General Characteristics

- JS is case sensitive.
- JS is whitespace insensitive.
- JS is dynamically (loosely) typed (vs strongly typed).

Data types

The latest ECMAScript standard defines nine types:

- Six **Data Types** that are primitives, checked by `typeof` operator:
  - undefined : `typeof instance === "undefined"`
  - Boolean : `typeof instance === "boolean"`
  - Number : `typeof instance === "number"`
  - String : `typeof instance === "string"`
  - BigInt : `typeof instance === "bigint"`
  - Symbol : `typeof instance === "symbol"`
- null : `typeof instance === "object"`. Special primitive type having additional usage for its value: if object is not inherited, then null is shown. Because JavaScript is case-sensitive, null is not the same as Null, NULL, or any other variant;
- Object : `typeof instance === "object"`. Special non-data but structural type for any constructed object instance also used as data structures;
- Function : a non-data structure, though it also answers for typeof operator: `typeof instance === "function"`. This is merely a special shorthand for Functions, though every Function constructor is derived from Object constructor.

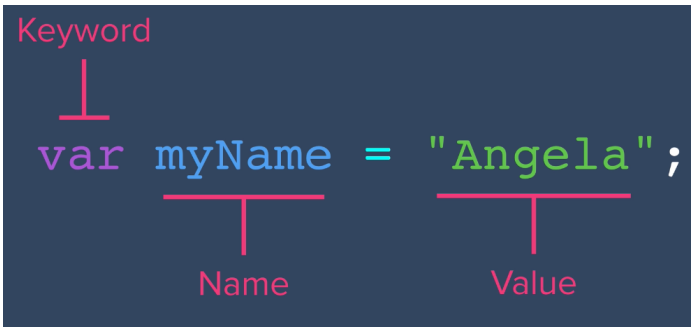
Keep in mind the only valuable purpose of `typeof` operator usage is checking the Data Type. If we wish to check any Structural Type derived from Object it is pointless to use `typeof` for that, as we will always receive "object". The indeed proper way to check what sort of Object we are using is `instanceof` keyword. But even in that case there might be misconceptions.

Note in case of `const name = "Ben", name instanceof String` will evaluate to false because strings are primitive and they don't have a prototype. So, `instanceof` is always searching the prototype chain to evaluate the expression.

All types except objects define immutable values (values, which are incapable of being changed). For example and unlike C, Strings are immutable. We refer to values of these types as "primitive values". For example, we can not change `'Ben'` by doing the following:

```
const myName = 'Ben';
myName[0] = 'Z';
```

Variables



Object: store a value in memory and be able to get hold of it whenever you need it.

Naming conventions

Variable names can only start with alphabets, $, _ and they should be camelCase.

Type coercion

Type coercion is the automatic or implicit conversion of values from one data type to another (such as strings to numbers). *Type conversion* is similar to *type*

coercion because they both convert values from one data type to another with one key difference — *type coercion* is implicit whereas *type conversion* can be either implicit *or* explicit.

https://dorey.github.io/JavaScript-Equality-Table/

```
const value1 = '5';
const value2 = 9;
let sum = value1 + value2;
```

sum will be 59

=== vs ==

Falsy values

The following values evaluate to false (also known as Falsy values):

- false
- undefined
- null
- 0
- NaN
- the empty string ("")

```
var b = new Boolean(false);
if (b) // this condition evaluates to true
if (b == true) // this condition evaluates to false
```

String

Properties

length

Static Methods

```
// returns a string created from the specified sequence of UTF-16 code units.
String.fromCharCode()
```

Methods

```
myName[i]

.toUpperCase();
.toLowerCase();

.split(" ");

.trim();

.indexOf("groovy");
// beginning position; returns -1 if not found

.lastIndexOf("Ben");
// finds the last occurrence of Ben if exists and returns the index of first character



.slice(6, 11);
// starting position (zero based) and ending position (not including), it is not changing the original string

.substring(start, end);

.substr(start, length);

.replace(old, new);

// returns an integer between 0 and 65535 representing the UTF-16 code unit at the given index.
.charCodeAt(index)

.charAt(index)}
```

ES6

```
.startsWith('J', ?position);
// determines whether a string begins with the characters of a specified string, returning true or false as appropriate.


.endsWith('th', ?position);
```

// determines whether a string ends with the characters of a specified string, returning `true` or `false` as appropriate.

```
.includes('th', ?position);
```

//determines whether one string may be found within another string, returning `true` or `false` as appropriate.

```
.repeat(n);
```

Assign Operator

What is going on when we do `x = x + 1`?

```
var y = x++;
var y = ++x;
```

Number

```
'37'- 7 = 30
var x = 100 / "10";     // x will be 10

var foo = 5;
var bar = "5";
     foo + bar = "55"

var foo = 5;
var bar = "b";
     foo + bar = 5b

var foo = "55";
var myNumber = Number(foo);

if (isNaN(myNumber)) {
}

var x = 200.6;
var y = Math.round(x);
var y = Math.max(x);

var y = Math.pow(base, exponent);
```

```
var y = Math.abs();

var z = Math.floor(x)
// returns the largest integer less than or equal to a given number

var z = Math.ceil(x)
// returns the smallest integer greater than or equal to a given number

var z = Math.sign(x)
// function returns 1, 0, or -1

var z = Math.random();
// it generates a 16 Decimal place number like 0.1111111111111111

parseFloat()

parseInt()
// ignores characters that are not numeric and appearing after a numeric value

toFixed(number-of-digits)
```

Functions

Functions are used to refactor a repeated block of code to a piece that can be called repeatedly.

All functions are objects, instances of `Function`. Therefore, functions get their methods from Function.prototype. That is

`fac instanceof Function` evaluates to true

Function Declarations

```
function calcRectArea(width, height) {
  return width * height;
}
```

A function declaration is neither a statement nor an expression. A function declaration can be called earlier than it is defined. It declares a new variable, creates a function object, and assigns it to the variable.

Function Expressions

(Anonymous) function expressions

A function expression produces a value—a function object:

```
function(x, y) {
      return x + y;
}
```

And we can pass it as an argument to another function. A function expression can be assigned to a variable as follows and by doing so the whole thing becomes a statement:

```
const sum = function (x, y) {
      return x + y;
};
```

Named function expressions

We can give a function expression a name. *Named function expressions* allow a function expression to refer to itself, which is useful for self-recursion:

```
let meFunction = function me(n) {
      if (n > 0) {
            return n * me(n-1);
      } else {
            return 1;
      }
}
```

The accessibility of function expressions assigned to variables will be determined by the life cycle of the variable.

In JavaScript function can be a property of an object:

```
      let obj = {
            square: function (a) {
```

```
            return a*a;
        }
    }

    let obj = {
        square(a) {
            return a*a;
        }
    }
```

Higher Order Functions

Higher order functions are functions that can take other functions as inputs.

