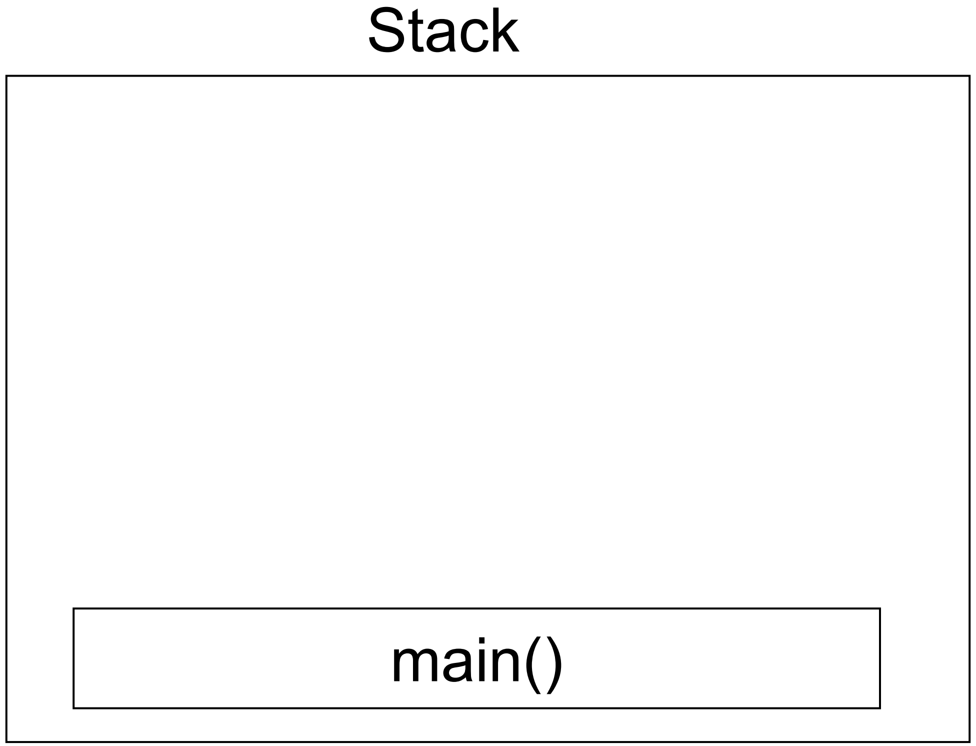
- JavaScript is asynchronous
  - events can happen outside of the main flow of your program

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```

setTimeout disappears from the stack!



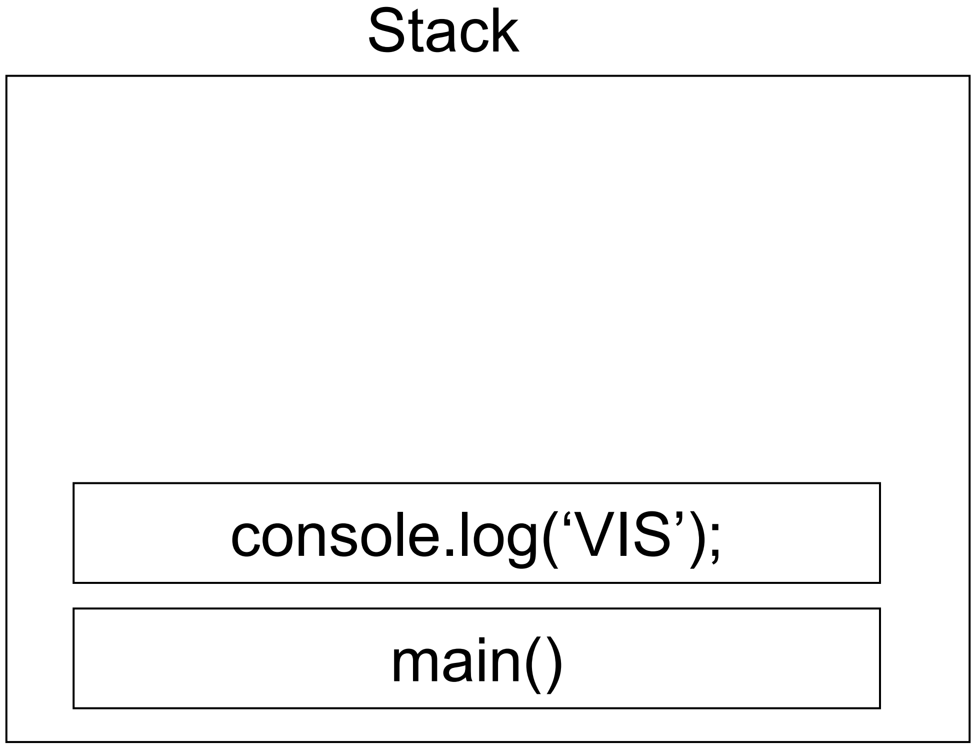
JS: Asynchronous callbacks

- JavaScript is asynchronous
  - events can happen outside of the main flow of your program

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```



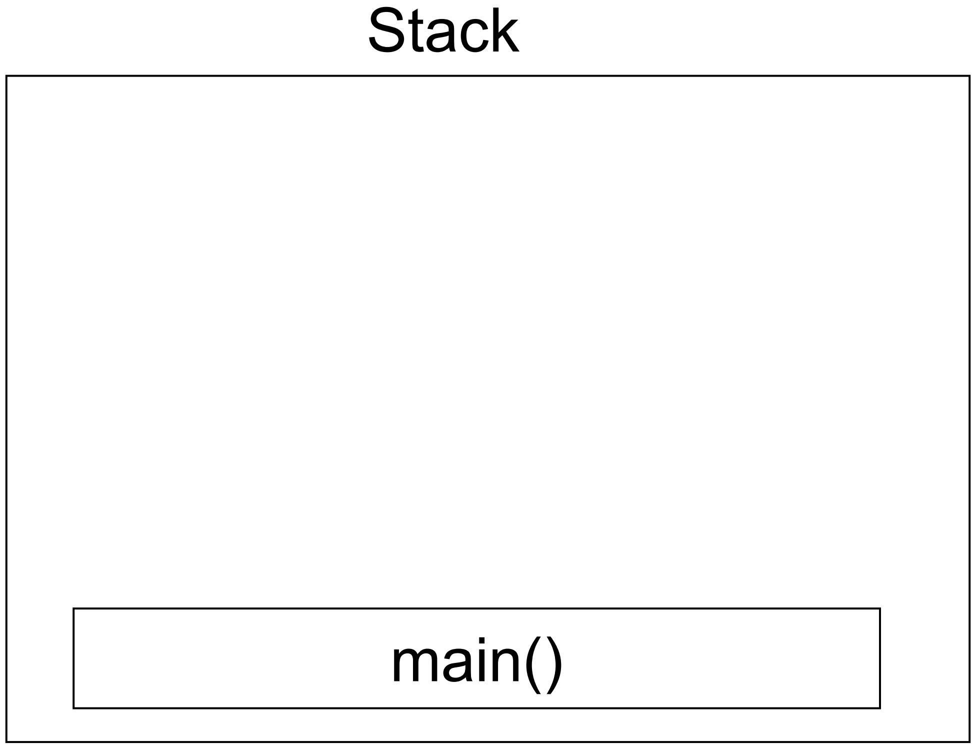
JS: Asynchronous callbacks

- JavaScript is asynchronous
  - events can happen outside of the main flow of your program

