

BASIC CONCEPTS

JAVASCRIPT – PROGRAMMING LANGUAGE

EXPRESSIONS

- ▶ Piece of javascript code that produces a ***value***
 - ▶ 1 -> produces 1
 - ▶ "What nonsense" -> produces "What nonsense"
 - ▶ 3 * 5 -> produces 15
 - ▶ num > 20 -> produces either 'true' or 'false'
 - ▶ (3 + 7) * 5 -> produces 50

STATEMENT

- ▶ Its an *instruction* to the computer
 - ▶ `let x = 10;`
 - ▶ `if(x < 50) { do something }`
 - ▶ `let num = (3 + 7) * 5;`
- ▶ Expressions are always part of a statement

BINDINGS (VARIABLES)

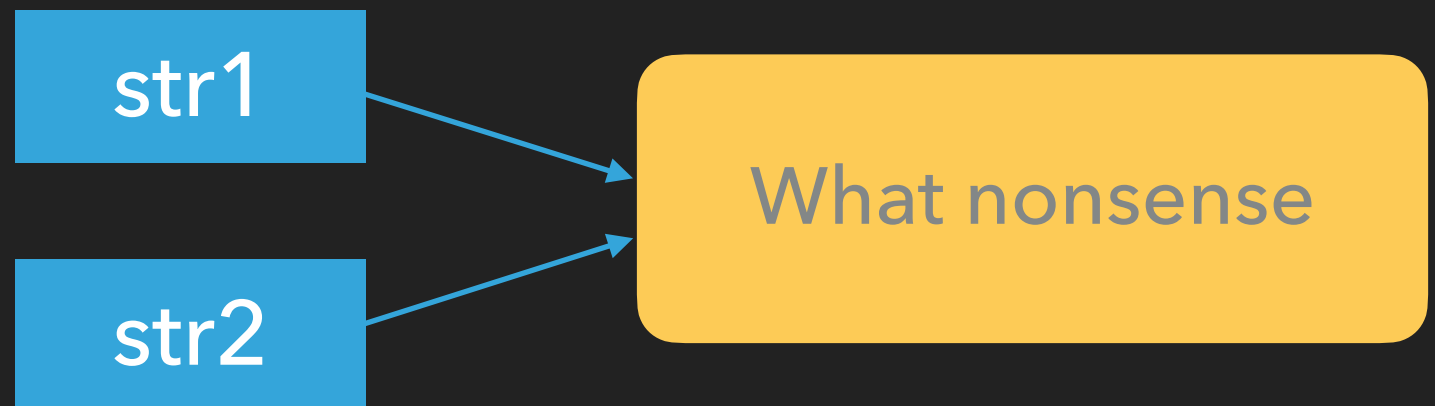
- ▶ To catch and hold values JavaScript provides a thing called binding (or variable)
 - ▶ `let x = 10;`
 - ▶ The `=` operator can be used at any time on existing bindings to disconnect them from their current value and have them point to a new one
 - ▶ You should imagine bindings as tentacles, rather than boxes. They do not contain values; they grasp them
 - ▶ two bindings can refer to the same value
- ▶ A function definition is a regular binding where the value of the binding is a function

BINDINGS (VARIABLES)

- ▶ You should imagine bindings as tentacles, rather than boxes.
- ▶ They do not contain values; they grasp them



```
1 let str1 = "What nonsense";  
2 let str2 = str1;
```



BINDINGS & SCOPE

- ▶ Each binding has a scope, which is the part of the program in which the binding is visible.
 - ▶ Where is this binding visible
- ▶ For bindings defined outside of any function or block, the scope is the whole program - **Global Scope**
- ▶ Bindings created for function parameters or declared inside a function can be referenced only in that function - **Local scope**
 - ▶ Every time the function is called, new instances of these bindings are created.
- ▶ Each local scope can also see all the local scopes that contain it, and all scopes can see the global scope
- ▶ This approach to binding visibility is called **lexical scoping**.

