

<https://javascript.info/>

ECMAScript: is the specification for the JavaScript, Browsers implements these specifications i.e.,

TC39 is the community, which is maintaining ECMAScript versions, this is an open community. Anyone can submit a proposal for new features in JavaScript. TC39 will review it and approve the proposal, then a Draft need to be prepared against this new feature.

TC39 always focus on the backwards compatibility, hence JavaScript is the 100% backward compatible language. Codes written in 2000 will still run-on latest browsers.

Versions

June 1997 First edition

[6th Edition – ECMAScript 2015](#)

[7th Edition – ECMAScript 2016](#)

[8th Edition – ECMAScript 2017](#)

[9th Edition – ECMAScript 2018](#)

[10th Edition – ECMAScript 2019](#)

[11th Edition – ECMAScript 2020](#)

[12th Edition – ECMAScript 2021](#)

All ECMAScript version comparison is available at the bottom of this document.

JavaScript is a dynamically typed interpreted language. Unlike statically typed languages i.e. C#, JavaScript determines types of the variable at the time of value assignment. Default value assigned to variables is undefined (which is a primitive type)

Transcompiler

Babel is a free and open-source JavaScript trans compiler that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript that can be run by older JavaScript engines. Babel is a popular tool for using the newest features of the JavaScript programming language.

Polyfill

A polyfill is a piece of code (usually JavaScript on the Web) used to provide modern functionality on older browsers that do not natively support it.

Strict Mode

The JavaScript strict mode is a feature in ECMAScript 5. You can enable the strict mode by declaring this in the top of your script/function. “use strict”; When a JavaScript engine sees this directive, it will start to interpret the code in a special mode.

Features

- Preventing the Use of Undeclared Variables.

- Eliminating Octal Literal Syntax
- Disallowing Duplicate Parameter Names
- Preventing the Use of **this** in Global Context: In non-strict mode, using **this** in the global context (outside of a function) refers to the global object (e.g., **window** in browsers). Strict mode changes this behavior, making **this** undefined in the global context.
- Does not allow to use reserve words as variable names.
- Does not allow to delete variable, functions.
- Does not allow variable with name eval.
- Does not allow to leak out variable declared within eval()

Primitive types are passed by value whereas Object types are passed by reference.

JavaScript simply ignores extra parameter passed to the function, and in case less parameter is passed then remaining parameters will get initialized with undefined.

Every function has an object i.e., **arguments** which has all input parameter as property. The "arguments" object is an array-like object that holds all the arguments passed to the function, regardless of whether the function explicitly defines parameters to receive those arguments.

Rest Operator:

Notation is (...), The **rest parameter** syntax allows a function to accept an indefinite number of arguments as an array, compresses or converts list of individual elements into array, most basic use-case scenario is function parameters.

- A function definition can have only one *...restParam*.
- The rest parameter must be the last parameter in the function definition.

```
function myFun(a, b, ...manyMoreArgs) {
  console.log("a", a)
  console.log("b", b)
  console.log("manyMoreArgs", manyMoreArgs)
}

myFun("one", "two", "three", "four", "five", "six")

// Console Output:
// a, one
// b, two
// manyMoreArgs, ["three", "four", "five", "six"]
```

Spread Operator:

Notation is (...), allows an iterable such as an array expression or string to be expanded, or an object expression to be expanded.

Iterable expression(string or array):

```
[...iterableObj, '4', 'five', 6]; // combine two arrays by inserting  
all elements from iterableObj
```

Object:

```
let objClone = { ...obj }; // pass all key:value pairs from an object
```

Template String:

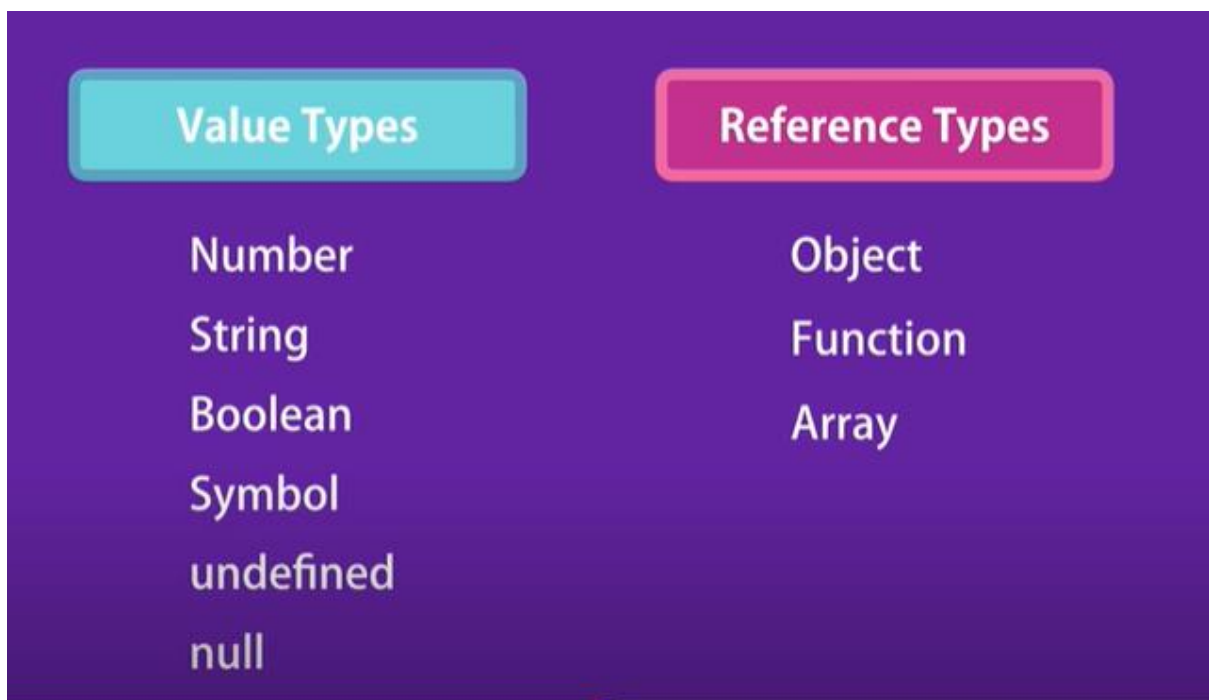
Notation is (``` Backtick), allowing embedded expressions called *substitutions*.

1. Untagged template literals result in strings, which makes them useful for string interpolation (and multiline strings, since unescaped newlines are allowed).

```
// Untagged, these create strings:  
`string text ${expression} string text`
```

2. Tagged template literals call a function (the tag function) with an array of any text segments from the literal followed by arguments with the values of any substitutions, which is useful for

```
// Tagged, this calls the function "example" with the template as  
// first argument and substitution values as subsequent arguments:  
example`string text ${expression} string text`  
function example(strings, ...keys)
```



Data types

1. String
2. Number
3. Boolean
4. Null
5. Undefined
6. Object

- Primitives except `null` and `undefined` provide many helpful methods.
- Formally, these methods work via temporary objects, but JavaScript engines are well tuned to optimize that internally, so they are not expensive to call.
- Temporary object just creates before calling the inbuilt functions and get destroyed once the function returns.

Undefined and Null

`undefined` means a variable has been declared but has not yet been assigned a value. `undefined` is a type by itself (`undefined`). Unassigned variables are initialized by JavaScript with a default value of `undefined`. On the other hand, `null` is an object. It can be assigned to a variable as a representation of no value. JavaScript never sets a value to `null`. That must be done programmatically.

Undefined vs null - the differences

1) Data types:

The data type of `undefined` is `undefined` whereas that of `null` is `object`.

2) In arithmetic operations:

When used in arithmetic operations, `undefined` will result in `NaN` (not a number), whereas `null` will be converted to `0` behind the screens.

3) `Undefined` and `null` are falsy:

When used in conditional logic both `undefined` and `null` will return `false`.

```
typeof null           // "object" (not "null" for legacy reasons)
typeof undefined      // "undefined"
null === undefined    // false
null == undefined     // true
null === null         // true
null == null          // true
!null                 // true
isNaN(1 + null)       // false
isNaN(1 + undefined) // true
```

NaN

`NaN` is a property of the global object. In other words, it is a variable in global scope. The initial value of `NaN` is Not-A-Number — the same as the value of `Number`. `NaN` In modern browsers, `NaN` is a non-configurable, non-writable property. Even when this is not the case, avoid overriding it. It is rather rare to use `NaN` in a program.