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```



Stack

main()

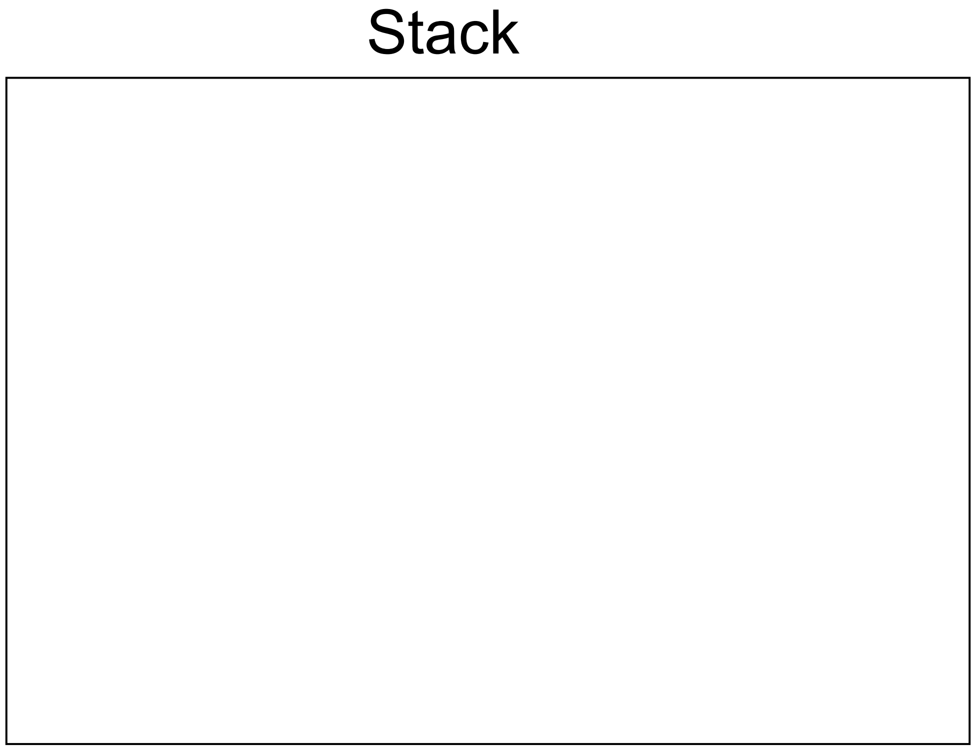
JS: Asynchronous callbacks

- JavaScript is asynchronous
  - events can happen outside of the main flow of your program

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```



JS: Asynchronous callbacks

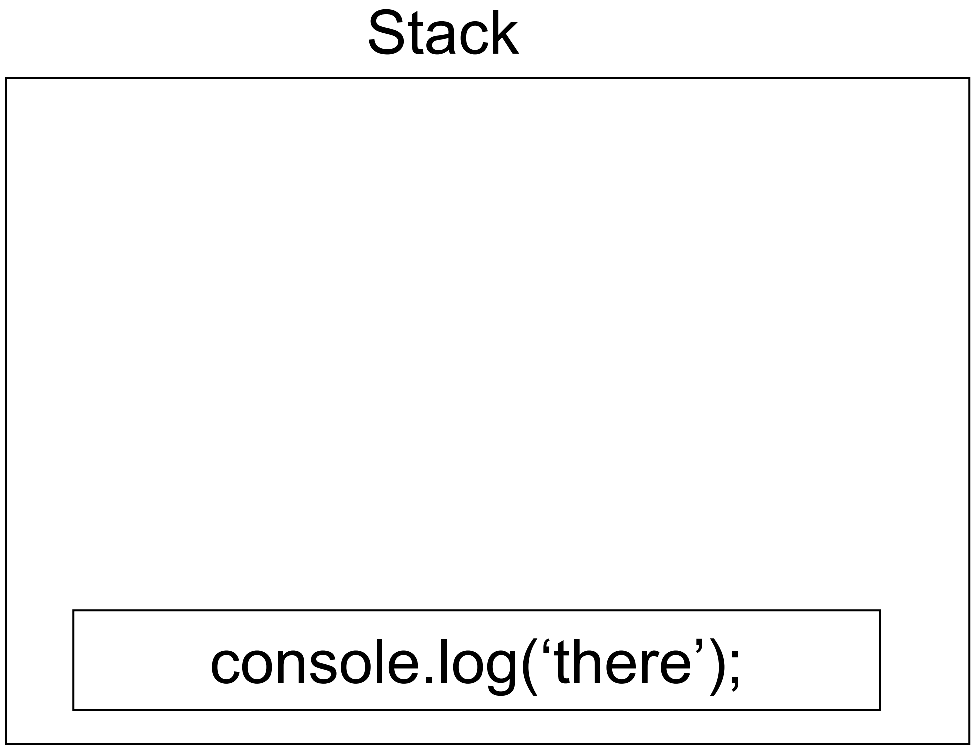
- JavaScript is asynchronous
  - events can happen outside of the main flow of your program