Callback Functions

Control Statements: if-else

Case1:

```
if (condition) {
    Eat apple;
} else {
    Eat orange;
}
```

Case2:

```
if (condition1) {
    Eat apple;
```

```
} else if (condition2) {

} else {

}
```

Case3:

```
if (condition1) {
      Eat apple;

} else if (condition2) {

}
```

// Order matters

```
&&

||

!
```

Control Statements: switch

```
switch (expression) {
      case expression:

            break;


      default:

}
```

Control Statements: While Loops

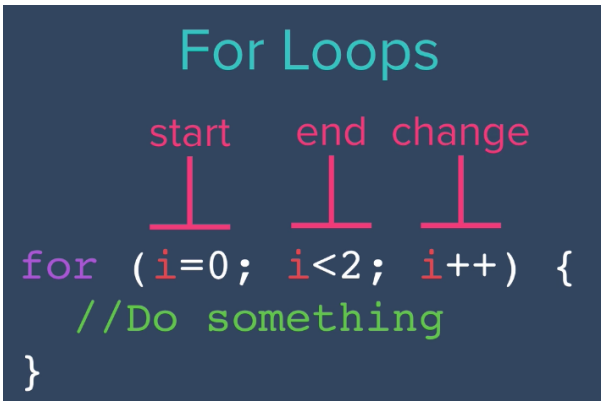
I.

```
while (condition) {
      ...
};
```

II.

```
do {
      ...
} while (condition);
```

Control Statements: For Loops




```
for (var i= 1; i < 100; i++) {
      if (i % 5 == 0) {
            continue;
            break;
      }
}

fields.forEach((current, index, array) => {

});

for (let item of array) {

}

for (let [i, value] of array.entries) {
```

```
}
```
// can be used for any iterable like array or string. We can also use break and continue inside this loop

```
for (let key in object) {

}
```
// can be used to loop over keys of properties of an object

Arrays

```
var multipleValues = [];
var multipleValues = new Array();
var multipleValues = new Array(50, 60, "Hello");
var multipleValues = new Array().fill(0).map(() => []);
var multipleValues = [50, 60, "Hello", true];
multipleValues[0] = 50;
```

Properties

```
length
```

Methods (functions that belong to an object)

```
multipleValues.includes(elementToSearch, indexToStart);

multipleValues.push('a');   // Add to the end - O(1)
multipleValues.unshift('x');   // Add to the beginning - O(n)

multipleValues.pop();        // Remove from the end - O(1)
multipleValues.shift();      // Remove from the beginning O(n)

multipleValues.slice(startIndex, excludedFinishIndex);
```
// copies a given part of an array and returns that copied part as a new array. It doesn't change the original array. It also can be used for String.
We can pass a list and it will return an array:

```
Array.prototype.slice.call(fields);

multipleValues.slice(2);
```
// Every element starting from index 0 till index 2 (excluding) will be removed

```
multipleValues.splice(startIndex, deleteCount,
itemToBeAdded);
```
O(n)
// changes the original array and returns an array of removed elements

```
multipleValues.splice(2);
```
// Every element starting from index 2, will be removed

```
multipleValues.indexOf('x');

multipleValues = multipleValues.concat(array2);
```
// O(n)

```
multipleValues.push(...array2);

multipleValues.reverse();
```

// creates and returns a new string by concatenating all of the elements in an array, separated by commas or a specified separator string. If the array has only one item, then that item will be returned without using the separator. It doesn't change the original array.

```
multipleValues.join();

multipleValues.sort();

let newMultipleValues = multipleValues.filter(e => e.name
!= 'john');
```
// returns a new array leaving the original array unchanged

```
multipleValues.find();
```
// returns the first element in the array that satisfies the condition. Returns `undefined` if no elements found

```
const sum = multipleValues.reduce((accumulator,
currentElement, index) => accumulator + currentElement,
<initial_value_to_start_from>);
```
// We use reduce when 1. we want to end up with one value at the end; 2. we want to persist the result of the operation after each iteration.

```
const sum = multipleValues.every((current, index, array)
=> {
```

```
});
```
// method determines whether *all* elements of the array match the predicate. the execution of every() is short-circuited. As soon as every() finds an array element that doesn't match the predicate, it immediately returns false and doesn't iterate over the remaining elements.

```
const sum = multipleValues.some((current, index, array) =>
{

});
```
// method determines whether *at least one* element of the array matches the given predicate. It only returns false if *none* of the array elements match the predicate. Note that some() stops iterating over the array as soon as it finds an element that matches the predicate. In that case, it immediately returns true without inspecting the remaining elements.

Objects

Everything in the world (including things in JavaScript) is an object.

```
const myObject = new Object({name: "Ben", age: 46})
Object.keys(obj)
```
// returns an array of keys - O(N)

```
Object.values(obj)
```
// returns an array of values - O(N)

```
Object.entries(obj)
```
// returns an array of arrays [key, value] - O(N)

```
obj.hasOwnProperty("firstName");
```
// returns true/false - O(1)

```
let myObj = {1: 23, 2: 53};
if (1 in myObj)

delete myObj[index];

Object.freeze(myObj);

Object.create(existingObj;
```

// creates a new object, using an existing object as the prototype of the newly created object.

Scope (Name space)

Visibility

Global vs Local

Defined by a function

```
function foo() {
     var test = 1;
}

console.log(test);
```

Variables defined by `var` are function-scoped that is, they are just available inside that function block.

Defined by a block

```
{
}

if () {
}

for () {
}
```

Variables defined by `let` and `const` are block-scoped that is, they are just available inside that block. The other difference is that variables declared using `let` are not hoisted. Moreover, when we declare a variable using `let` in the global scope it doesn't get attached to the `window` object.

When a variable is being declared using `const`, it should be assigned at the same time.

What that means is that variables defined by `var` inside a if block in a function is still available outside that block within that function.

this

The question is to which object ***`this` keyword*** refers to.

Global context

In the global execution context (outside of any function), this refers to the global object, whether in strict mode or not, that is `window` in browsers.

Function context

Inside a function, the value of `this` depends on how that function is called. Regular functions use the caller to determine what is inside `this` keyword.

Simple call

Regular Functions

In not strict mode:

```
function f1() {
      return this;
}

f1() === window;
// In a browser:

f1() === global;
// In Node:
```

In strict mode:

```
function f2() {
      return this;
}

f2() === undefined;
```

Arrow Functions

Arrow functions make use of lexical `this` (they are lexically scoped). That is, they bind the value of `this` to the value of `this` of the caller.

Call while bounded to an object

```
let person = {
      firstName: 'Nathan',
      lastName: 'Pearson',
      fullName1: function () {
            console.log(this);
      },
      fullName2: () => {
            console.log(this);
      }
}

person.fullName1();
// when JS engine reaches the above line since a function is getting invoked a new execution context gets created (i.e. a new callee). Since this function is being invoked as a method of an object, the value of this is set to that object.

person.fullName2();
// when JS engine reaches the above line since a function is getting invoked a new execution context gets created (i.e. a new callee). Since an arrow function is being used, this is not bound at all. It just inherits from the parent execution context (caller). That is, this refers to this of the caller.
```

Lexical scope refers to the fact that the nested functions are lexically bound. They are bound to the scope of their parents or of their

outer functions. But, it is a one way relationship; it doesn't work the other way around.

"use strict"

When placed in a JS file:

- We cannot use a variable before declaring it;
- `this` refers to undefined inside regular functions;

Mutable vs Immutable

In JavaScript primitive data type values are immutable, that is we cannot change the value of primitive value types.

In contrast, Objects and Arrays are mutable. That is, we can change their values.

Execution Stack/Context

Host (browser/node) has a JavaScript Engine (V8, JavaScriptCore, ChakraCore, SpiderMonkey). Takes the JS code and executes it.

First JS code gets parsed by a parser to check the syntax. If the syntax is correct the parser creates a data structure called Abstract Syntax Tree. Then AST gets translated to machine code. Last, the machine code gets executed.