There are five different types of operations that return `NaN`:

- Number cannot be parsed (e.g., `parseInt("blabla")` or `Number(undefined)`)
- Math operation where the result is not a real number (e.g., `Math.sqrt(-1)`)
- Operand of an argument is `NaN` (e.g., `7 ** NaN`)
- Indeterminate form (e.g., `0 * Infinity`, or `undefined + undefined`)

- Any operation that involves a string and is not an addition operation (e.g., "foo" / 3)

```
NaN === NaN;           // false
Number.NaN === NaN;    // false
isNaN(NaN);             // true
isNaN(Number.NaN);      // true
Number.isNaN(NaN);      // true

function valueIsNaN(v) { return v !== v; }
valueIsNaN(1);           // false
valueIsNaN(NaN);         // true
valueIsNaN(Number.NaN);  // true
```

Execution Context:

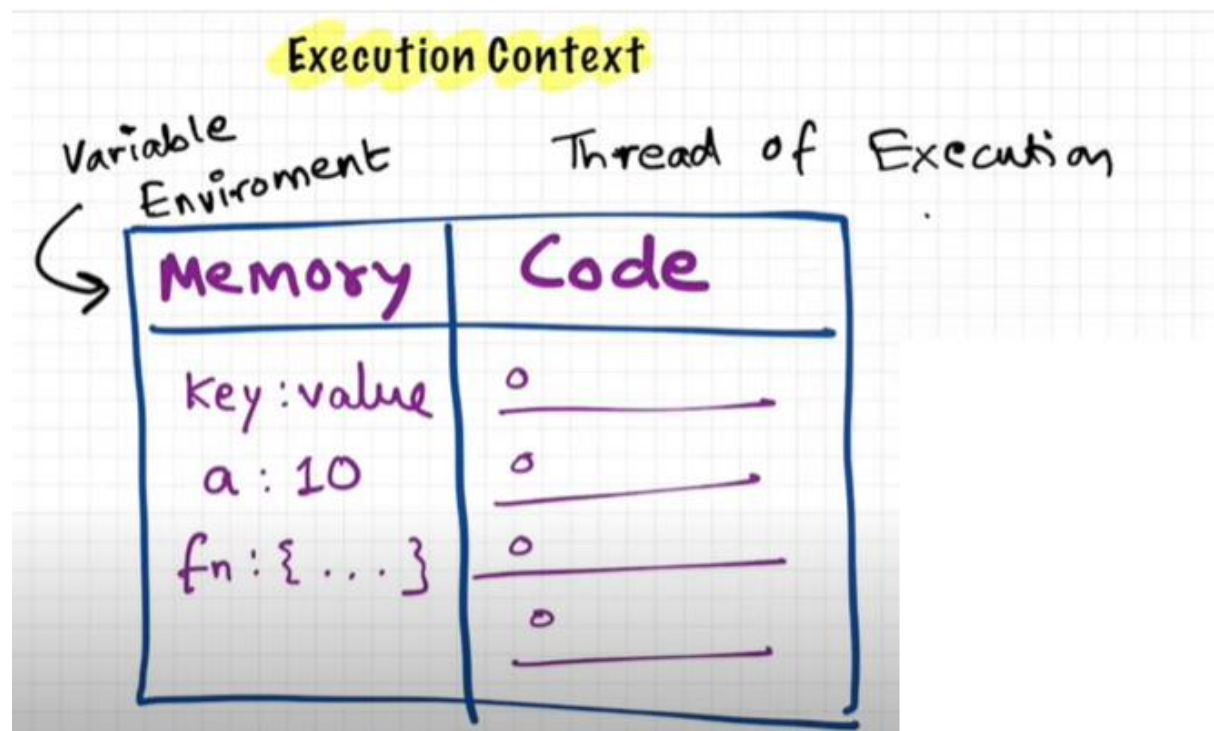
**“Everything in JavaScript
happens inside an
Execution Context”**

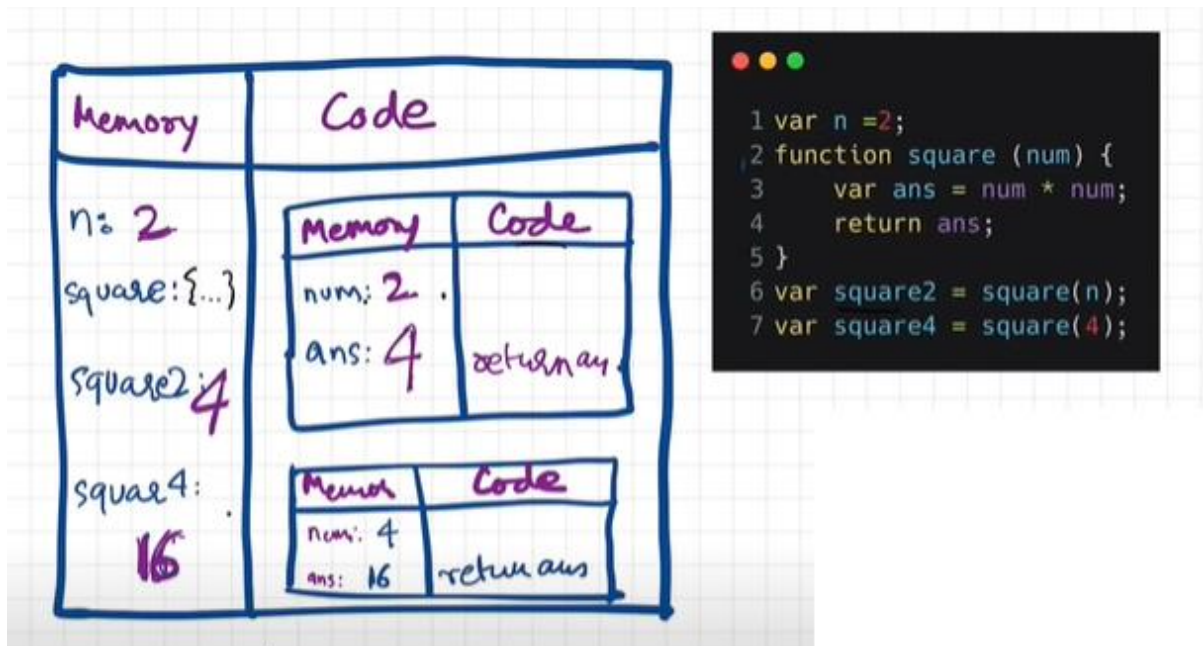
“JavaScript is a
synchronous
single-threaded
language”

Execution Context: Memory + Code || (Variable Environment + Thread of Execution)

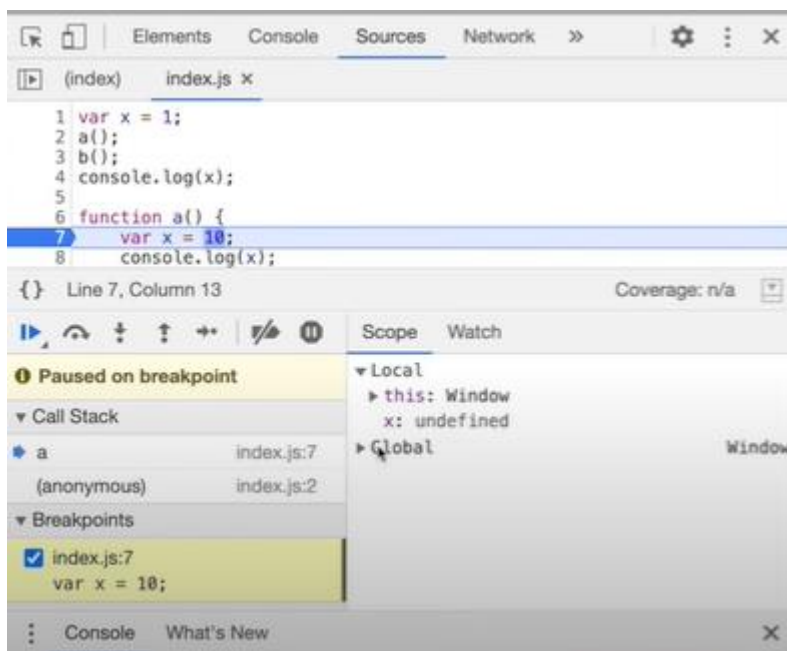
Two Phases

1. Creation Phase
2. Execution Phase





“Call stack maintains the **order of execution** of execution contexts”