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```

…5 seconds later…



Concurrency and the event loop

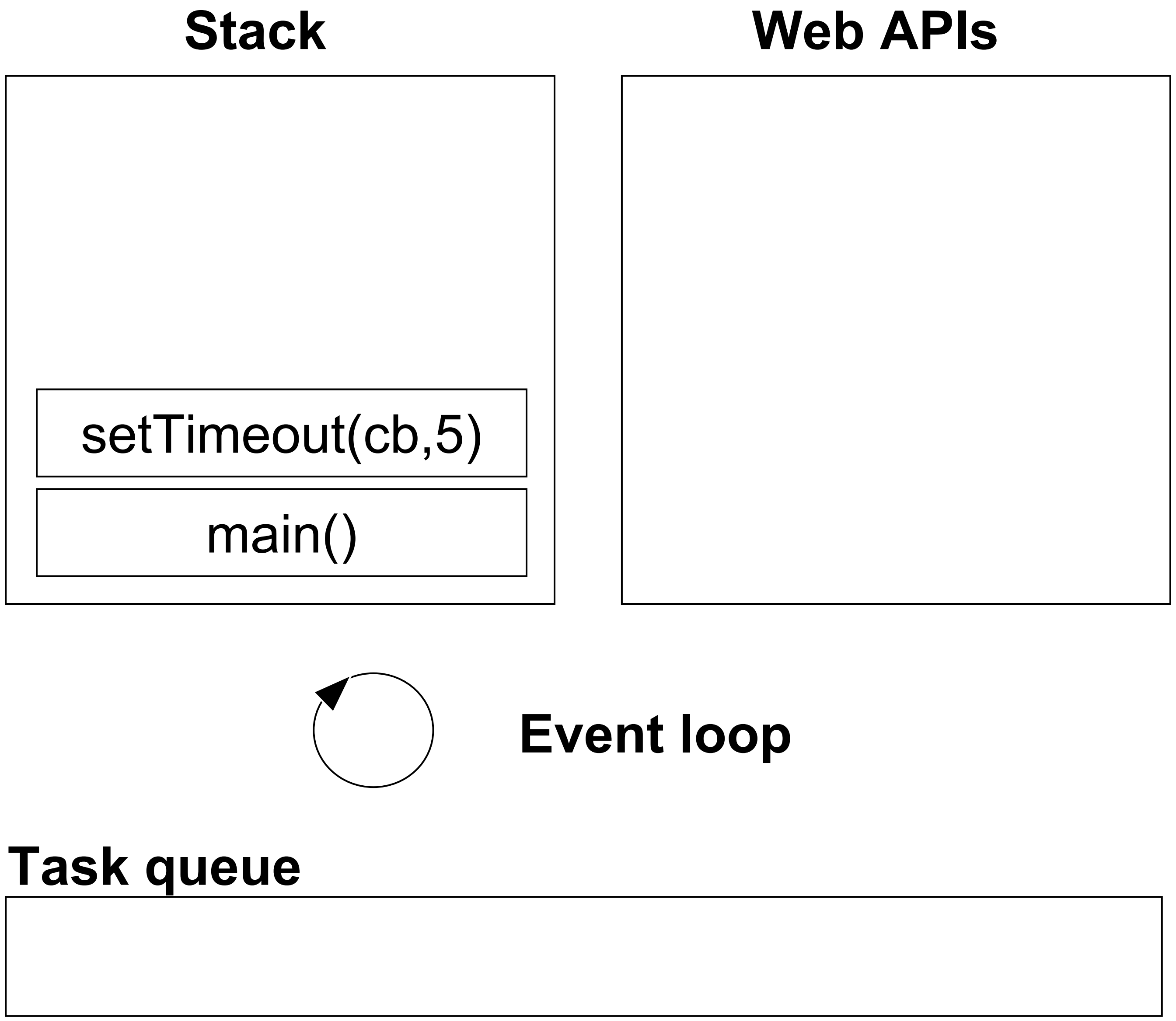
- One thing at a time, except not really!
  - the runtime environment (= stack + heap) can only do one thing at a time
  - the reason we can do things concurrently is that the browser is more than just the runtime environment
    - the Web APIs of the browser (or C++ APIs in Node.js) allow us to do more things…

JS: Async calls & Stack

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```



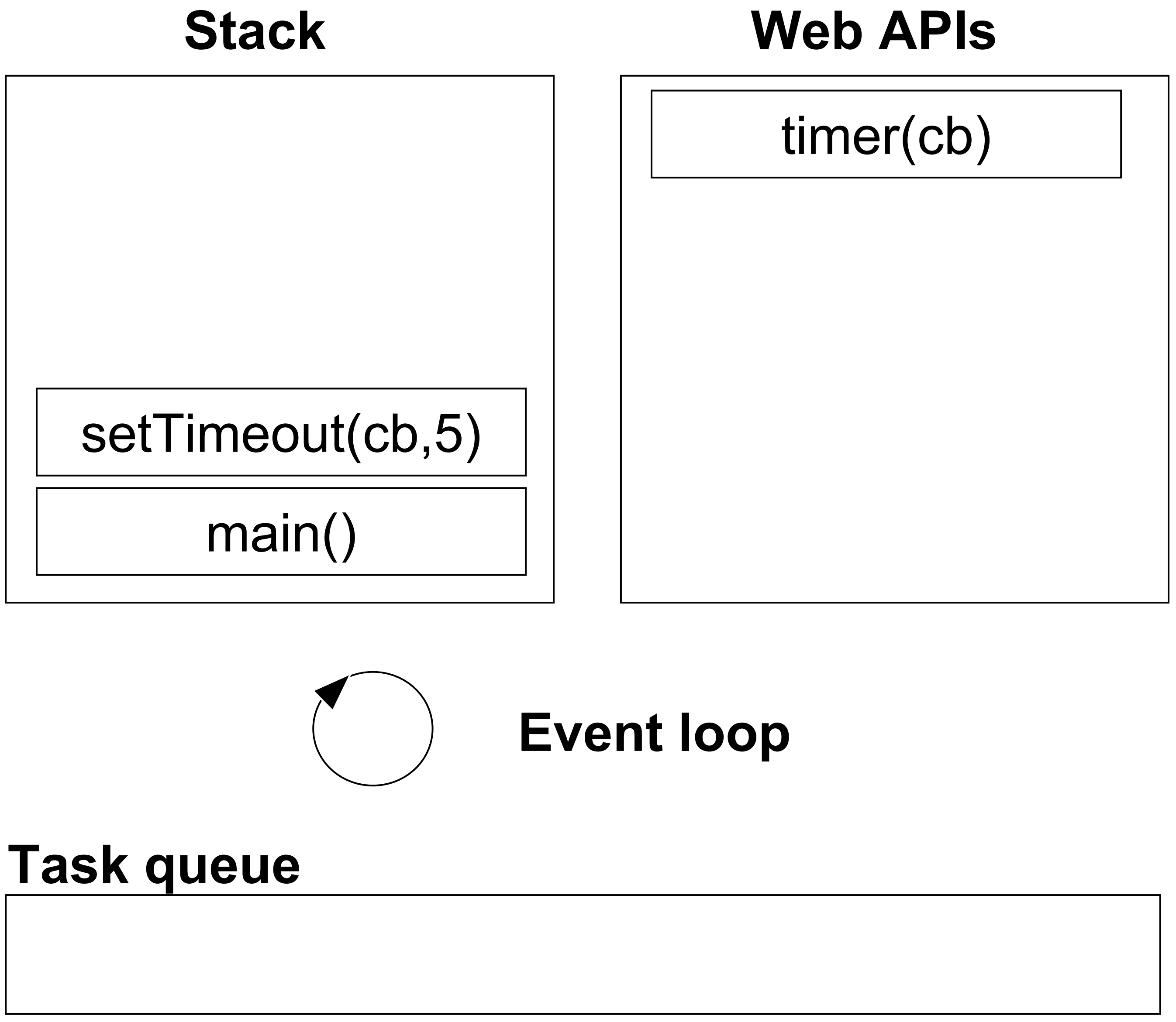
JS: Async calls & Stack

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```

- setTimeout disappears from the stack!