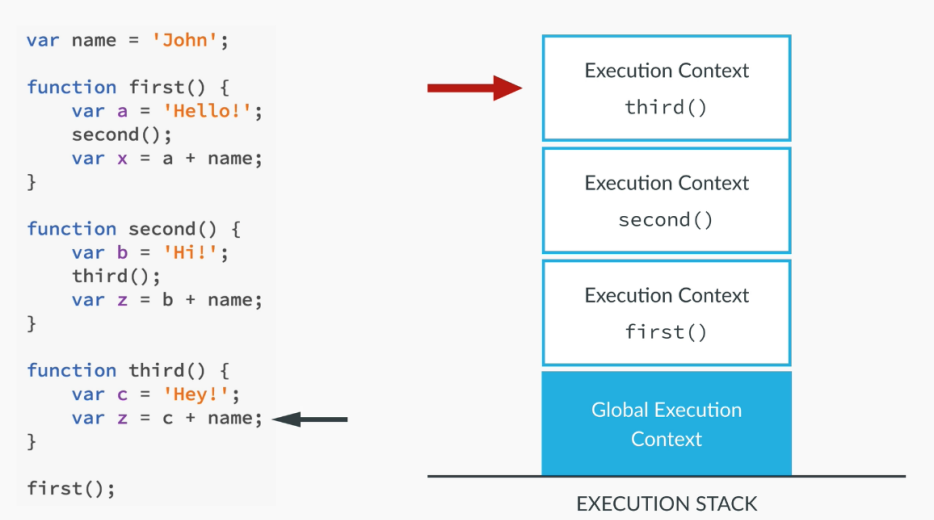
All JS codes need to run in environments. These environments are called Execution Context. An execution context is a container or wrapper which stores variables and in which a piece of our code is evaluated and executed.

The default execution context is Global Execution Context. That is, code that is not inside any function and associated with the global object (in the browser, that's the window object).

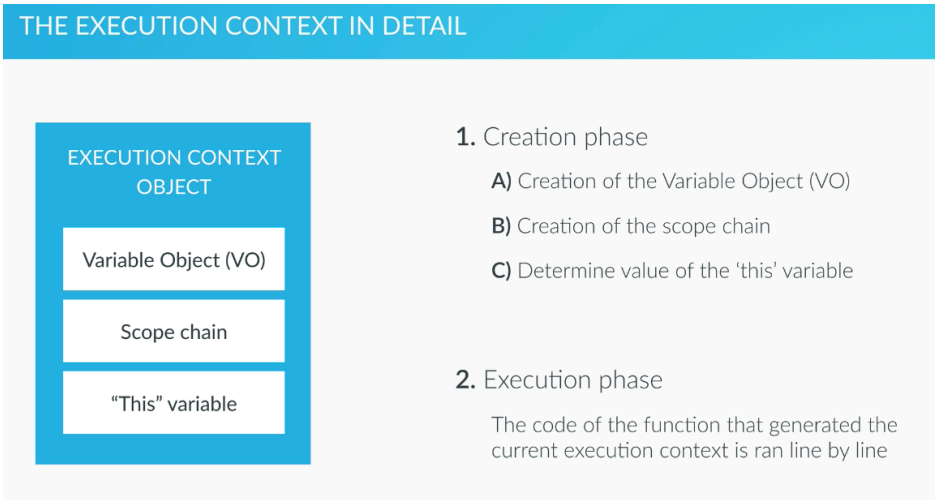
A context which creates (*calls*) another context is called a *caller*. A context which is being created is called a *callee*.When a caller creates a callee, the caller suspends its execution and passes the control flow to the callee. The callee is pushed onto the stack and becomes a running (active) execution context. After the callee's code is executed entirely, it returns control to the caller, and the

evaluation of the caller's context proceeds (it may activate other contexts) till its end, and so on.

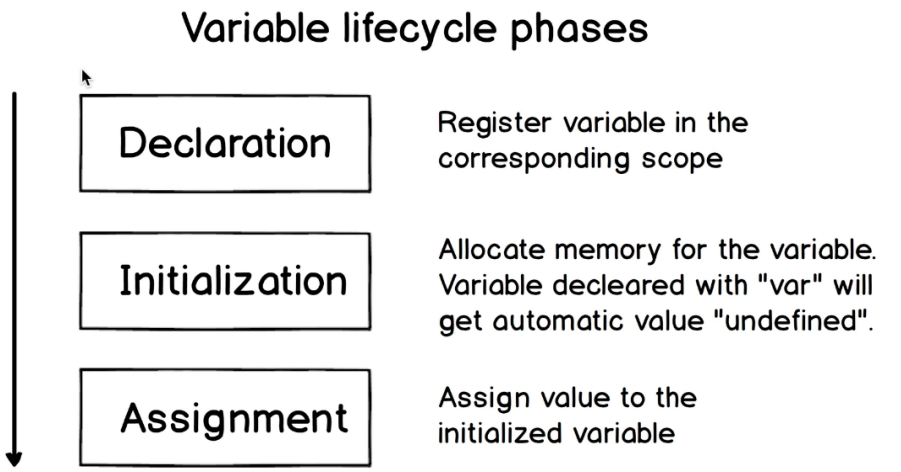
Everytime a function is invoked, a new `execution context` is created and gets pushed to the top of the `execution stack`.

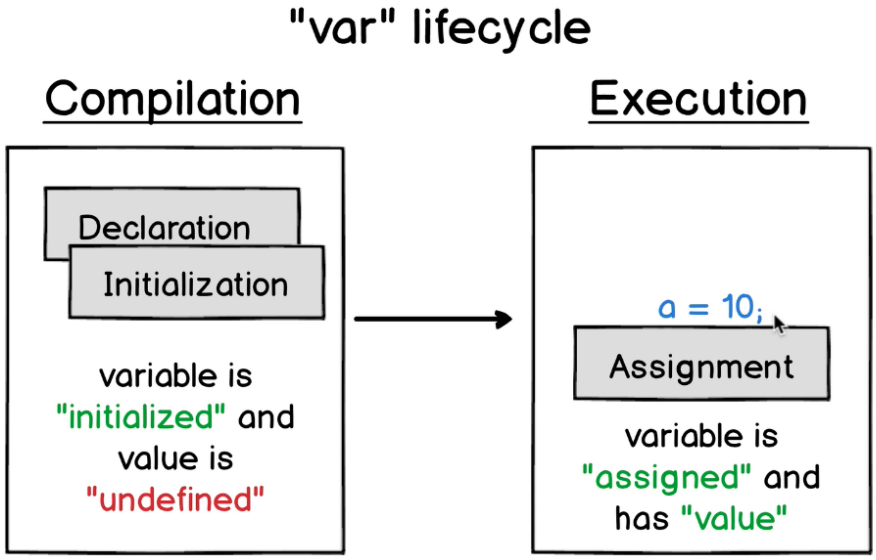


However, inside the JavaScript interpreter, every call to an execution context has two phases; Creation phase (sometimes called Compilation phase) and Execution phase. The key here is to pay attention to what happens in each phase.