Types

1. Global Execution Context
 - a. global object[window] + this + variable environment + lexical environment + thread of execution
2. Function Execution Context
 - a. arguments object + this + variable environment + lexical environment + thread of execution

🚦 Whenever Global Execution context is created “this”, and “global” object get created where “this” points to “global object”; In case of browser “global” is “window”

Scope:

Context in which variables are visible or can be referenced is called as the scope, There three types of scopes available

1. Global: visibility everywhere
2. Function: visibility inside Functions only
3. Block: visibility inside blocks only (if let or const is used, else global visibility)

Memory sections

1. Block : let and const in block
2. Script: let and const at outside any function
3. Local : everything inside function
4. Closure:
5. Global: var declare outside any function or inside block and that block is outside any function

Variables with var keyword inside block are available in global context, hence let and const were introduced, these keywords will limit the visibility of declared variable to Block level only. Const can get initialized only once.

let and const :

- does not get declared inside global/window object
- cannot be re-declared: JavaScript throws syntax error even before executing a single line

****Uncaught SyntaxError: Identifier 'a' has already been declared**

```
main.js X
1 // "use strict";
2
3 var a=10;
4 let a=100; ❌
```

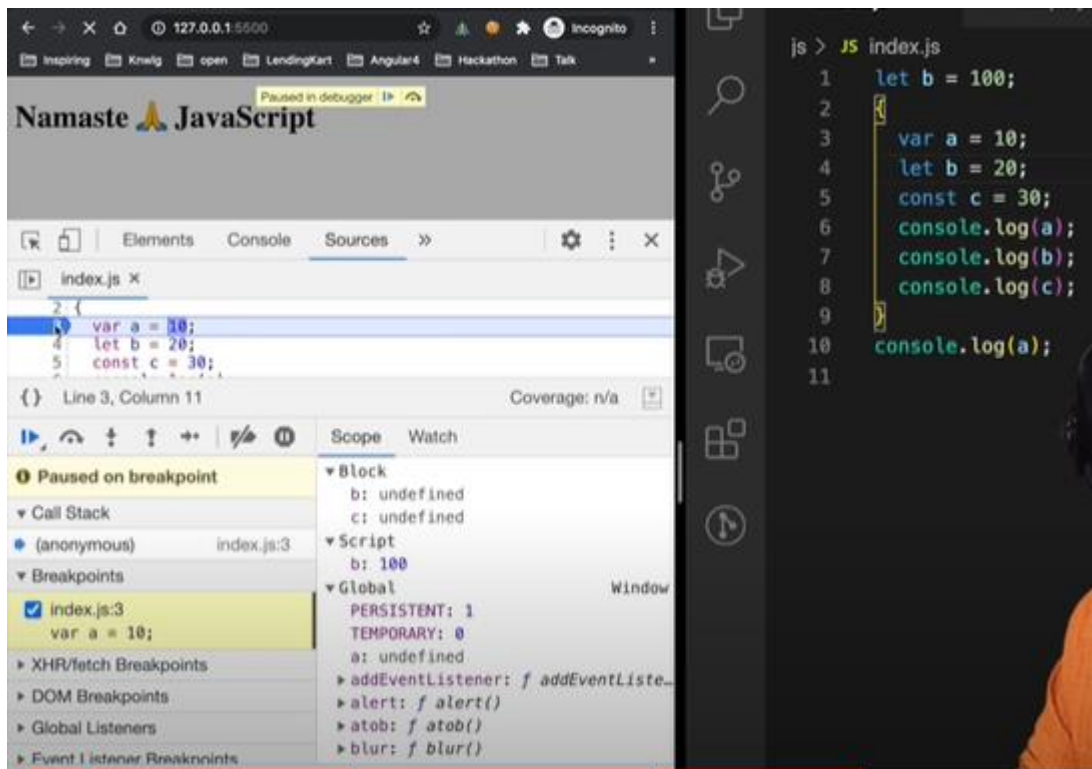
What is a Block: Block is a compound statement that is defined by curly braces {} and used to combine multiple statements into one statement where JavaScript expects only one statement. And all the variables and functions that can be accessed inside a block are said to be inside that block scope, hence called **Block scoped**.

Shadowing: Now, when a variable is declared in a certain scope having the same name defined on its outer scope and when we call the variable from the inner scope, the value assigned to the variable in the inner scope is the value that will be stored in the variable in the memory space. This is known as **Shadowing or Variable Shadowing**. In JavaScript, the introduction of let and const in ECMAScript 6 along with block scoping allows variable shadowing.

(Shadowing can overwrite outer scoped variable i.e., in case of var & var)

According to variable scoping rules, the inner scope should always be able to access a variable defined in the outer scope, but in practice, shadowing will prevent that from happening.

Illegal Shadowing: Now, while shadowing a variable, it should not cross the boundary of the scope, i.e., we can shadow var variable by let variable but cannot do the opposite. So, if we try to shadow let variable by var variable, it is known as **Illegal Shadowing**, and it will give the error as “variable is already defined.”



Hoisting:

JavaScript Hoisting refers to the process whereby the interpreter appears to move the *declaration* of functions, variables, or classes to the top of their scope, prior to execution of the code.

Note: Conceptually variable hoisting is often presented as the interpreter "splitting variable declaration and initialization and moving (just) the declarations to the top of the code".

Variables declared with let and const are also hoisted but, unlike var, are not initialized with a default value. An exception will be thrown if a variable declared with let or const is read before it is initialized.

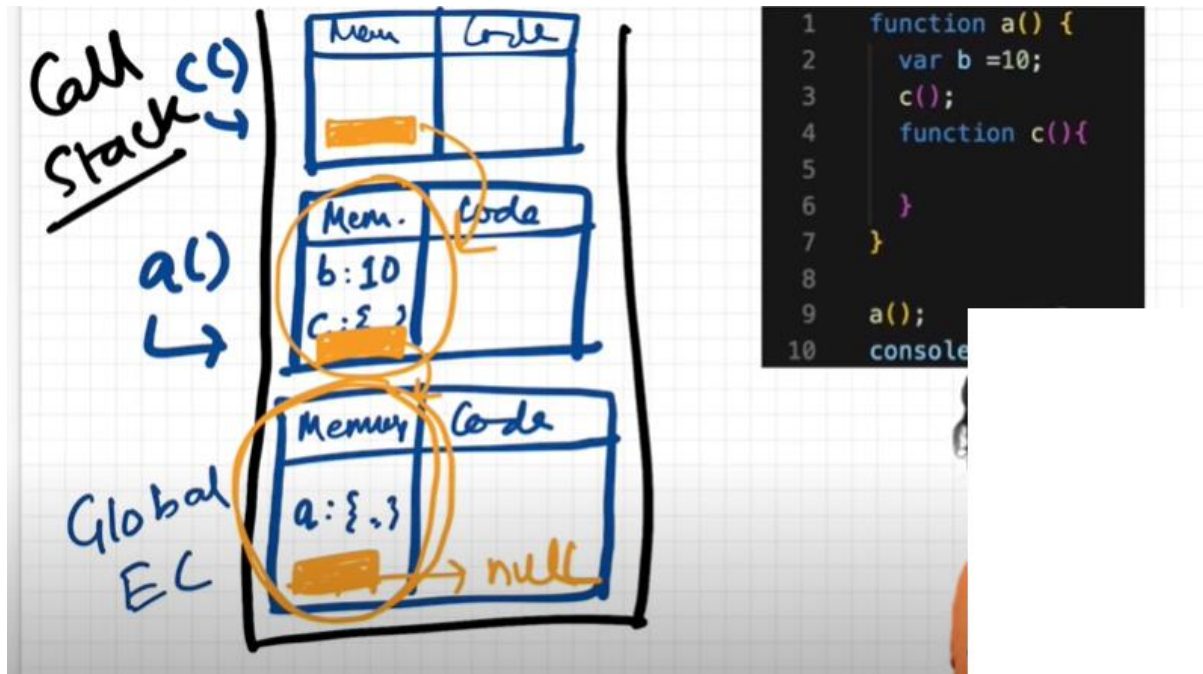
```
console.log(num); // Throws ReferenceError exception as the variable value is uninitialized
let num = 6; // Initialization
```

Temporal Dead Zone: time since when let/const variable is hoisted till it is initialized. Variable cannot be accessed during this time.