JS KEY CONCEPTS USED IN PRACTICE

- ▶ Closure
- ▶ Arrow functions
- ▶ Rest parameters, Spread Operator, Destructuring

ARROW FUNCTIONS

- ▶ An arrow function expression is a compact alternative to a traditional function expression
- ▶ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions
- ▶ Arrow functions don't have their own bindings to this
- ▶ Code examples
- ▶ <https://www.codementor.io/@dariogarciamoya/understanding-this-in-javascript-with-arrow-functions-gcpjwfyuc>

CLOSURE

- ▶ Functions can be treated as values
- ▶ Local bindings are recreated every time a function is called
- ▶ What happens to **local bindings** when the function call that created them is no longer active?



```
1  function wrapValue(n) {  
2    let local = n;  
3    return () => local;  
4  }  
5  
6  let wrap1 = wrapValue(1);  
7  let wrap2 = wrapValue(2);  
8  console.log(wrap1());  
9  console.log(wrap2());
```

CLOSURE

- ▶ Local bindings are created anew for every call
- ▶ Different calls can't trample on one another's local bindings.
- ▶ Being able to reference a specific instance of a local binding in an enclosing scope—is called **closure**
- ▶ A function that references bindings from local scopes around it is called a *closure*
- ▶ Think of function values as containing both the code in their body and the environment in which they are created
- ▶ When called, the function body sees the environment in which it was created, not the environment in which it is called

USES OF CLOSURE

- ▶ Maintaining state in Async calls
- ▶ Closures allow each API request to retain its context until response is received
- ▶ Closures help to retain the state for each individual request



```
1 function fetchData(url, id) {  
2   fetch(url).then(response => {  
3     // Id is preserved in each call.  
4     console.log(`response for ID ${id}:`, response);  
5   })  
6 }  
7 const urls = ['/api/user/1', '/api/user/2'];  
8 urls.forEach((url, index) => fetchData(url, index + 1));  
9
```