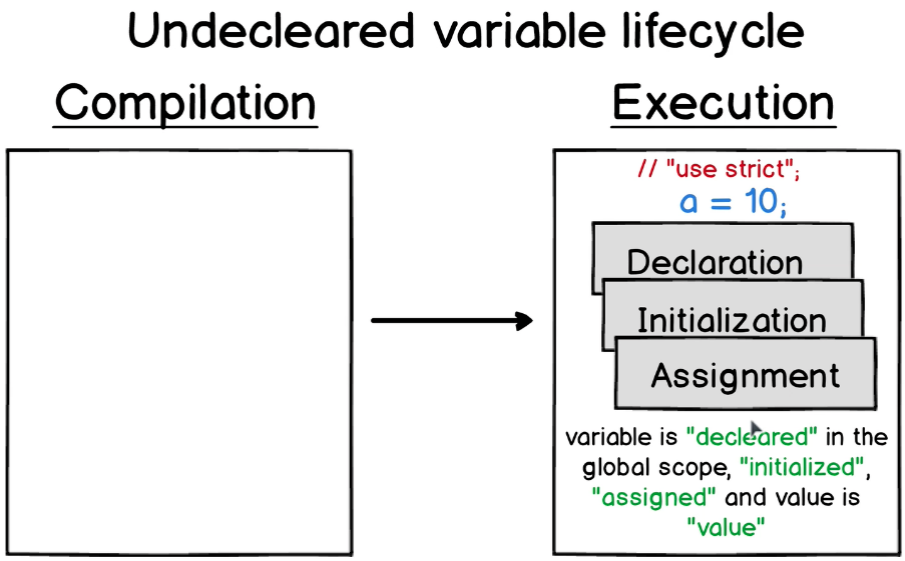


Now, before going through what happens in each phase, we first look at variable and function lifecycle to see which ones happen in Creation phase and which ones happen in the Execution phase.





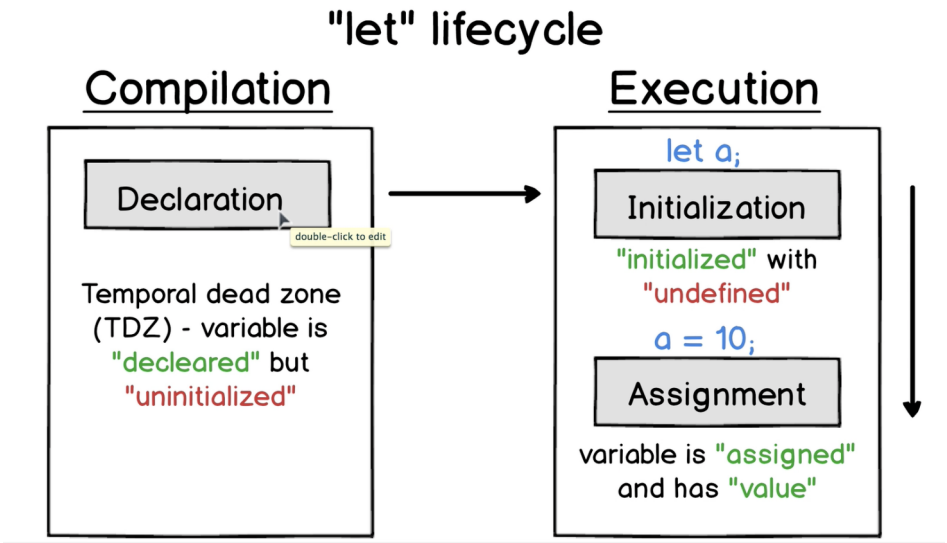
Note that the lifecycle of undeclared variables are quite different. That is, if JS engine cannot find its declaration in the chain scope, it declares, initializes, and assigns it in global scope:

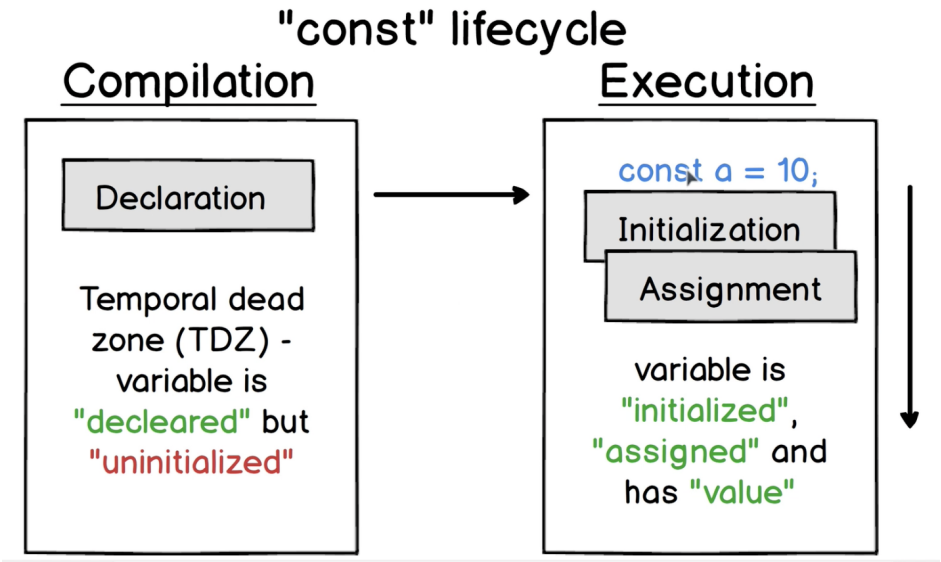



```
function fn() {
  function fn2() {
    console.log(a);
    a = 10;
  }

  fn2();
}

fn();
```

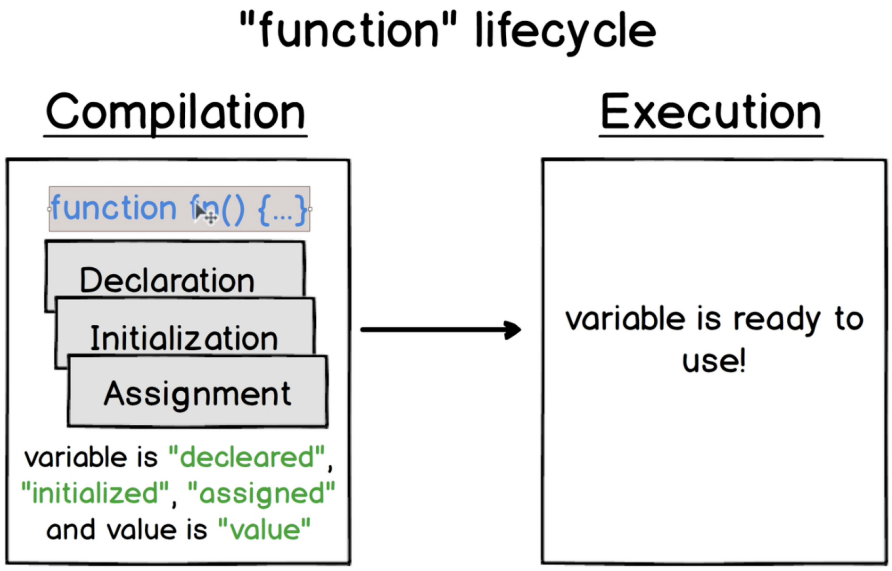
```
top
Uncaught ReferenceError: a is not defined
    at fn2 (index.js:3)
    at fn (index.js:7)
    at index.js:10
```

Moreover, the life cycle of variables declared using `let` is also different:



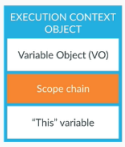


By now, you should have realized that a variable is accessible only when it gets initialized.