Scope Chain:

Whole chain of (local memory + lexical environment of parent) is called as the scope chain.

Lexical Environment: Local Memory + Lexical Environment of its Parent



IIFE:

Immediately Invoked Function Expression

An **IIFE** (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined

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

Use Cases

1. Avoid polluting the global namespace

Because our application could include many functions and global variables from different source files, it's important to limit the number of global variables. If we have some initiation code that we don't need to use again, we could use the IIFE pattern. As we will not reuse the code again, using IIFE in this case is better than using a function declaration or a function expression.

```
(function () {
  // some initiation code
  let firstVariable;
  let secondVariable;
})();

// firstVariable and secondVariable will be discarded after the function is executed.
```

2. For loop with var before ES6

We could see the following use of IIFE in some old code, before the introduction of the statements **let** and **const** in **ES6** and the block scope. With the statement **var**, we have only function scopes and the global scope. Suppose we want to create 2 buttons with the texts Button 0 and Button 1 and we click them, we would like them to alert 0 and 1. The following code doesn't work:

```
for (var i = 0; i < 2; i++) {
  const button = document.createElement("button");
  button.innerText = "Button " + i;
  button.onclick = function () {
    alert(i);
  };
  document.body.appendChild(button);
}
console.log(i); // 2
```

When clicked, both Button 0 and Button 1 alert 2 because `i` is global, with the last value 2. To fix this problem before ES6, we could use the IIFE pattern:

```
for (var i = 0; i < 2; i++) {
  const button = document.createElement("button");
  button.innerText = "Button " + i;
  button.onclick = (function (copyOfI) {
    return function() {alert(copyOfI);};
  })(i);
  document.body.appendChild(button);
}
console.log(i); // 2
```

Closure:

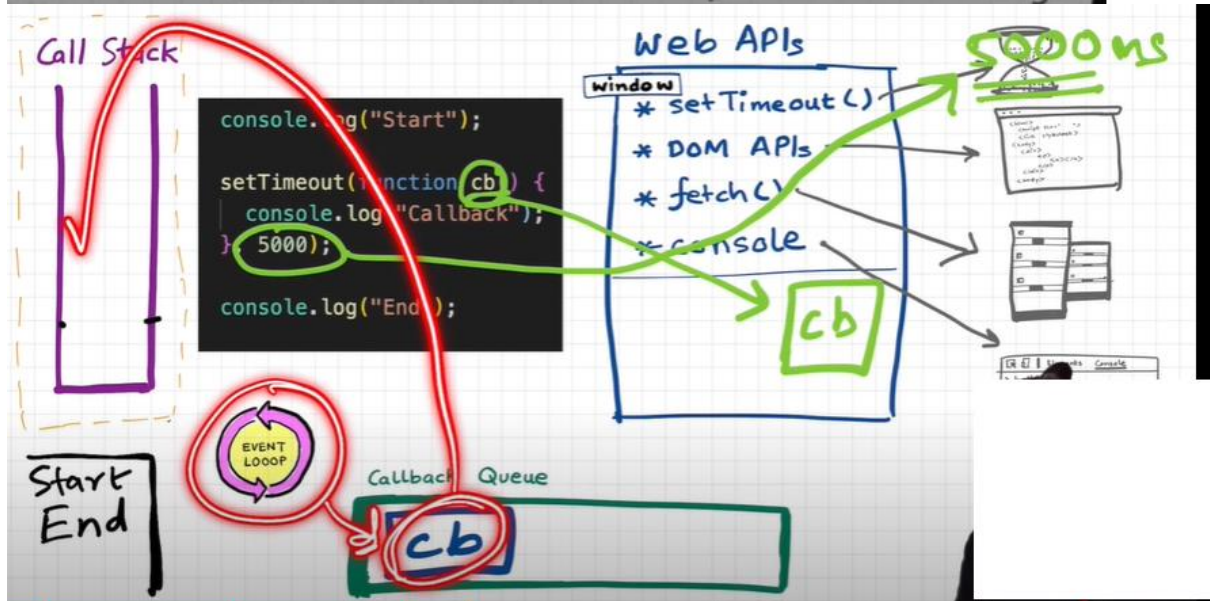
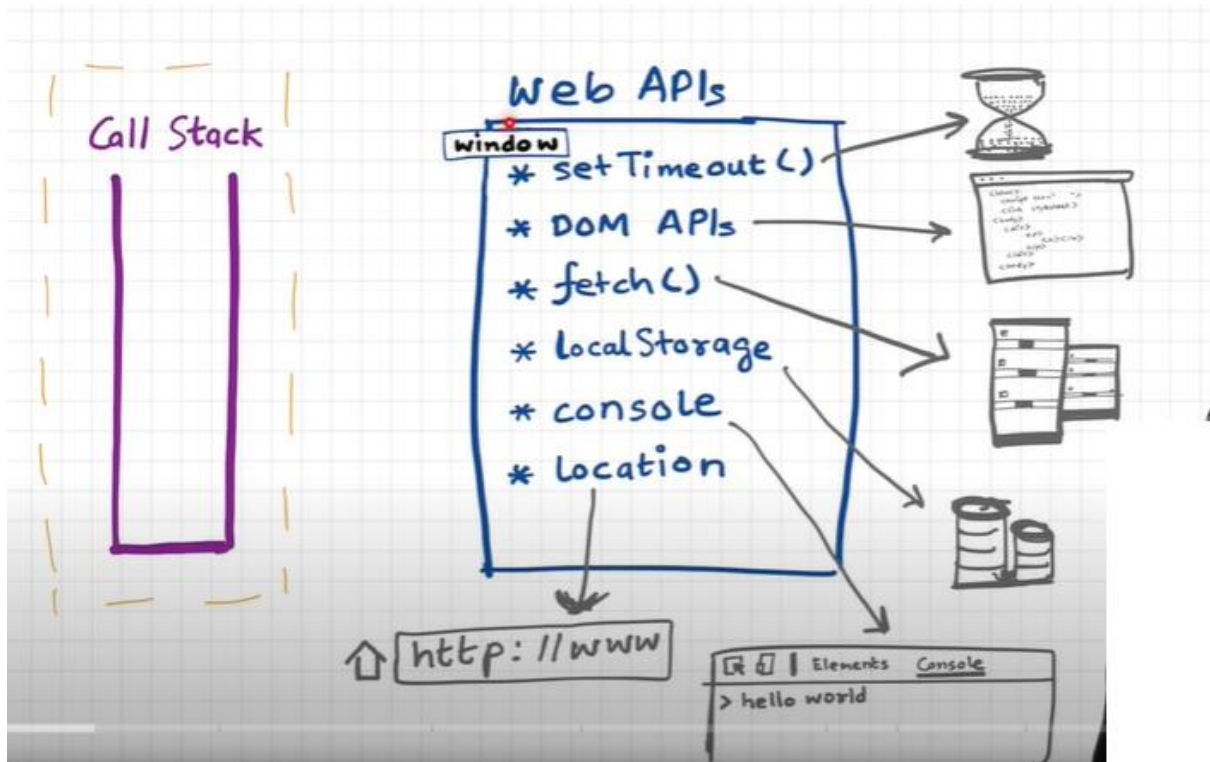
A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer