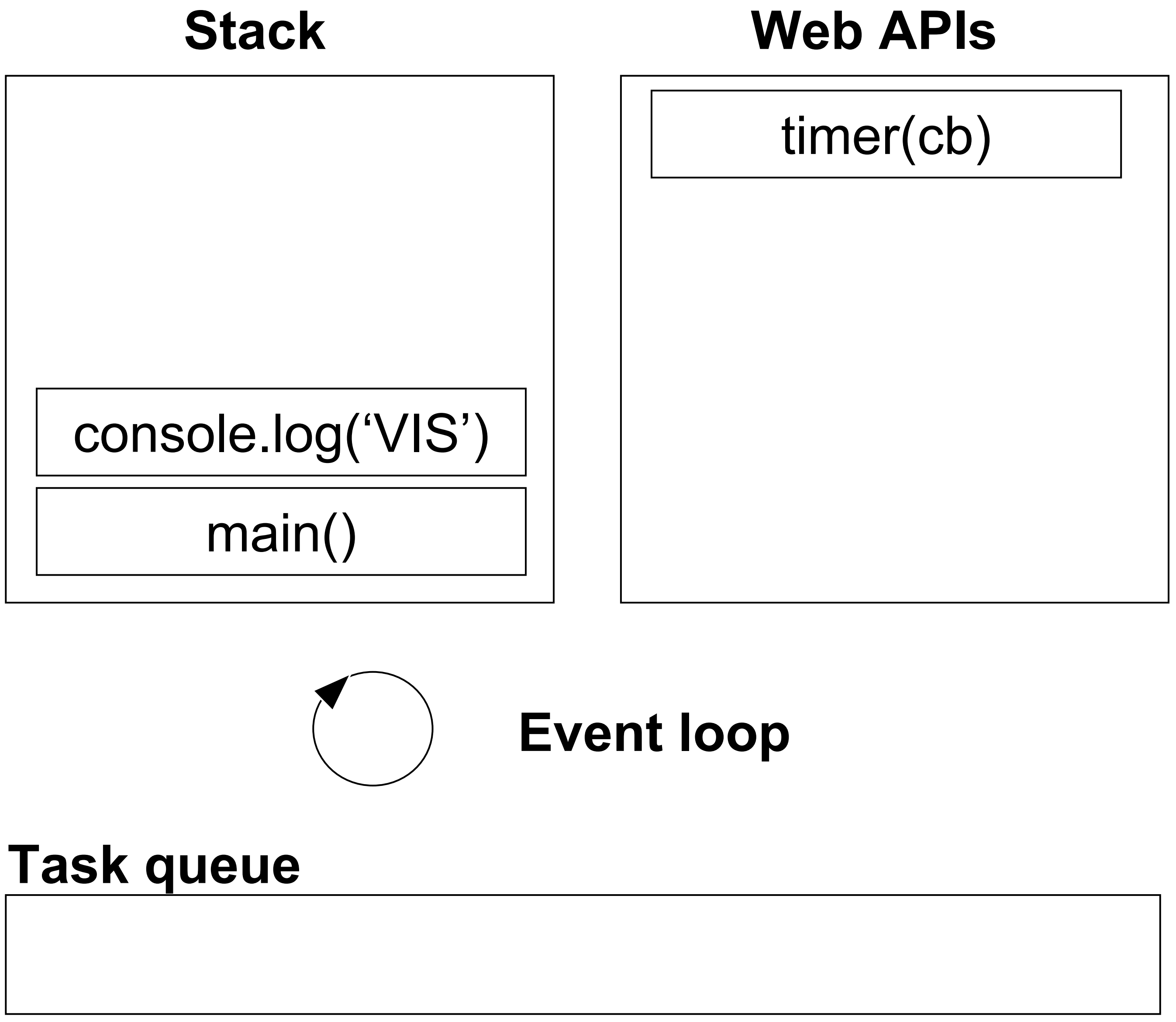
JS: Async calls & Stack

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```

- setTimeout disappears from the stack!