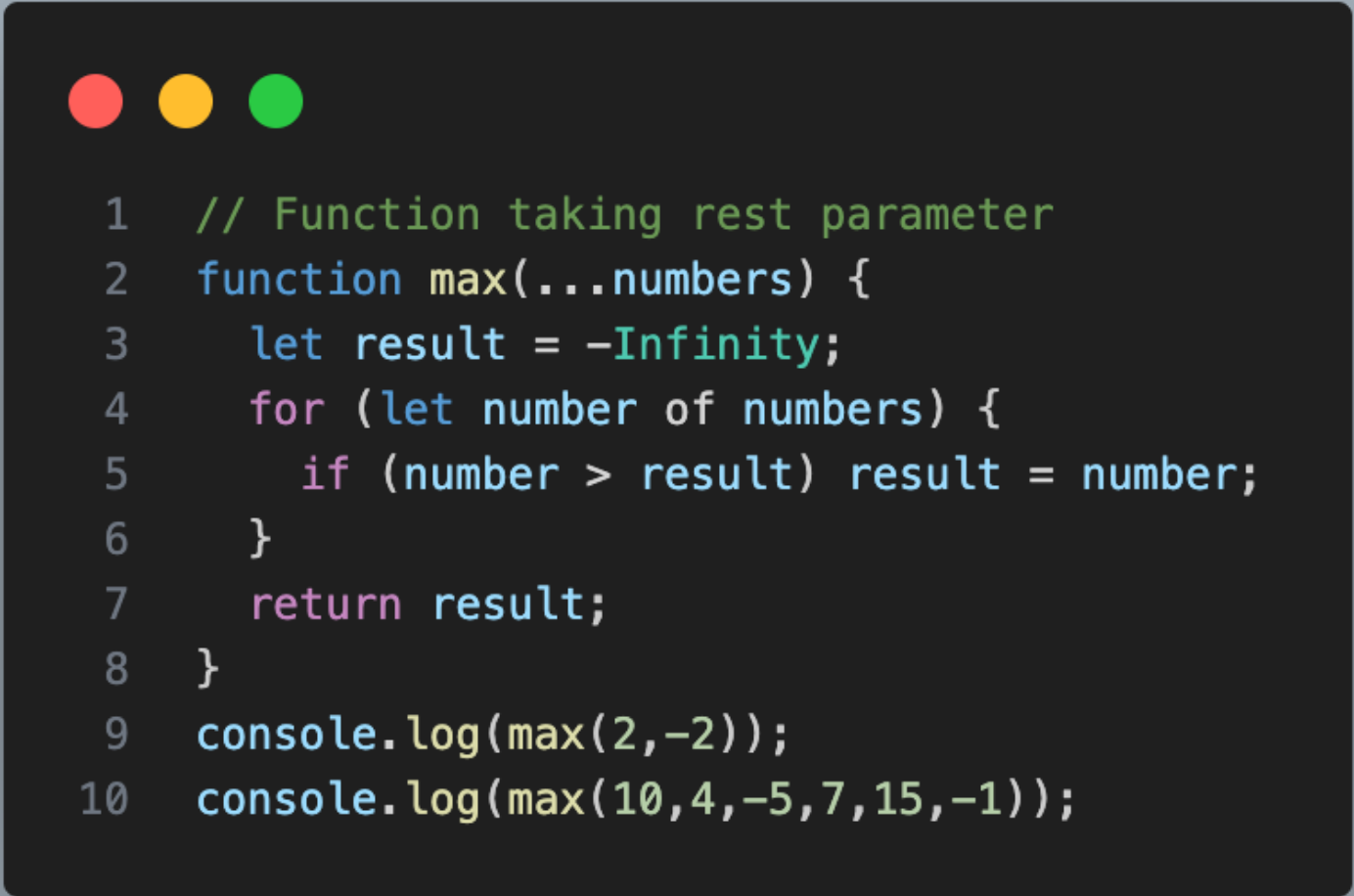
USES OF CLOSURE

- ▶ Closures create state variables within functions
- ▶ Each time **createCounter** is called, a new **count** variable is created in a separate closure
- ▶ The variable **count** is defined inside **createCounter** and is accessible only to the functions **increment** and **getCount** because they form a closure around **count**
- ▶ The **increment** and **getCount** functions retain access to this specific **count** variable, even after **createCounter** finishes executing

```
1 function createCounter() {  
2   // `count` is a state variable  
3   // Its private to the `createCounter` function  
4   let count = 0;  
5  
6   return {  
7     increment() {  
8       count++;  
9       return count;  
10    },  
11    getCount() {  
12      return count;  
13    }  
14  };  
15 }  
16  
17 const counter1 = createCounter();  
18 console.log(counter1.increment()); // 1  
19 console.log(counter1.increment()); // 2  
20 console.log(counter1.getCount());  // 2  
21 const counter2 = createCounter();  
22 console.log(counter2.increment()); // 1  
23 console.log(counter2.getCount());  // 1  
24
```

REST PARAMETERS

- ▶ It can be useful for a function to accept any number of arguments
- ▶ For example, `Math.max` computes the maximum of all the arguments



```
1  // Function taking rest parameter
2  function max(...numbers) {
3      let result = -Infinity;
4      for (let number of numbers) {
5          if (number > result) result = number;
6      }
7      return result;
8  }
9  console.log(max(2,-2));
10 console.log(max(10,4,-5,7,15,-1));
```

SPREAD OPERATOR

- ▶ The spread (...) syntax allows an iterable, such as an array or string, to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected.
- ▶ In an object literal, the spread syntax enumerates the properties of an object and adds the key-value pairs to the object being created.


```
1  // Spread operator with array
2  let arr = [1,2,3,4];
3  let [a,b,c,d] = [...arr];
4  console.log(a, b, c, d);
5
6  // Spread operator with function
   args
7  function sum(a,b) {
8      return a+b;
9  }
10 let numbers = [10,5];
11 let result = sum(...numbers);
12 console.log(result);
13
14 // Spread operator with objects
15 let obj = {
16     name: 'Rajesh',
17     age: 30
18 };
19
20 let {name, age} = obj;
21 console.log(name, age);
```