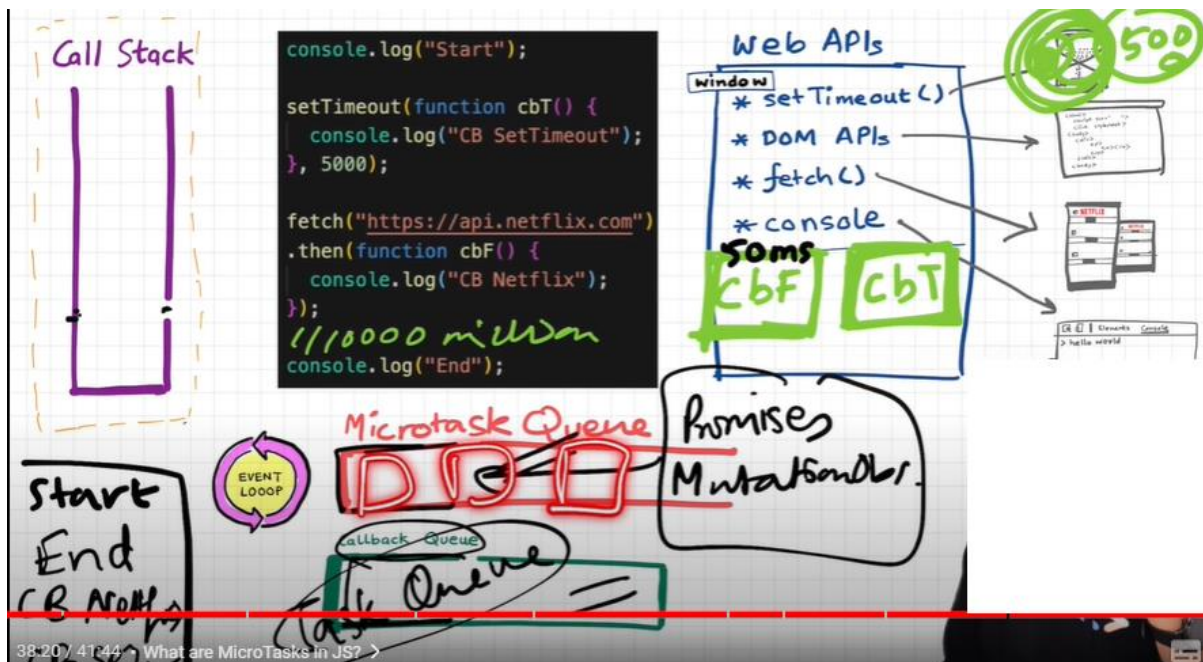
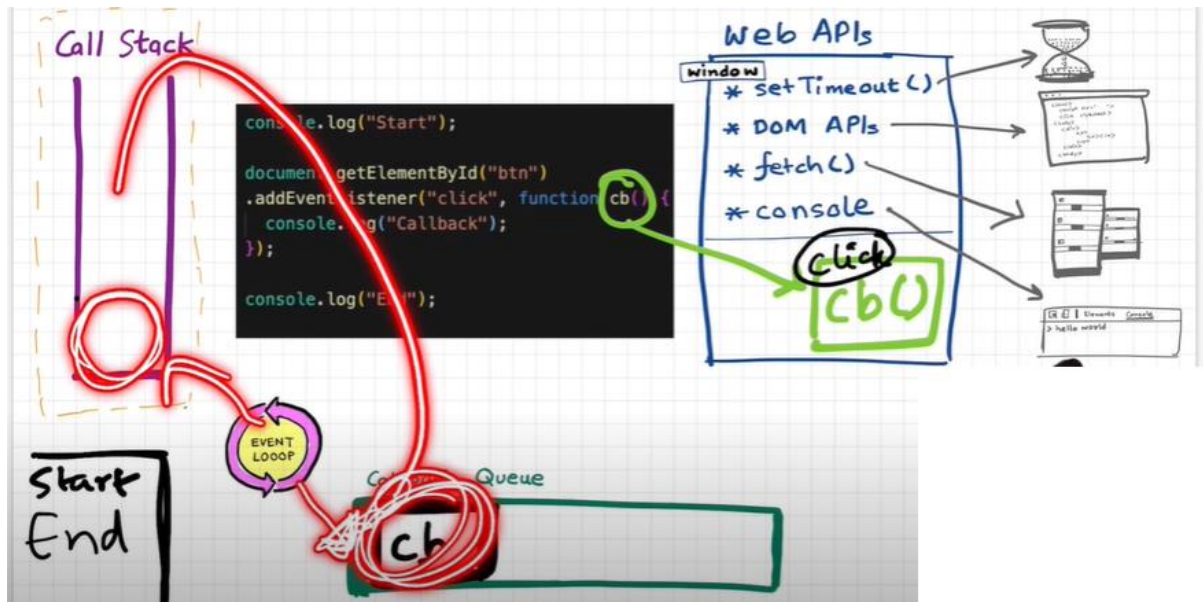
function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

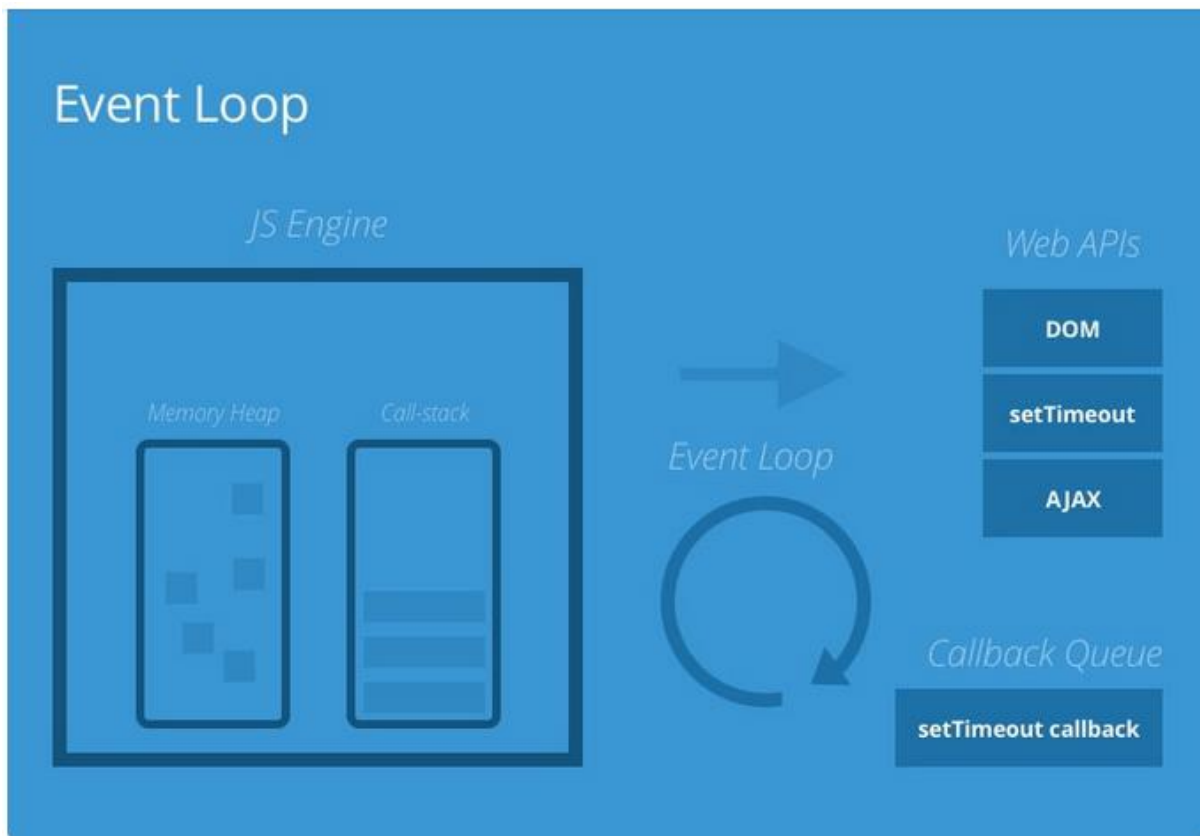
- > Uses of Closures:
 - Module Design Pattern
 - Currying
 - Functions like once
 - memoize
 - maintaining state in async world
 - setTimeouts
 - Iterators
 - and many more...

Some jargons related to Functions

- Function Statement
- Function Declaration
- Function Expression
- Named Function Expression
- Anonymous Function
- Difference between Parameters and Arguments
- First Class Function
- Higher order Function/"accepts callback function"
- Arrow Function







Event Loop: Picks callback functions from Callback Queue/Microtask Queue and push it into the call stack of JS engine; Event loop pushes CBFs only if Call stack is empty.

Promises and Mutation Observers goes to Microtask Queue; all other callback functions go to Callback queue.

Event Loop picks callback queue functions only if Microtask Queue is empty. (Microtask Queue has precedence over Callback Queue)

STARVATION of Callback Queue: Due to Microtask Queue's precedence there might arise case that callback queue function does not get chance to get picked by event loop, for a long time.

in the case of event listeners(for example click handlers), the original callbacks stay in the web API environment forever, that's why it's advised to explicitly remove the listeners when not in use so that the garbage collector does its job.