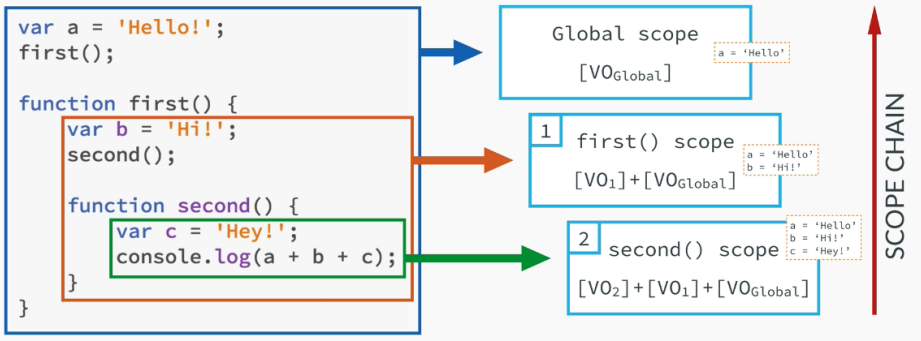


The value is the content of the function in string format.

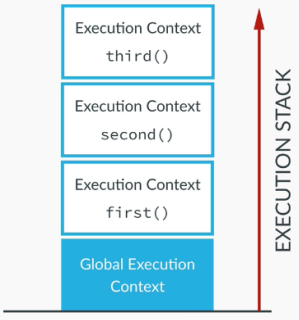
SCOPING IN JAVASCRIPT



- Scoping answers the question "where can we access a certain variable?"
- **Each new function creates a scope**: the space/environment, in which the variables it defines are accessible.
- **Lexical scoping**: a function that is lexically within another function gets access to the scope of the outer function.



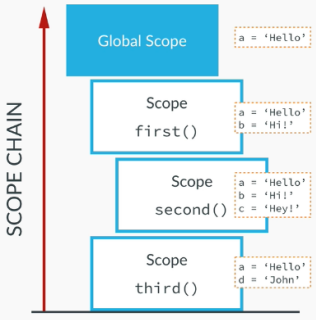
EXECUTION STACK VS SCOPE CHAIN



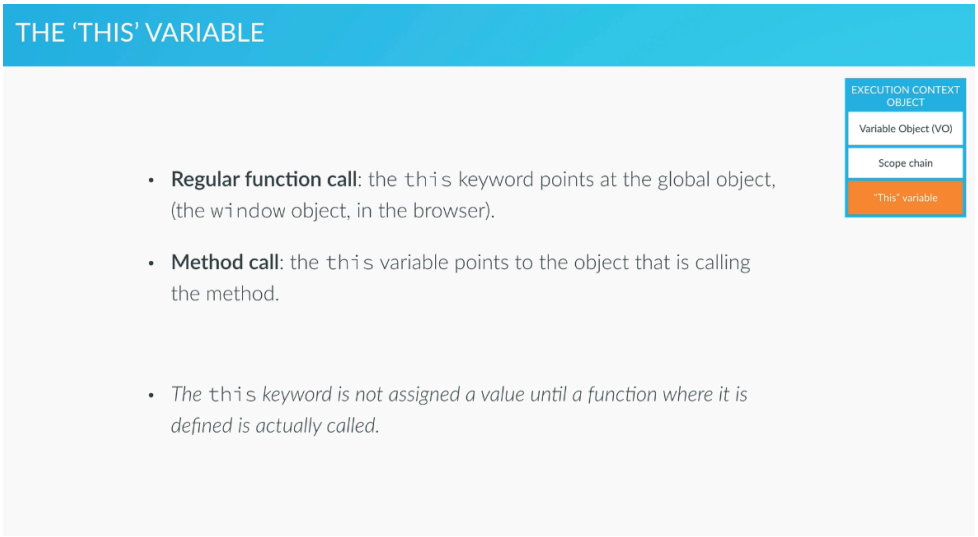
Order in which functions are **called**

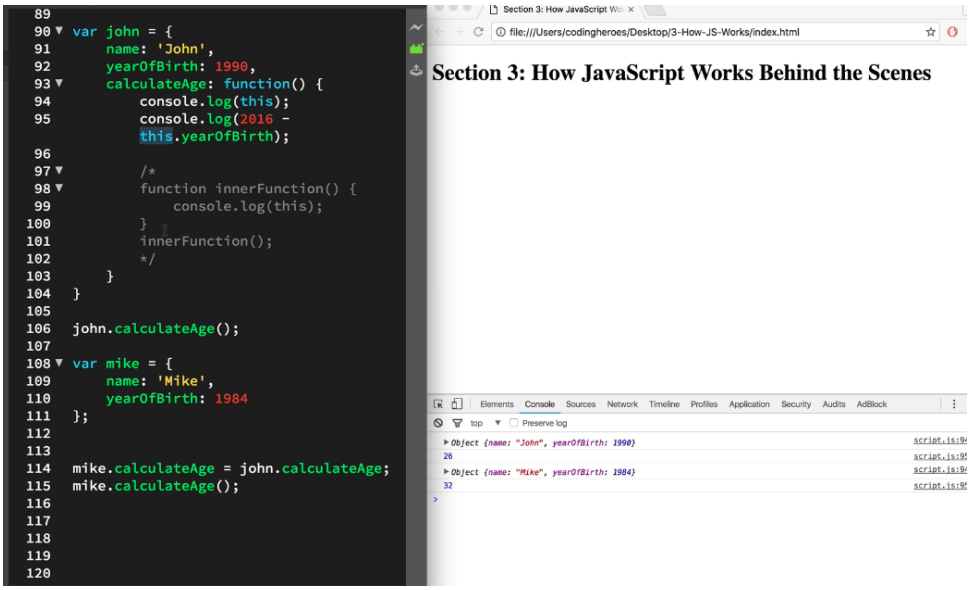
```
var a = 'Hello!';
first();

function first() {
    var b = 'Hi!';
    second();

    function second() {
        var c = 'Hey!';
        third()
    }
}

function third() {
    var d = 'John';
    console.log(a + b + c + d);
}
```

≠



Order in which functions are **written lexically**





Hoisting

Hoisting is just the process of initializing variable declarations with a default value and placing function declarations into memory during the creation phase. In other words, hoisting means where in the code we can access a variable or invoke a function before it has been declared lexically.

Function declarations are hoisted completely, variable declarations only partially, i.e. `var` declarations are hoisted, but only the declarations with undefined assignment and not final assignment.

Therefore, using a var declaration and a function expression similarly to the previous example results in an error:

```
foo();  // TypeError: undefined is not a function
var foo = function () {

    ...
};
```

However, `let` and `const` declarations are not hoisted.