DESTRUCTURING

- ▶ The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

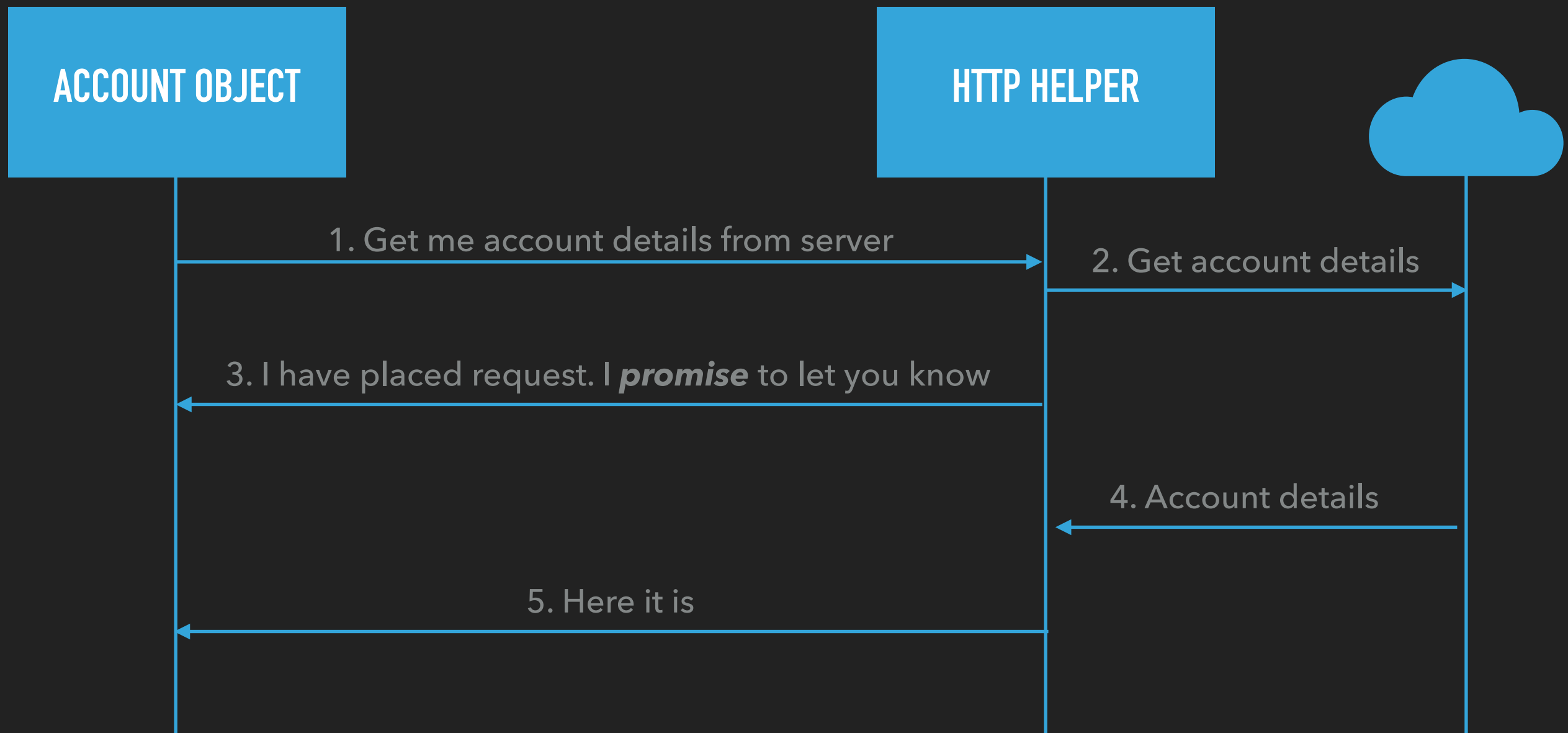


```
1  const arr = [1,2,3,4,5];
2  let [x,y,...rest] = arr;
3  console.log(x,y); // 1 2
4  console.log(rest); // [3,4,5]
5
6  const obj = {
7    a:1,
8    b:{
9      c: 2
10   }
11 }
12 let {a} = obj;
13 let {b: {c:d}} = obj;
14
15 console.log("a:",a); // 1
16 console.log("d:",d); // 2
```



ASYNCHRONOUS PROGRAMMING

WHAT'S THE PROBLEM?