****Triggering clicks, hover etc. via js code, will not put callback functions into the callback queue, it will behave like synchronous code execution as these uses internally new DispatchEvent() object.**

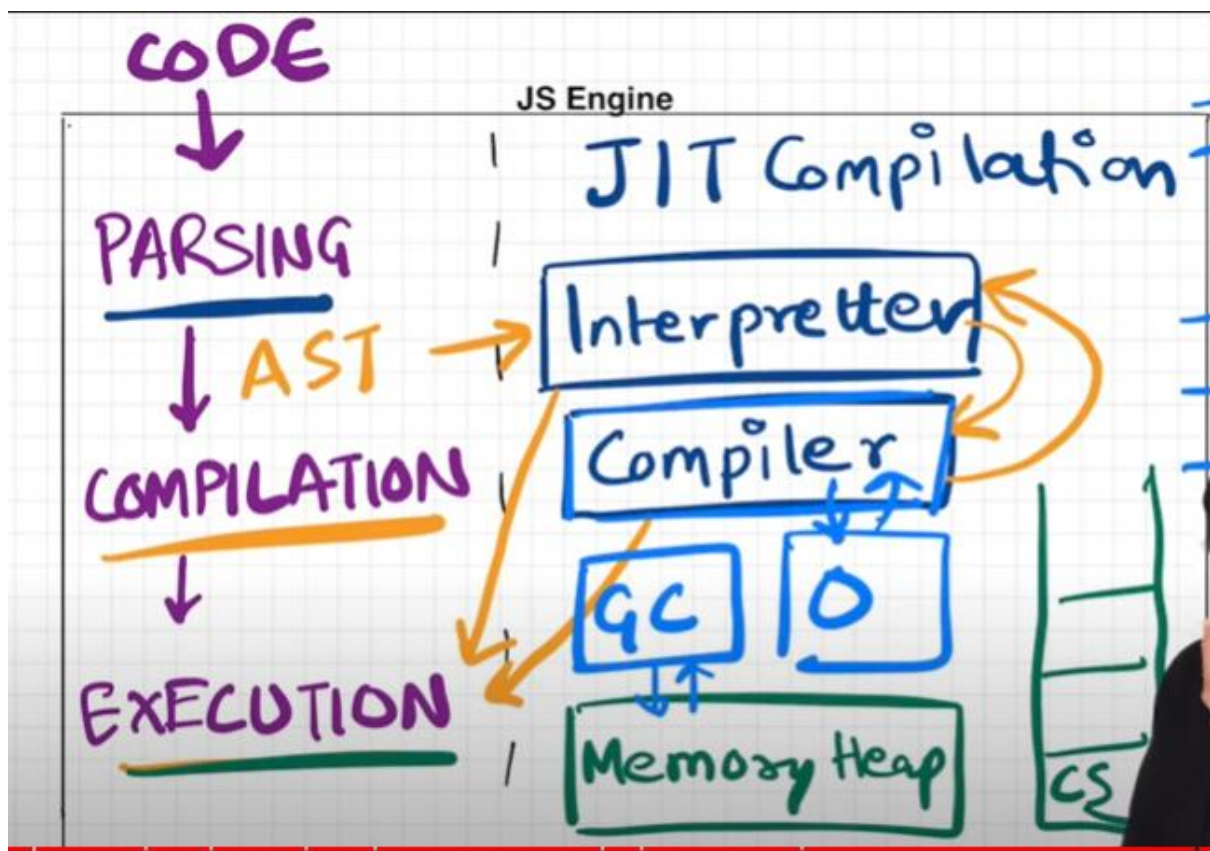
Question: lets assume the CBF is making a call to some global function but since the call stack is empty by the time CBF is called then how it will be referring the global function?

Question: why call stack need to be empty before event loop could pick and push item from callback/microtask queue?

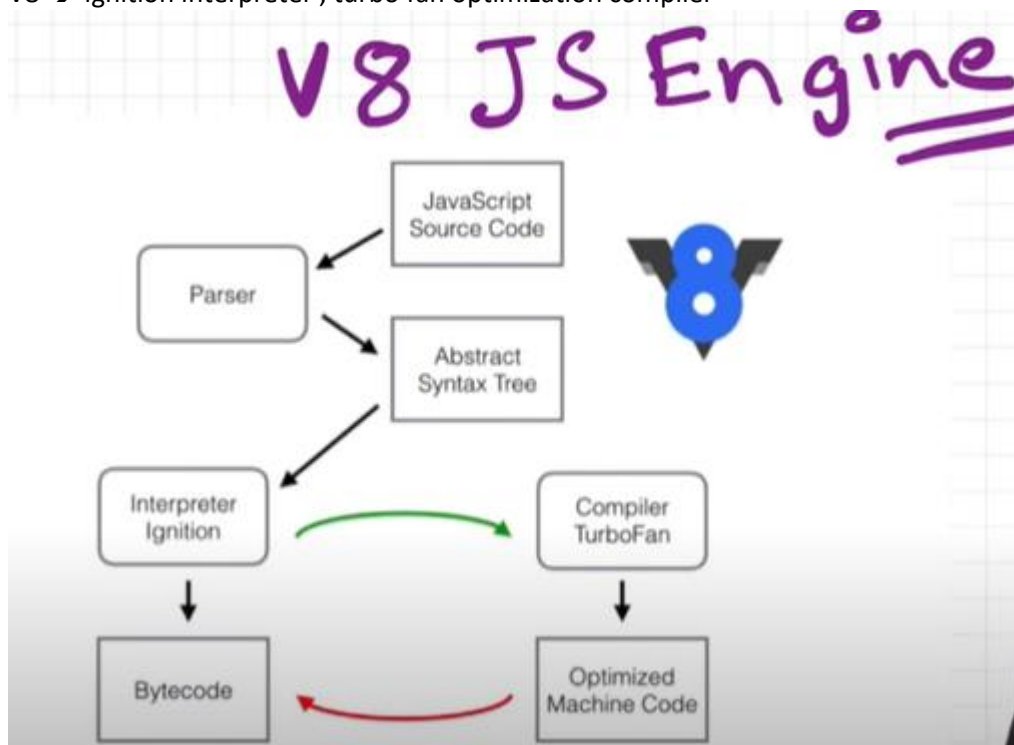
First JS engine was Spider Monkey by Brendon Eich for Mozilla.

<https://astexplorer.net/>

JIT Compilation = Compilation and Interpretation



V8 → ignition interpreter , turbo fan optimization compiler



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.

```
const x = [1, 2, 3, 4, 5];  
const [y, z] = x;  
console.log(y); // 1  
console.log(z); // 2
```

```
const o = {p: 42, q: true};  
const {p: foo, q: bar} = o;  
  
console.log(foo); // 42  
console.log(bar); // true
```

Looping

- For
 - Break and continue support.
 - Return will exit out of the function; will not run next sequence.
 - Using return without “for” loop wrapped inside a function, will raise an error.
- forEach(works on array only)
 - using break & continue will raise error.
 - using return will not terminate the loop sequence.
 - One of the disadvantages of forEach is that you cannot use continue or break and return statements don't behave as you might expect.
- For—In(works on iterable[string/array] and Objects)
 - Used to loop over object property keys.
 - Used to loop over array element index.
 - Used to loop over string char index.
- For—of(works on iterable[string/array])
 - Used to loop over array element values.
 - Used to loop over string char values.

“this” keyword

“this” keyword depends upon the way function has been invoked.