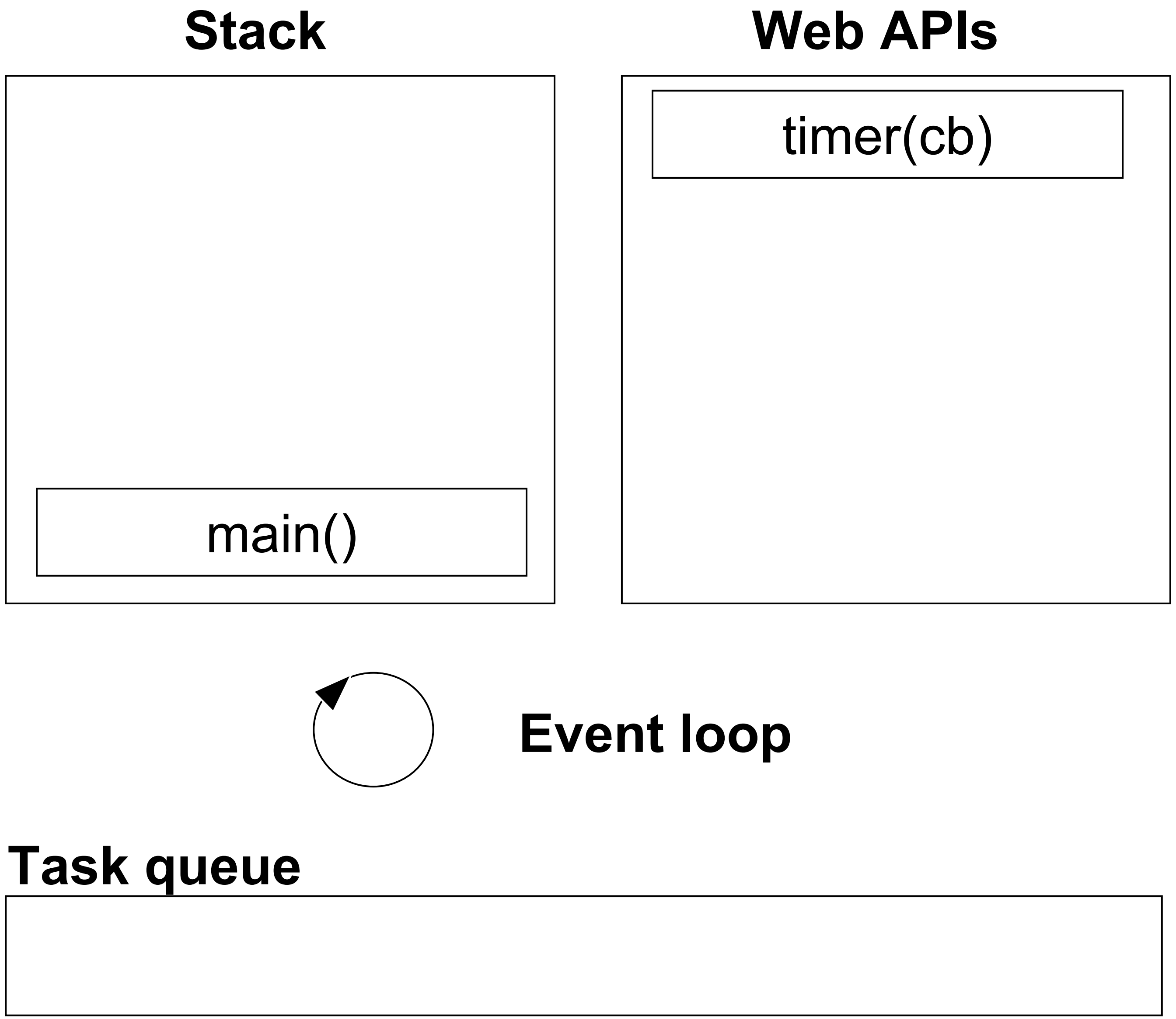


JS: Async calls & Stack

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```

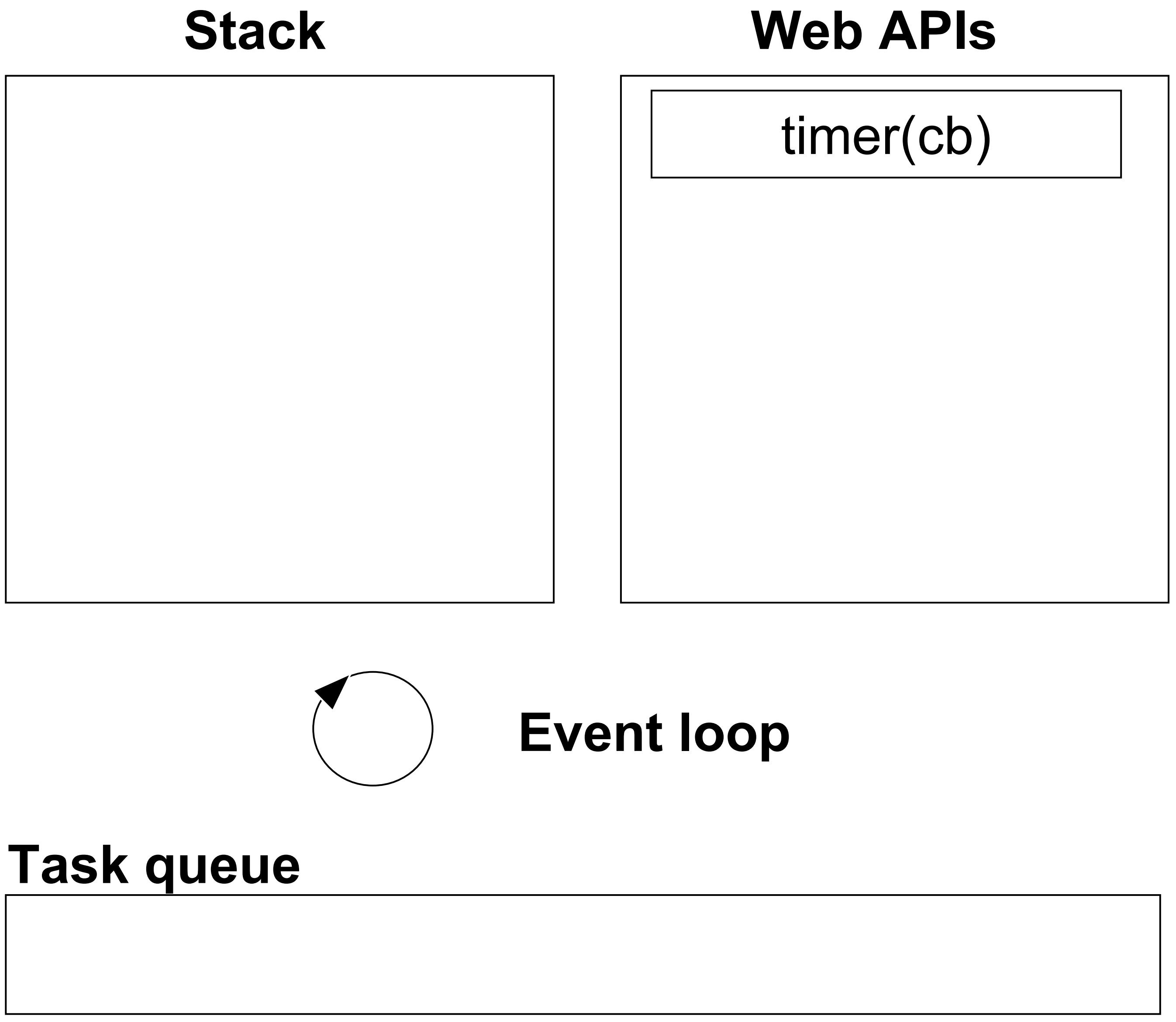


JS: Async calls & Stack

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```



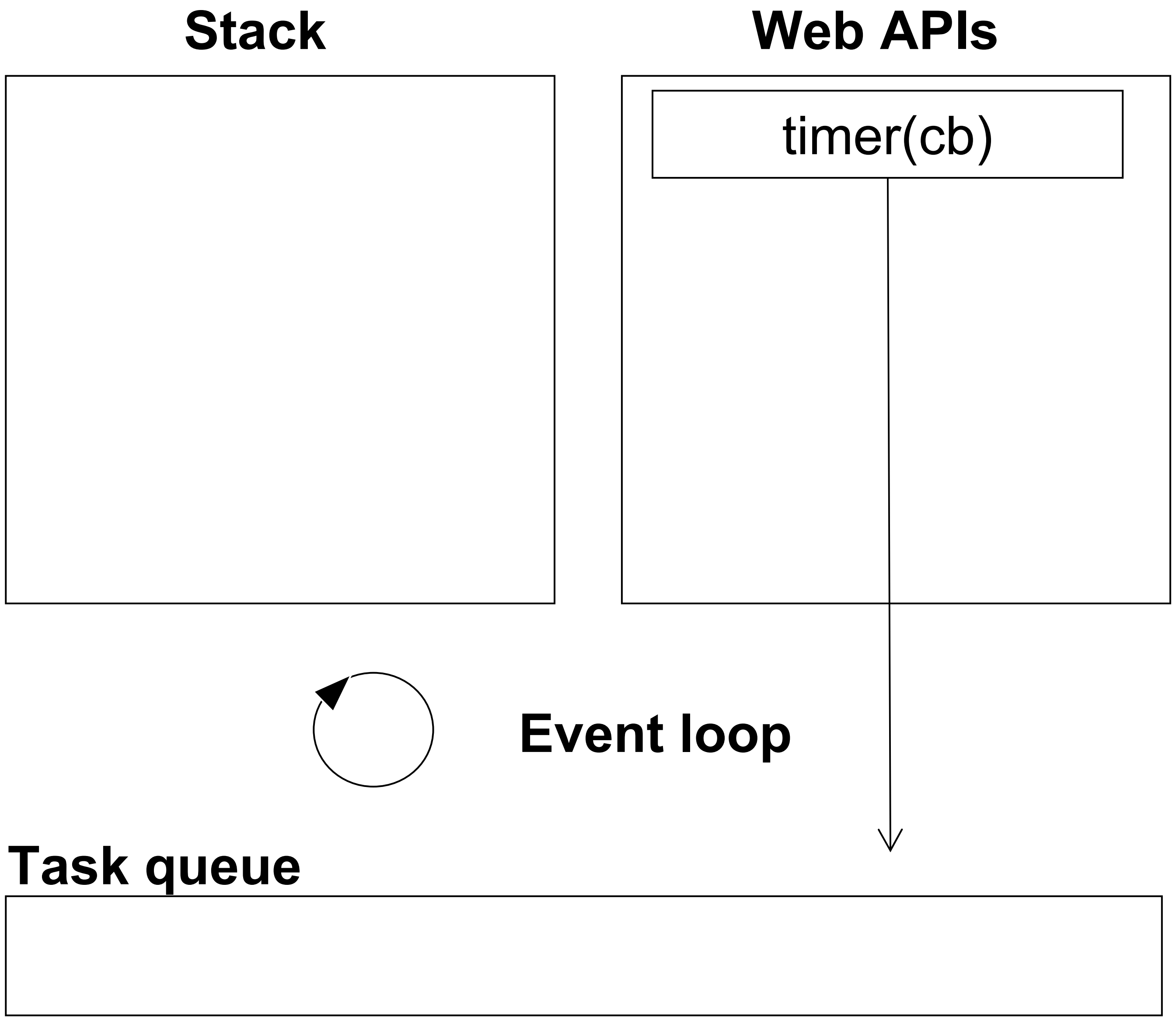
JS: Async calls & Stack

```
console.log('hi');

setTimeout(function() {
  console.log('there')
}, 5000);

console.log('VIS');
```