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.
- ▶ Instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.
- ▶ A promise is in one of the following states
 - ▶ **pending**: initial state, neither fulfilled nor rejected.
 - ▶ **fulfilled**: meaning that the operation was completed successfully.
 - ▶ **rejected**: meaning that the operation failed.

PROMISE CHAINING



```
1  doSomething(function (result) {  
2    doSomethingElse(result, function (newResult) {  
3      doThirdThing(newResult, function (finalResult) {  
4        console.log(`Got the final result: ${finalResult}`);  
5      }, failureCallback);  
6    }, failureCallback);  
7  }, failureCallback);  
8
```



```
1  doSomething()  
2    .then(function (result) {  
3      return doSomethingElse(result);  
4    })  
5    .then(function (newResult) {  
6      return doThirdThing(newResult);  
7    })  
8    .then(function (finalResult) {  
9      console.log(`Got the final result: ${finalResult}`);  
10   })  
11   .catch(failureCallback);  
12
```

ASYNC FUNCTION

- ▶ Each time when an async function is called, it returns a new Promise
- ▶ As soon as the body returns something, that promise is resolved. If it throws an exception, the promise is rejected.
- ▶ Async functions can contain zero or more await expressions
- ▶ Inside an async function, the word await can be put in front of an expression to wait for a promise to resolve and only then continue the execution of the function
- ▶ await makes the async function appear synchronous

EVENT LOOP



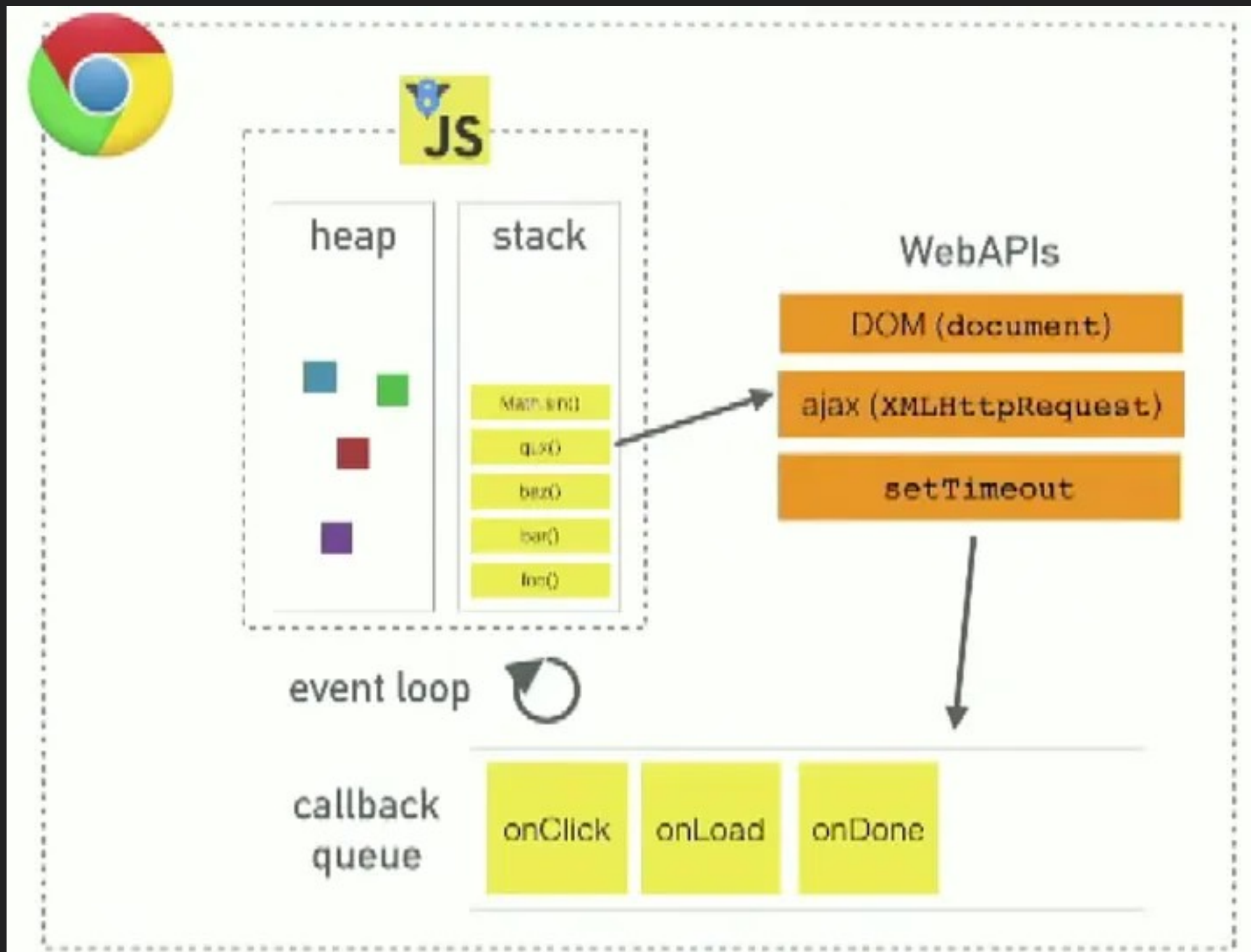
WHAT IS EVENT LOOP?