Functions, in JavaScript can be invoked in multiple ways :

1. Function invocation (“this” will refer “global object” | “window”)
2. Method invocation (“this” will refer to the object on which function is called)
3. Constructor invocation (“this” will refer to the same constructor object)

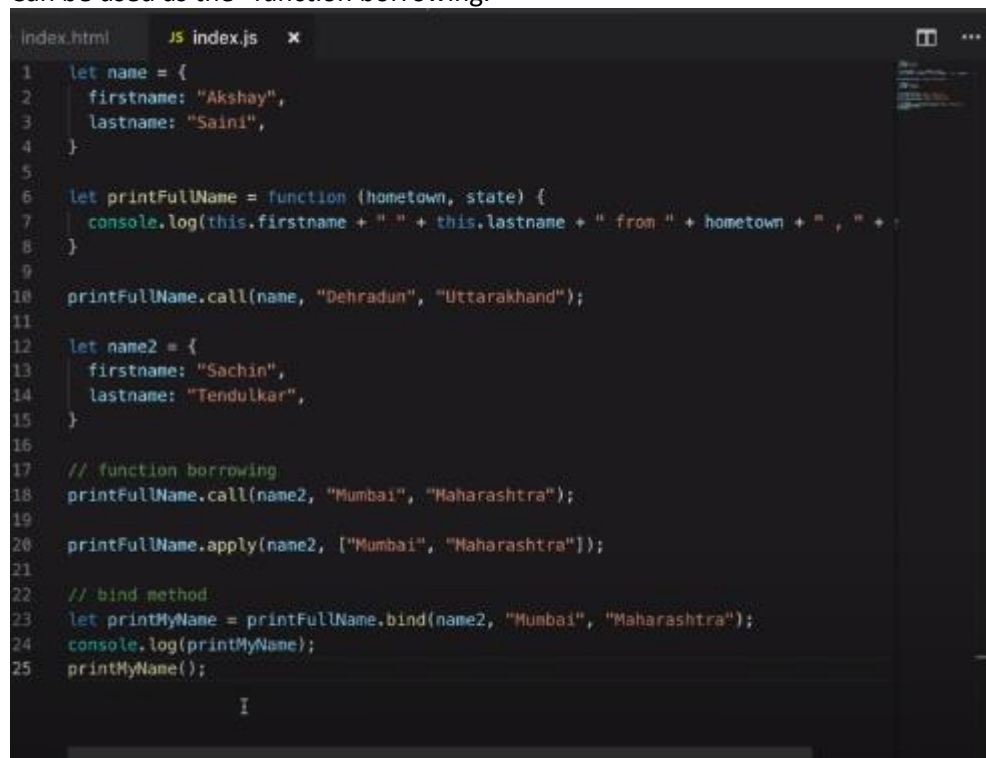
Constructor function behind the scenes: what happens when we call constructor function.

1. Firstly, an empty object is created that is an instance of the function name used with new(i.e. : people(name, age)). In other words, it sets the constructor property of the object to the function used in the invocation(people(name, age)).
2. Then, it links the prototype of the constructor function(people) to the newly created object, thus ensuring that this object can inherit all properties and methods of the constructor function.
3. Then, the constructor function is called on this newly created object. If we recall the method invocation, we will see that is similar. Thus, inside the constructor function, this gets the value of the newly created object used in the call.
4. Finally, the created object, with all its properties and methods set, is returned to person1.

Call, Apply, Bind :

Bind can be applied directly on Function Statement, but not to Function Expressions way to stabilizing the behavior of “this.”

Can be used as the “function borrowing.”



```
1  let name = {
2    firstname: "Akshay",
3    lastname: "Saini",
4  }
5
6  let printFullName = function (hometown, state) {
7    console.log(this.firstname + " " + this.lastname + " from " + hometown + " , " + state);
8  }
9
10 printFullName.call(name, "Dehradun", "Uttarakhand");
11
12 let name2 = {
13   firstname: "Sachin",
14   lastname: "Tendulkar",
15 }
16
17 // function borrowing
18 printFullName.call(name2, "Mumbai", "Maharashtra");
19
20 printFullName.apply(name2, ["Mumbai", "Maharashtra"]);
21
22 // bind method
23 let printMyName = printFullName.bind(name2, "Mumbai", "Maharashtra");
24 console.log(printMyName);
25 printMyName();
```

Normal Functions internal representation:

```
function Circle(radius) {
  this.radius = radius;
  this.draw = function() {
    console.log('draw');
  }
}
```

```
Circle.call(window, 1)
```

```
const another = Circle(1);
```

Constructor Functions Internal representation:

```
1 function Circle(radius) {
2   this.radius = radius;
3   this.draw = function() {
4     console.log('draw');
5   }
6 }
7
8
9 Circle.call({}, 1)
10
11 const another = new Circle(1);
```

Fat Arrow | Arrow Function

“this” keyword is lexically bound

If {} used, then return need to be explicitly defined

ChildObject

Object.Create();

Prototype:

One object trying to access the property of the other Object

Object.prototype.__proto__ == null

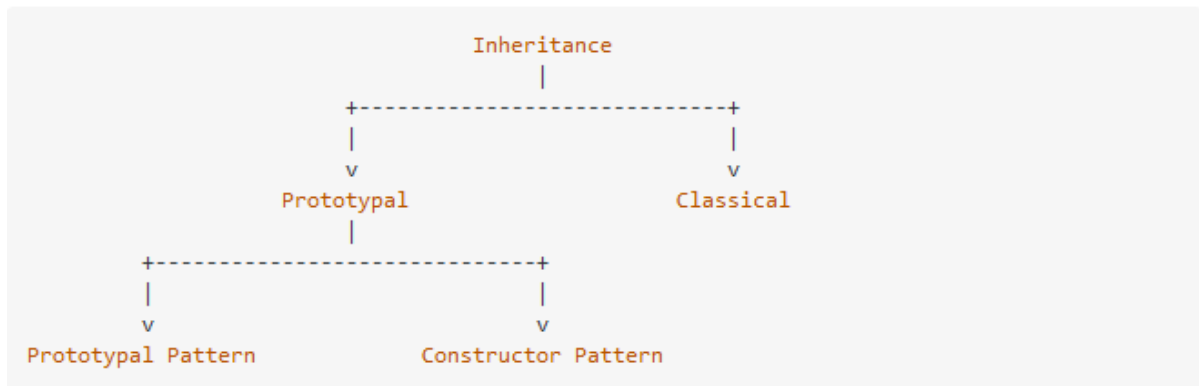
object.__proto__ == Object.prototype

array.__proto__ == Array.prototype

function.__proto__ == Function.prototype

Ways to Implement Inheritance

- Pseudo classical
- Prototypal



Constructor pattern or Pseudoclassic pattern

1. Create constructor function.
2. Create properties using this keyword.
3. Add functions to the prototype of the constructor function.
4. Create new object using “new” keyword.
5. Use `SubFunction.prototype=Object.create(“SuperFunction.prototype”);` for inheritance chaining
6. Use `SuperObject.call(this,params)` to instantiate the properties for SubObject

Every function has a property “prototype” which points to an object let say “prototype_object” which constructor property points to itself, when we use the constructor function, JavaScript internally points the `__proto__` property of the newly created object to that “prototype_object”. Hence all instances created through the constructor function have the same “prototype_object”. Therefore, we should add the function to constructor function using prototype property instead of creating the function inside constructor function. This way we can reduce memory uses. But the downside of this approach cannot implement the abstraction (data hiding).

Prototypal Pattern

1. without using “new” keyword
2. doesn’t support abstraction or hiding at all
3. use `Object.create` to make inheritance chaining

<https://medium.com/@eamonocallaghan/prototype-vs-proto-vs-prototype-in-javascript-6758cadcbae8>

//Don’t use `__proto__` property to set prototype

The `__proto__` is considered outdated and somewhat deprecated (in browser-only part of the JavaScript standard).

The modern methods are:

- `Object.create(proto, [descriptors])` – creates an empty object with given `proto` as `[[Prototype]]` and optional property descriptors.
- `Object.getPrototypeOf(obj)` – returns the `[[Prototype]]` of `obj`.
- `Object.setPrototypeOf(obj, proto)` – sets the `[[Prototype]]` of `obj` to `proto`.

These should be used instead of `__proto__`.

```
var asim = Object.create(Animal, {food: {value: "foo"}});
```

*Every object has a constructor property that references the function that is used to create the object

```
Let object = {} // new Object();
```

```
Let number = 10; // new Number();
```

```
Let string = "10" // new String();
```

```
Let boolean = true // new Boolean();
```

Functions are objects; Every function object has its constructor function "Function", internally a Function Statement can be expressed as given below.