- 5 seconds later…
- …when the Web APIs
  are done,
  they push your callbacks
  onto
  the task queue



Stack

Web APIs

timer(cb)

Event loop

Task queue

JS: Async calls & Stack

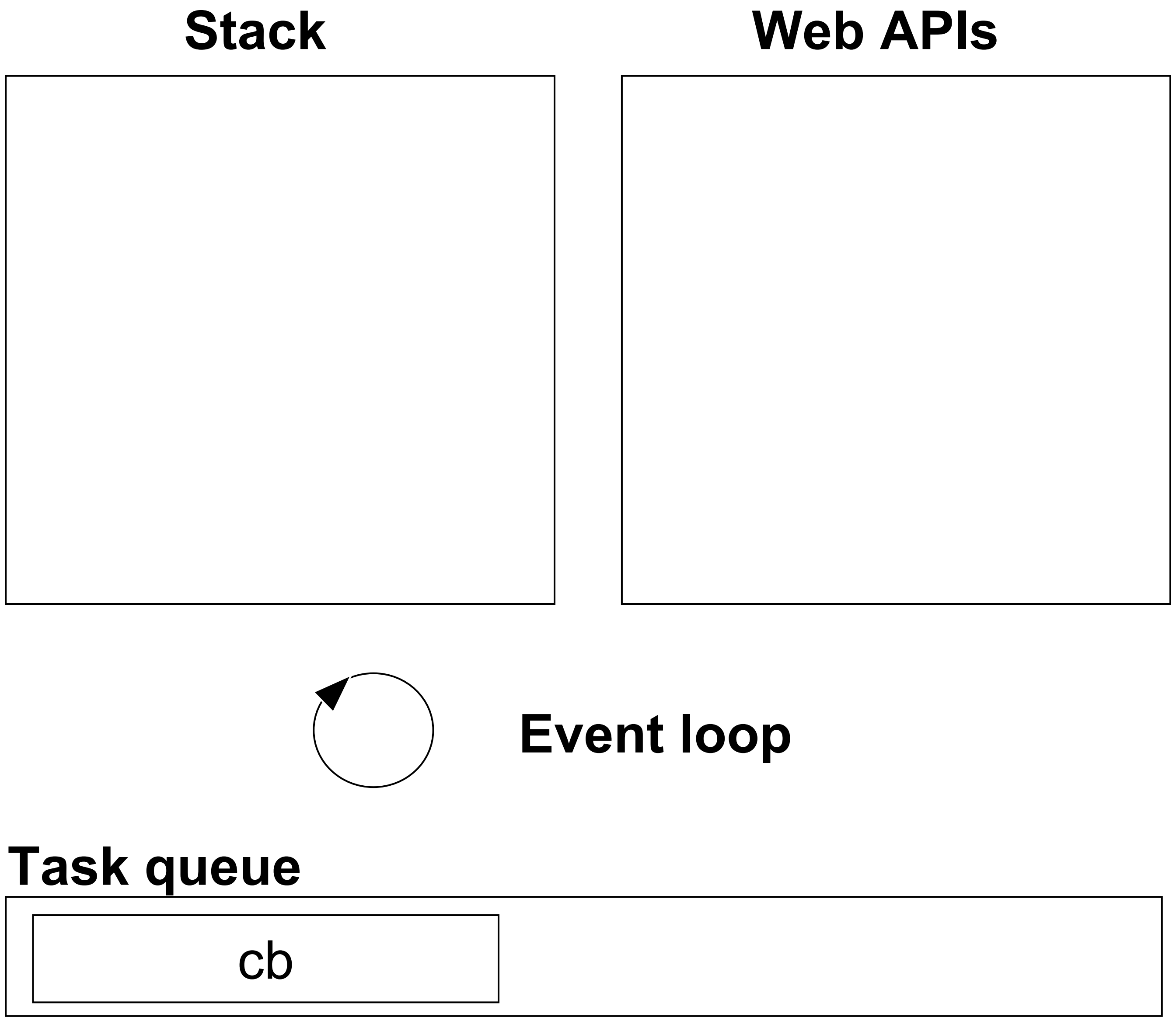
```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```

- 5 seconds later…

- …when the Web APIs
  are done,
  they push your callbacks
  onto
  the task queue



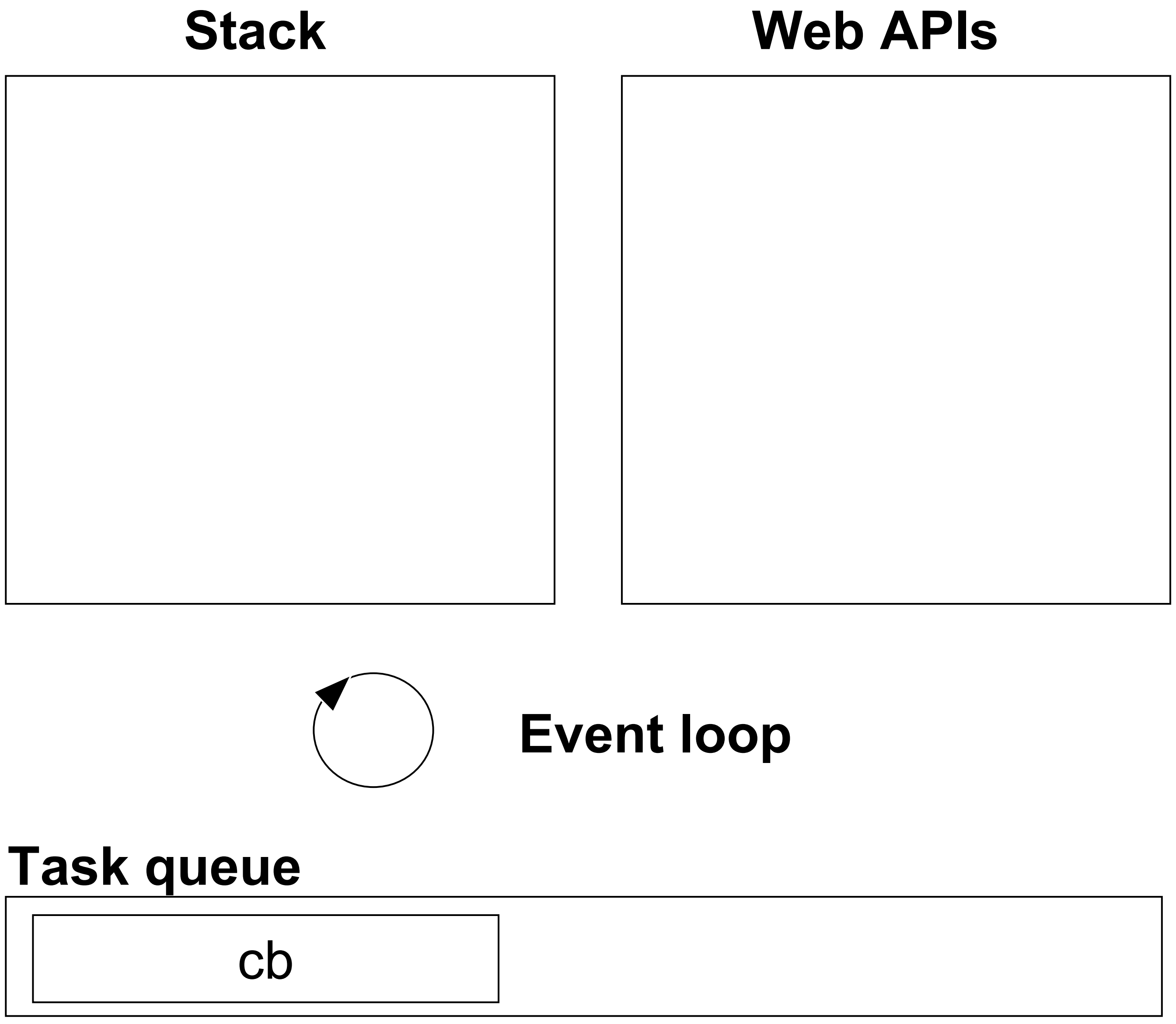
JS: Async calls & Stack

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```

- The event loop’s job is to constantly monitor the stack and the task queue