Closure

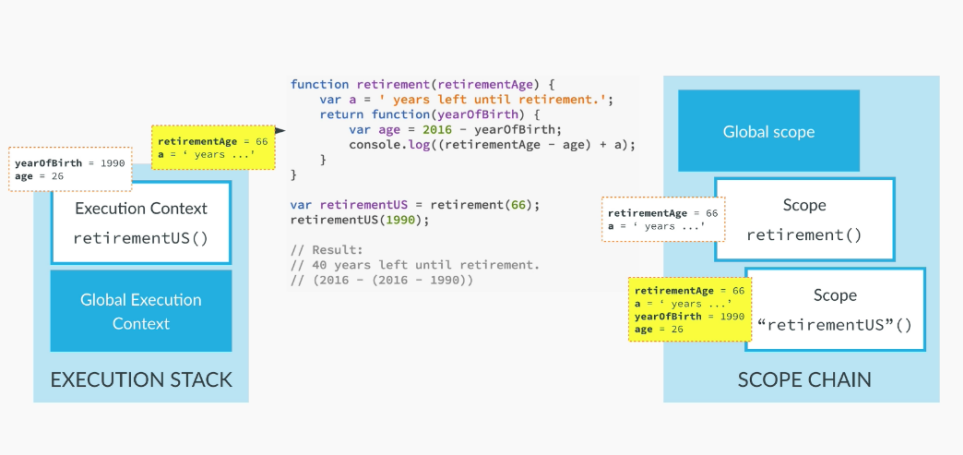
When a function gets declared in a creation phase, it not only contains a function definition but an object called closure. The closure is a collection of all the variables in scope at the time of declaring the function.

The closure becomes relevant when a function (referred to as outer function) returns another function (referred to as inner function). The returned inner function has access to the outer function's scope chain through its closure. The closure has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function's variables, and it has access to the global variables. Remember that the inner function has access not only to the outer function's variables, but also to the outer function's parameters.

The way I will always remember closures is through **the backpack analogy**. When a function gets created and passed around or returned from another function, it carries a backpack with it. And in the backpack are all the variables that were in scope when the function was declared.

```
function showName(firstName) {
    var nameInto = "Your name is";
```

```
      function makeFullName(lastName) {
            return nameInro + firstName + " " + lastName;
      }

      return makeFullName();
}

const mjName = showName("Michael");
mjName("Jackson");
```



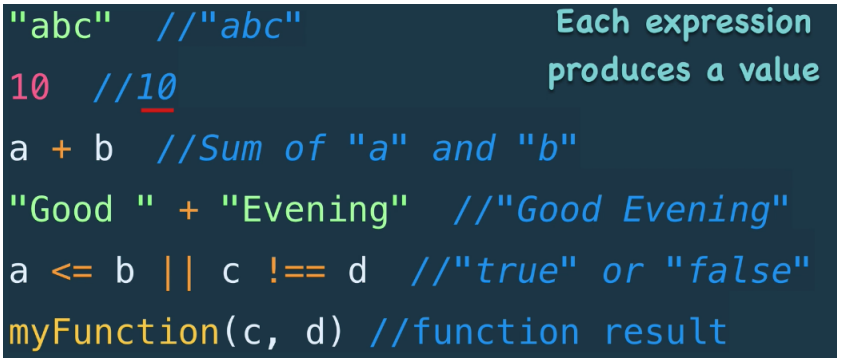
Important point to remember is that closures store references to the outer function's variables and they do not store the actual value. Closures get more interesting when the value of the outer function's variable changes before the closure is being used. And this powerful feature can be harnessed in creative ways, such as this private variables example first demonstrated by Douglas Crockford:

```
function celebrityID() {
      var celebrityID = 999;
      return {
            getID: function () {
                  return celebrityID;
            },
            setID: function (theNewID){
                  celebrityID = theNewID;
            }
      }
}

var mjID = celebrityID();
mjID.getID(); 999
```

```
mjID.setID(677);
mjID.getID(); 677
```

Expressions vs Statements

An expression produces a value and can be written wherever a value is expected.



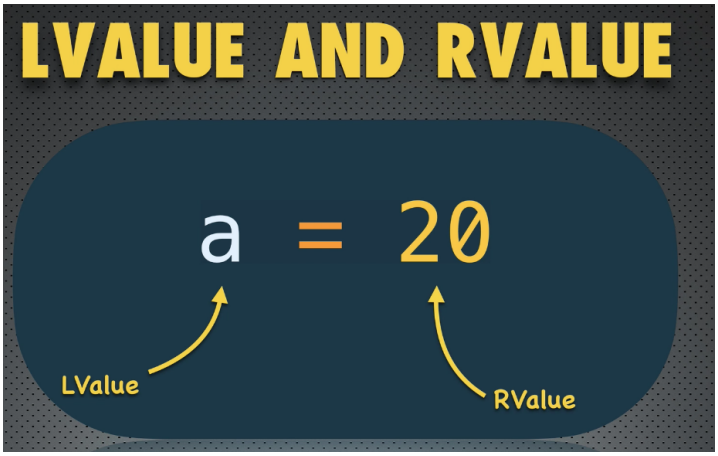
Keep in mind that if a function doesn't return a value explicitly it returns `undefined` by default. That's why we see `undefined` each time we call console.log().

Examples:

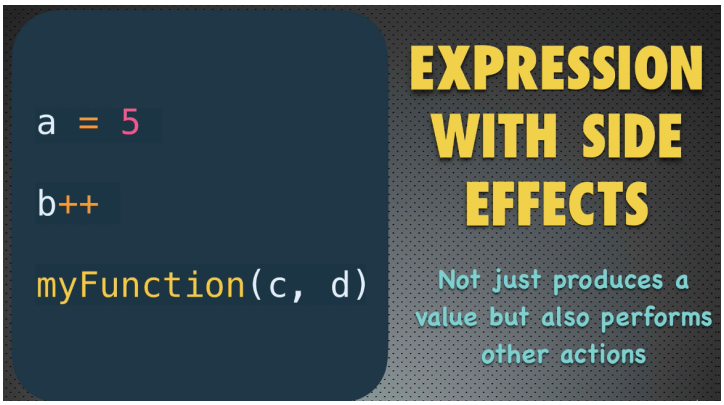
1. `delete myObject.x`

Assignment Expression

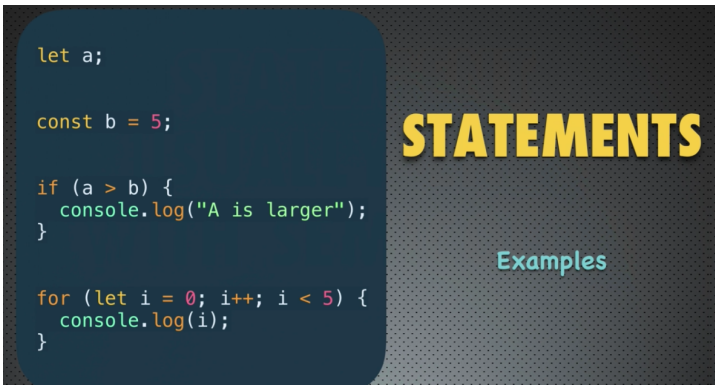
Each time that we use an assignment operator like `a = 10` and we assign a value to a variable, we are creating an assignment expression. It is an expression and not a statement because it produces the value that is assigned to the variable.



Expression with side effects



A statement performs an action. For example, an if statement cannot become the argument of a function.