The image shows a code editor with the following JavaScript code:

```
function Circle(radius) {  
  this.radius = radius;  
  this.draw = function() {  
    console.log('draw');  
  }  
}  
  
const Circle1 = new Function('radius', `  
  this.radius = radius;  
  this.draw = function() {  
    console.log('draw');  
  }  
`);  
  
const circle = new Circle1(1);  
const another = new Circle(1);
```

On the right, the runtime output is shown:

```
> circle  
< {radius: 1, draw: f}  
  ▶ draw: f ()  
    radius: 1  
  ▶ __proto__: Object  
> |
```

1. Encapsulation:
 - wrapping property and method along with in class
 - Benefit: Reduce Complexity + Increased Reusability
2. Abstraction:
 - Hide details and complexity and show only necessary thing outside
 - Benefits: Isolate impact of change
3. Inheritance:
 - Inheriting other objects property
 - Benefits: Eliminate redundant code
4. Polymorphism:
 - Many Form, same methods behave different for different objects
 - Benefits: Remove ugly switch-case

Getter/setter

```
let defaultLocation = { x: 0, y: 0 };

this.getDefaultLocation = function() {
  return defaultLocation;
};

this.draw = function() {
  console.log('draw');
};

Object.defineProperty(this, 'defaultLocation', {
  get: function() {
    return defaultLocation;
  },
  set: function(value) {
    defaultLocation = value;
  }
});
}
```

Promises:

Introduced in ES6

Then take two parameters, resolve handler and error handler

Promises are async by default

“throw” will call reject

Then always return promises

Promise.all([promise1, promise2, promise3,]);

Promise.resolve(“”)

Promise.reject(“”)

.then((resolveHandler, rejectHandler)=>{})

.catch(err=>{})

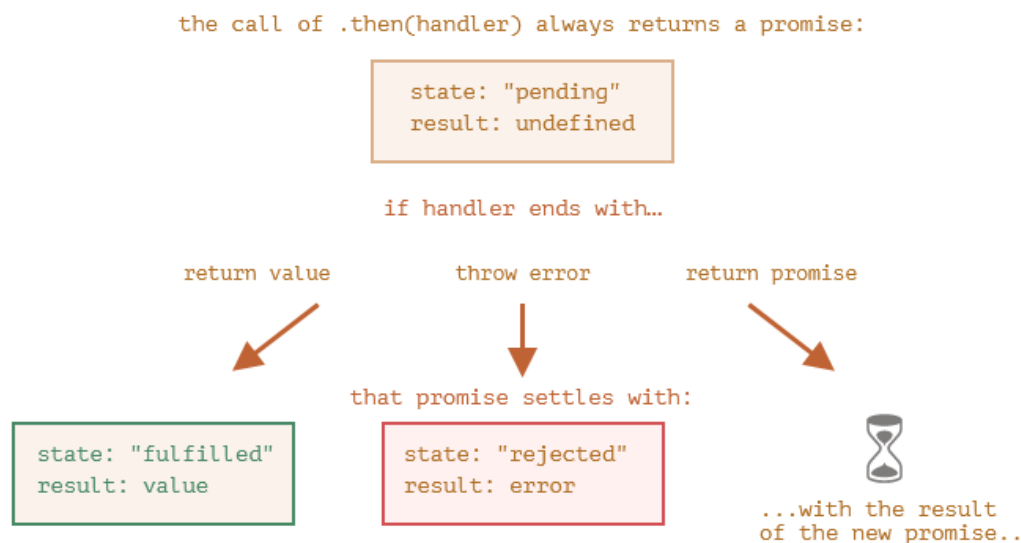
.finally(()=>{})

Return value

Once a `Promise` is fulfilled or rejected, the respective handler function (`onFulfilled` or `onRejected`) will be called **asynchronously** (scheduled in the current thread loop). The behavior of the handler function follows a specific set of rules. If a handler function:

- returns a value, the promise returned by `then` gets resolved with the returned value as its value.
- doesn't return anything, the promise returned by `then` gets resolved with an `undefined` value.
- throws an error, the promise returned by `then` gets rejected with the thrown error as its value.
- returns an already fulfilled promise, the promise returned by `then` gets fulfilled with that promise's value as its value.
- returns an already rejected promise, the promise returned by `then` gets rejected with that promise's value as its value.
- returns another **pending** promise object, the resolution/rejection of the promise returned by `then` will be subsequent to the resolution/rejection of the promise returned by the handler. Also, the resolved value of the promise returned by `then` will be the same as the resolved value of the promise returned by the handler.

Fork vs Chaining



Async/await

"await" can only be used inside "async" function

"await" causes non-blocking code to act like blocking one(provides synchronicity)

"async" function returns promise

Same Origin Policy:

CORS: cross origin resource sharing

Get method.

- Browser blocks response from different origin
- Browser always sends origin with the request, upon receiving the response, if response header "Access-Control-Allow-Origin" has the same requested origin or *, then response will propagate else get blocked.

Put/Post/Delete

- Browser sends pre-flight request[OPTIONS method] with headers, "Access-Control-Allow-Origin:moo.com" and "Access-Control-Request-Method: put"
- Server returns "Access-Control-Allow-Origin:moo.com or *", "Access-Control-Allow-Methods:put" if origin and methods match, Browser will issue the put request else block the put request.

JSONP:

Allow to bypass the same origin policy, only works with get requests.

same origin policy doesn't work for script tags hence we can point any APIs of get method type, and if response is in *jsonp* format, we can receive the actual response in *processresponse* function.

Every event occurred starts looking for handler from Root to Target and then move backwards from Target to Root.

Event Capturing Phase : Root to target.

Event Bubbling Phase : Target to root

By default, *addEventListener* works for "Event bubbling phase", optional parameter specify phase.

True for Capturing phase , False for Bubbling Phase.

StopPropagation vs PreventDefault

what are some key features of each ECMAScript version?

*****ECMAScript 5 (ES5 - 2009):*****

- Strict Mode: A stricter mode of JavaScript that helps catch common coding mistakes and "silent" errors.
- JSON (JavaScript Object Notation): Native support for parsing and stringifying JSON data.
- `Object.defineProperty()`: Allows fine-grained control over object property behavior.
- `Array` Methods: Introduction of several useful array methods like `forEach`, `map`, `filter`, `reduce`, etc.
- `bind()`: The `bind` method allows you to create a new function with a specified `this` value and initial arguments.

*****ECMAScript 6 (ES6 / ES2015):*****

- Arrow Functions: A concise syntax for writing functions.
- Classes: Introduces class syntax and inheritance.
- `let` and `const`: Block-scoped variable declarations.
- Template Literals: A way to embed expressions inside string literals.
- Destructuring: Extract values from arrays or objects using a concise syntax.

- Spread and Rest Operators: ``...`` used for expanding elements in arrays/objects or gathering function arguments.
- Promises: A better way to handle asynchronous operations.
- Modules: Native support for modularizing code.

ECMAScript 7 (ES7 / ES2016):

- Exponentiation Operator (``**``): A concise way to calculate exponentiation.
- ``Array.prototype.includes()``: A method to check if an array includes a certain element.

ECMAScript 8 (ES8 / ES2017):

- Async/Await: Simplified asynchronous code with ``async`` and ``await`` keywords.
- `Object.values()` and `Object.entries()`: Methods for getting object values and key-value pairs.
- String Padding: Methods to pad strings with characters to a specified length.
- Shared Memory and Atomics (for multi-threading): Introduced shared memory and atomic operations for better control over concurrent operations.

ECMAScript 9 (ES9 / ES2018):

- Asynchronous Iteration: Allows asynchronous operations within loops.
- Rest/Spread Properties: Improved object and array manipulation.
- `Promise.prototype.finally()`: A method to run code regardless of the promise's outcome.
- ``for-await-of``: A way to loop through asynchronous iterators.

ECMAScript 10 (ES10 / ES2019):

- ``Array.prototype.flat()`` and ``Array.prototype.flatMap()``: Methods for flattening and mapping arrays.
- ``Object.fromEntries()``: Converts a list of key-value pairs into an object.
- ``String.prototype.trimStart()`` and ``String.prototype.trimEnd()``: Methods for removing whitespace from the beginning or end of a string.

ECMAScript 11 (ES11 / ES2020):

- ``BigInt``: A built-in object for arbitrary-precision integers.
- Optional Chaining: A safer way to access deeply nested properties.
- Nullish Coalescing Operator (``??``): A way to provide default values only for null or undefined values (not falsy values).

ECMAScript 12 (ES12 / ES2021):

- ``Promise.any()``: Resolves the value of the first resolved promise.
- Logical Assignment Operators (``&&=``, ``||=``, ``??=``): Combines logical operations with assignment in a concise way.
- ``String.prototype.replaceAll()``: Replaces all occurrences of a substring within a string.

ECMAScript 13 (ES13 / ES2022):

- ``Array.prototype.at()``: Access elements in an array using a more versatile method.
- ``String.prototype.matchAll()``: Returns an iterator of all matched substrings.