  - if the stack is empty, it takes the first thing on the queue, and pushes it on to the stack, where it is executed



Stack

Web APIs

Event loop

Task queue

cb

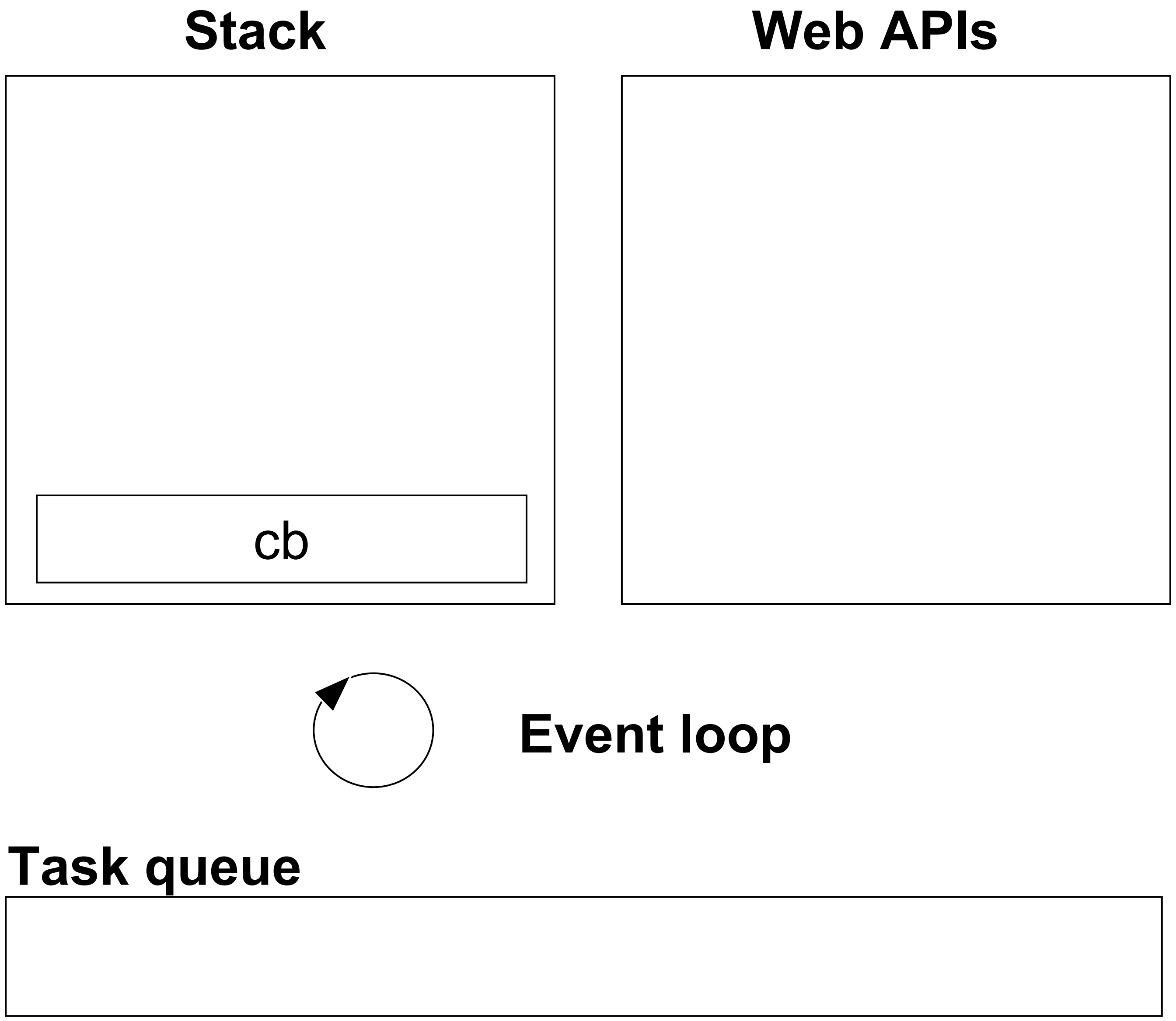
JS: Async calls & Stack

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```

- The event loop’s job is to constantly monitor the stack and the task queue
  - if the stack is empty, it takes the first thing on the queue, and pushes it on to the stack, where it is executed

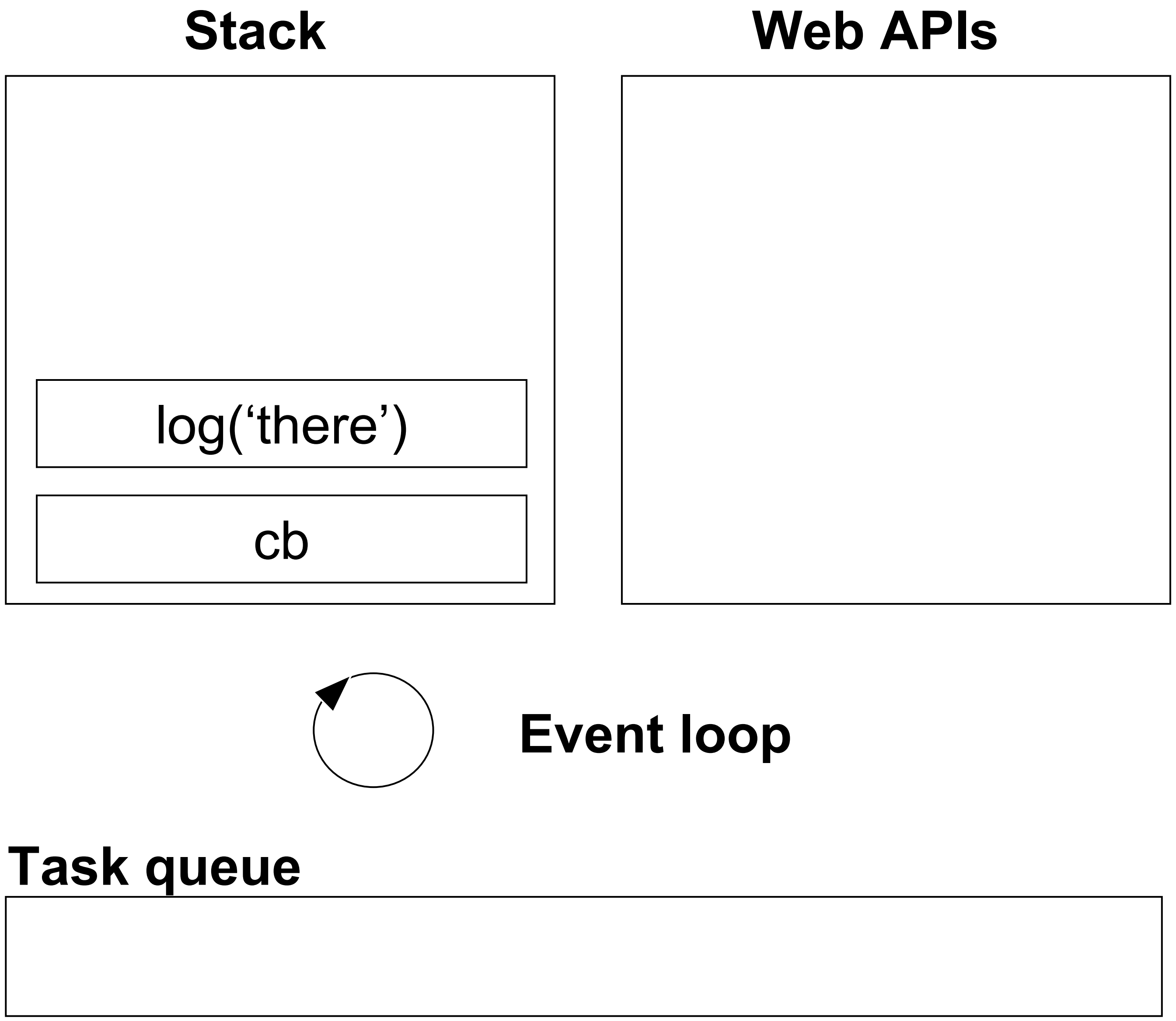


JS: Async calls & Stack

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```



Stack

log(‘there’)

cb

Web APIs

Event loop

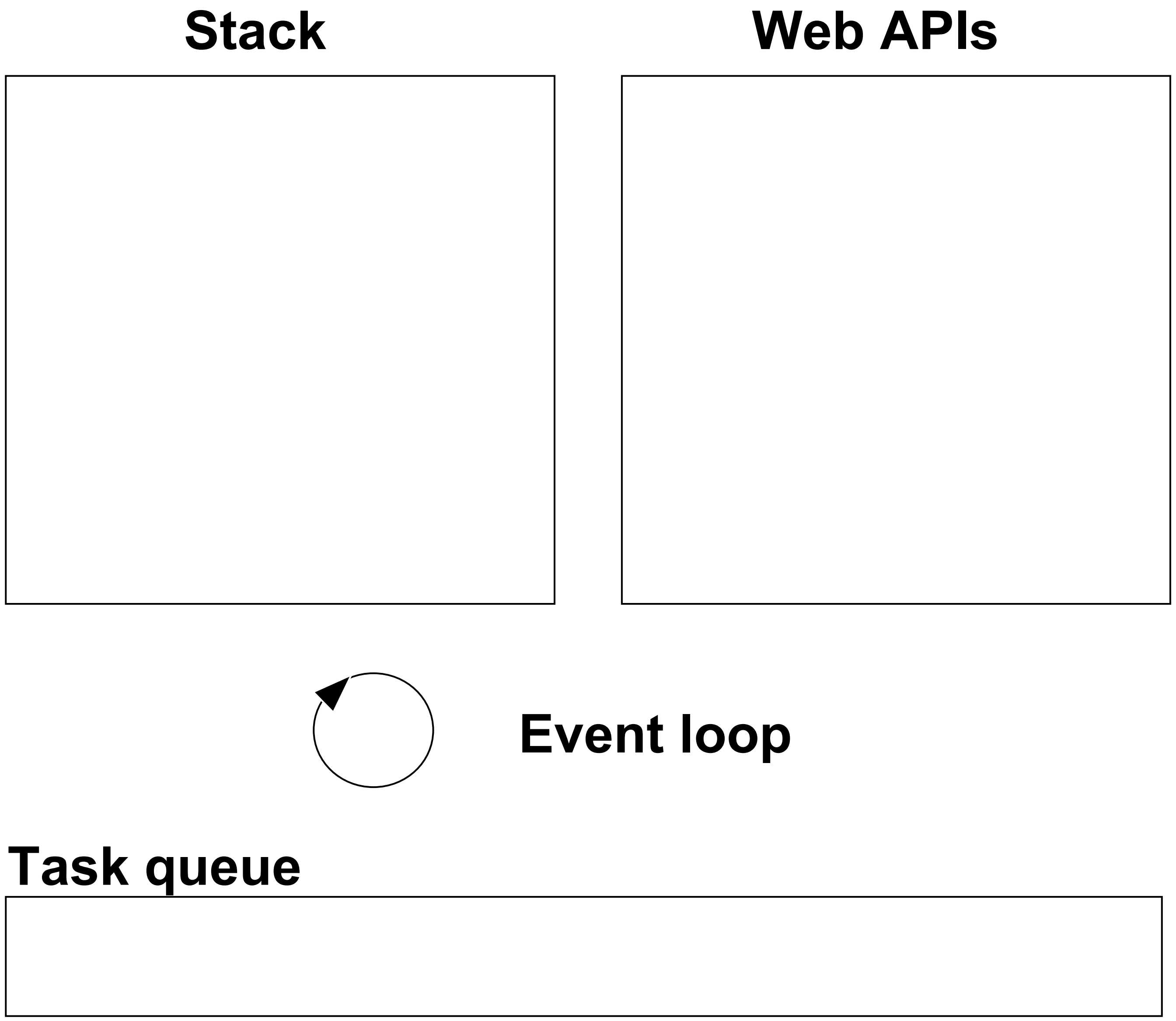
Task queue

JS: Async calls & Stack

```
console.log(‘hi’);

setTimeout(function() {
  console.log(‘there’)
}, 5000);

console.log(‘VIS’);
```



Books

- JavaScript: The Good Parts
  - O'Reilly Media / Yahoo Press
  - by Douglas Crockford
    - affiliation PayPal
- Professional JavaScript for Web Developers
  - Wrox
  - by Nicholas C. Zakas
    - affiliation ~~Y!~~ box

Bibliography

- [Murray 13] Scott Murray, “*Interactive Data Visualization for the Web*”. O’Reilly Media, 1st ed., 2013
- [Judd 75] Deane B. Judd, “*Color in business, science and industry*”. Wiley-Interscience, 3rd ed., 1975
- [TM] http://www.teaching-materials.org/