Examples:

- `return …;`
- `var test = function myFunc() {`

  `};`

Note that a statement can contain an expression like the following statement:

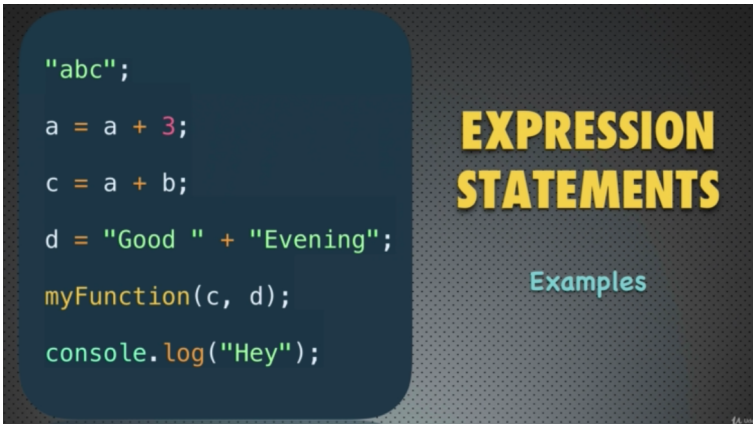
```
const b = 5;
```

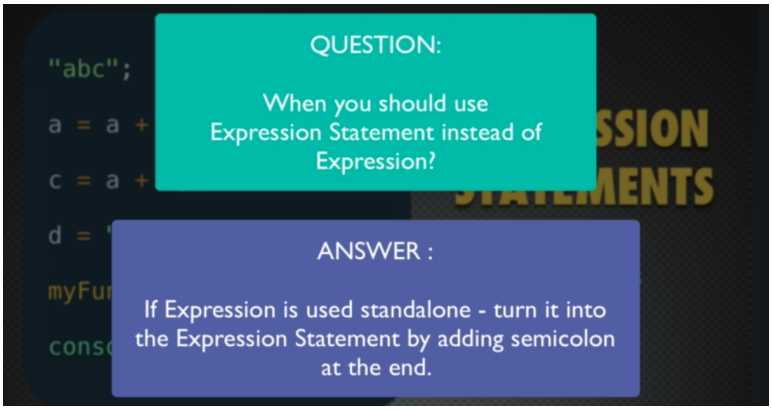
If we use a comma operator between two expressions. The operator evaluates both expressions and returns the result of the second one:

```
var x = ('a', 'b');
```
x has in this case the value of 'b'

Expression Statements

Expressions can be turned into statements by appending a semicolon:





Stand-alone blocks

The following code illustrates one use case for such blocks: we can give them a label and break from them

```
function test(printTwo) {
    printing: {
        console.log("One");
        if (!printTwo) break printing;
        console.log("Two");
    }
    console.log("Three");
}
```

Using ambiguous expressions as statements

Two kinds of expressions look like statements—they are ambiguous with regard to their syntactic category:

- Object literals (expressions) look like blocks (statements):

  ```
  {
      foo: bar(3, 5)
  }
  ```

  The preceding construct is either an object literal or a block followed by the label foo:, followed by the function call bar(3, 5).

- Named function expressions look like function declarations (statements):

  ```
  function foo() { }
  ```

> The preceding construct is either a named function expression or a function declaration. The former produces a function, the latter creates a variable and assigns a function to it.

In order to prevent ambiguity during parsing, JavaScript does not let you use object literals and function expressions as statements. That is, expression statements must not start with:

- ★ A curly brace
- ★ The keyword function

If an expression starts with either of those tokens, it can only appear in an expression context. You can comply with that requirement by, for example, putting parentheses around the expression. Next, we'll look at two examples where that is necessary.

Evaluating an object literal via eval()

eval parses its argument in statement context. You have to put parentheses around an object literal if you want eval to return an object:

```
eval('{ foo: 123 }')
123

eval('({ foo: 123 })')
{ foo: 123 }
```

Immediately invoking a function expression

The following code is an *immediately invoked function expression* (IIFE), a function whose body is executed right away:

```
(function () { return 'abc' }())
'abc'
```

If you omit the parentheses, you get a syntax error, because JavaScript sees a function declaration, which can't be anonymous:

```
function () { return 'abc' }()
SyntaxError: function statement requires a name
```

If you add a name, you also get a syntax error, because function declarations can't be immediately invoked:

```
function foo() { return 'abc' }()
SyntaxError: Unexpected token )
```

Whatever follows a function declaration must be a legal statement and () isn't.

Control Flow Statements and Blocks

For control flow statements, the body is a single statement. Here are two examples:

```
if (obj !== null) obj.foo();

while (x > 0) x--;
```

However, any statement can always be replaced by a *block*, curly braces containing zero or more statements. Thus, we can also write:

```
if (obj !== null)  {
    obj.foo();
}

while (x > 0) {
    x--;
}
```

Rules for Using Semicolons

The basic rules are:

- Normally, statements are terminated by semicolons.

- The exception is statements ending with blocks.

No Semicolon After a Statement Ending with a Block

The following statements are not terminated by semicolons if they end with a block:

- Loops: for, while (but not do-while)

```
while (a > 0) {
  a--;
} // no semicolon

do {
        a--;
} while (a > 0);
```

- Branching: if, switch, try
- Function declarations (but not function expressions). A function declaration is a statement.

Function Declaration:

```
function foo() {
  // ...
} // no semicolon
```

Expression function:

```
var foo = function () {
  // ...
};
```

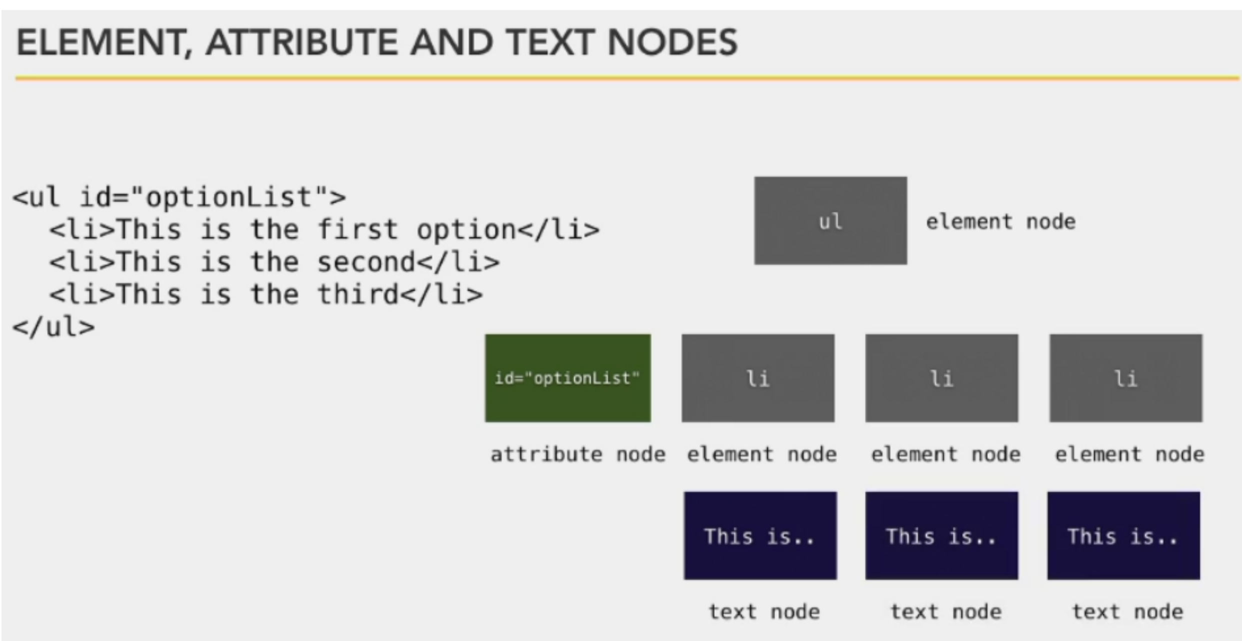
DOM

When a web page is loaded, the browser creates a **D**ocument **O**bject **M**odel of the page.