- ▶ Event loop is responsible for executing code, collecting and processing events and executing queued sub-tasks
- ▶ JavaScript language is single-threaded
- ▶ Can do one thing at a time
- ▶ Asynchronous behaviour is a part of the “environment”
- ▶ <https://medium.com/front-end-weekly/javascript-event-loop-explained-4cd26af121d4>
- ▶ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Event_loop
- ▶ <https://www.youtube.com/watch?v=8aGhZQkoFbQ&t=6s>
 - ▶ (12:48 - 14:50) - execution of code and event loop

ARCHITECTURE

- ▶ Heap - Objects are allocated in a heap which is just a name to denote a large mostly unstructured region of memory
- ▶ Stack
 - ▶ This represents the single thread provided for JavaScript code execution.
 - ▶ Function calls form a stack of frames
- ▶ Browser or Web APIs -
 - ▶ Not a part of JS runtime
 - ▶ Built on JS language
- ▶ Queue
 - ▶ 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.
- ▶ Event loop
 - ▶ Wait for messages to appear in the queue
 - ▶ Take the first message (oldest) out and put the associated function on the stack and start execution

ARCHITECTURE





JS & BROWSER

DOM

- ▶ Document Object Model
 - ▶ https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction
- ▶ DOM is the data representation of the objects that comprise the structure and content of a document on the web
- ▶ DOM is also the programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects
- ▶ It resides in the browser's memory
- ▶ Its a tree like structure
- ▶ DOM exposes many APIs to query and update the DOM structure.

DOM APIS

- ▶ https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/Using_the_Document_Object_Model
- ▶ The DOM APIs become available when the browser loads the JS
 - ▶ document or window object to access DOM
- ▶ Fundamental data types
 - ▶ document - entry point to the DOM tree
 - ▶ Node
 - ▶ Abstract base class.
 - ▶ There are different types of nodes
 - ▶ Methods for traversal of the DOM
 - ▶ Element
 - ▶ Type of node
 - ▶ Methods that describe an element (For example: Table)
- ▶ Attr - Elements attributes are represented as objects

FREQUENTLY USED APIS

- ▶ Selector interface

- ▶ The Selectors API provides methods that make it quick and easy to retrieve Element nodes from the DOM by matching against a set of selectors
- ▶ https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/Locating_DOM_elements_using_selectors

- ▶ Creating DOM tree

- ▶ createElement()
- ▶ setAttribute()
- ▶ appendChild()
- ▶ https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/How_to_create_a_DOM_tree

EVENTS

- ▶ https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events
- ▶ Events are fired inside the browser window, and tend to be attached to a specific item that resides in it
- ▶ Single or multiple elements may be associated with the event
- ▶ Types of events
 - ▶ Click, hover, select
 - ▶ Key press event
 - ▶ Window resized
 - ▶ Page loaded
 - ▶ Form submission
 - ▶ Error
- ▶ To react to an event, you attach an event handler to it - `addEventListener(event, handler)`