The Browser Object Model is a larger representation of everything provided by the browser including the current document, location, history, frames, and any other functionality the browser may expose to JavaScript. The Browser Object Model is not standardized and can change based on different browsers.

The Document Object Model is standardized and is specific to current HTML documents. It is exposed by the Browser Object Model.

The **window** object is supported by all browsers. It represents the browser's window. All global JavaScript objects, functions, and variables automatically become members of the window object; global variables are properties of the window object and global functions are methods of the window object. Even the document object (of the HTML DOM) is a property of the window object.



Methods to query elements

```
document.firstElementChild;

document.lastElementChild;

document.getElementById("optionList");

document.getElementsByTagName("li");

document.querySelector('#optionList');

document.querySelector('button');
// selects the first matching element

document.querySelectorAll('button');
// selects all matching elements and returns an array
```

Methods to get elements' attributes

```
myElement.nodeType

myElement.innerHTML (sometimes returns whole content)

myElement.childNodes

myElement.getAttribute("align");

myElement.value;
```

Methods to set elements' attributes

```
myElement.setAttribute("align", "right");

myElement.textContent = 6;

myElement.innerHTML = '<em>' + dice + '</em>';

element.classList.add('name-of-class');

element.classList.remove('name-of-class');
```

Methods to create new elements

```
var myNewElement = document.createElement("li");
myElement.appendChild(myNewElement);
myNewElement.innerHTML = "New item text";

Or

var myText = document.createTextNode("New list item text");
myNewElement.appendChild(myText);


var myNewElement = document.createElement("li");
var secondItem = myElement.getElementsByTagName("li")[1];
myElement.insertBefore(myNewElement, secondItem);

document.querySelector(element).insertAdjacentHTML('beforeend',
newHtml);
```

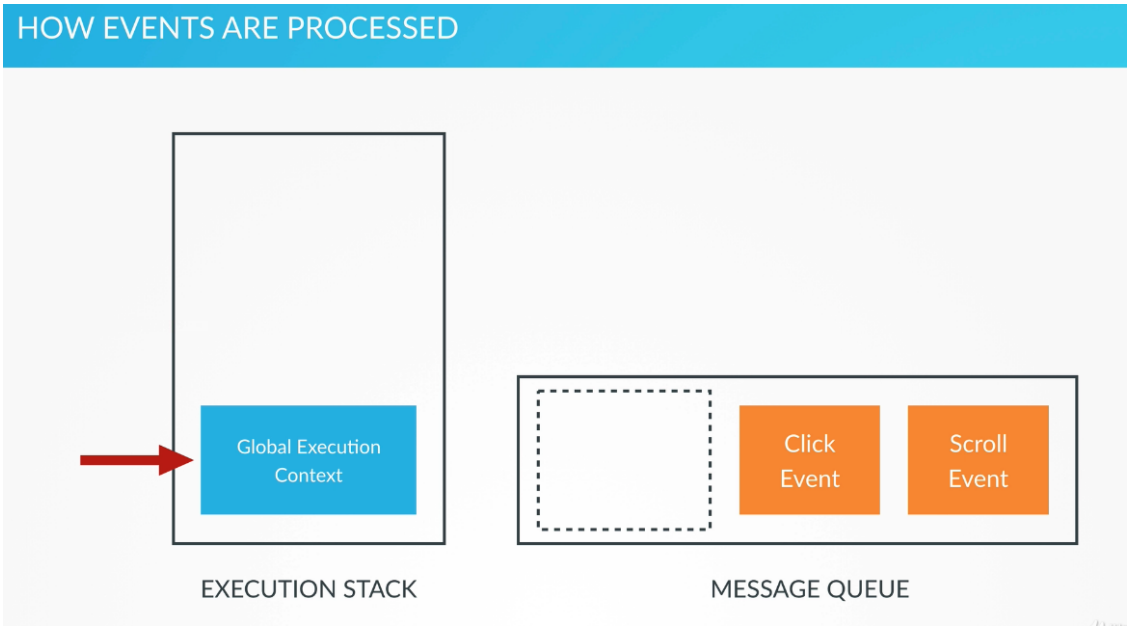
```
<!-- beforebegin -->
<p>
  <!-- afterbegin -->
  foo
  <!-- beforeend -->
</p>
<!-- afterend -->
```

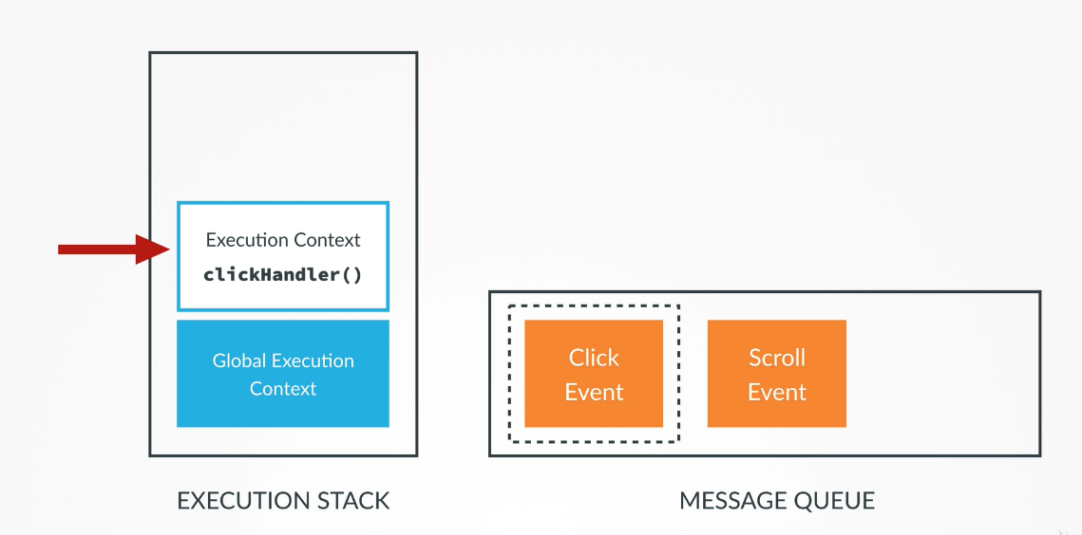
Methods to remove elements

```
element.parentNode.removeChild(element);
```

EVENTS and EVENT LISTENERS

When an event happens, it gets queued in MESSAGE QUEUE. Only when all execution contexts are being processed, the event listener of corresponding event is pushed to execution stack:





Handling events

Method 1:

Directly in html tag:

```
<button onclick="alert('Hello');" ></button>
```

```
<body onload=”alert(‘Hello’);” >
</body>
```

Method 2:

```
element.event = function () {
};

window.onload = function () {
};

nameField.onfocus = function () {
};

nameField.onblur = function () {
};

myElement.onclick = function() {
};
```
//we put semicolon since the whole thing is a statement)

Method 3:

```
document.addEventListener(event, function,
useCapture);
```

useCapture

Optional. A Boolean value that specifies whether the event should be executed in the capturing or in the bubbling phase.

- true - the event handler is executed in the capturing phase;
- false - Default. The event handler is executed in the bubbling phase.

```
document.addEventListener(‘click’, myFunction,
false);
document.removeEventListener(‘click’,
anotherFunction, false);
```

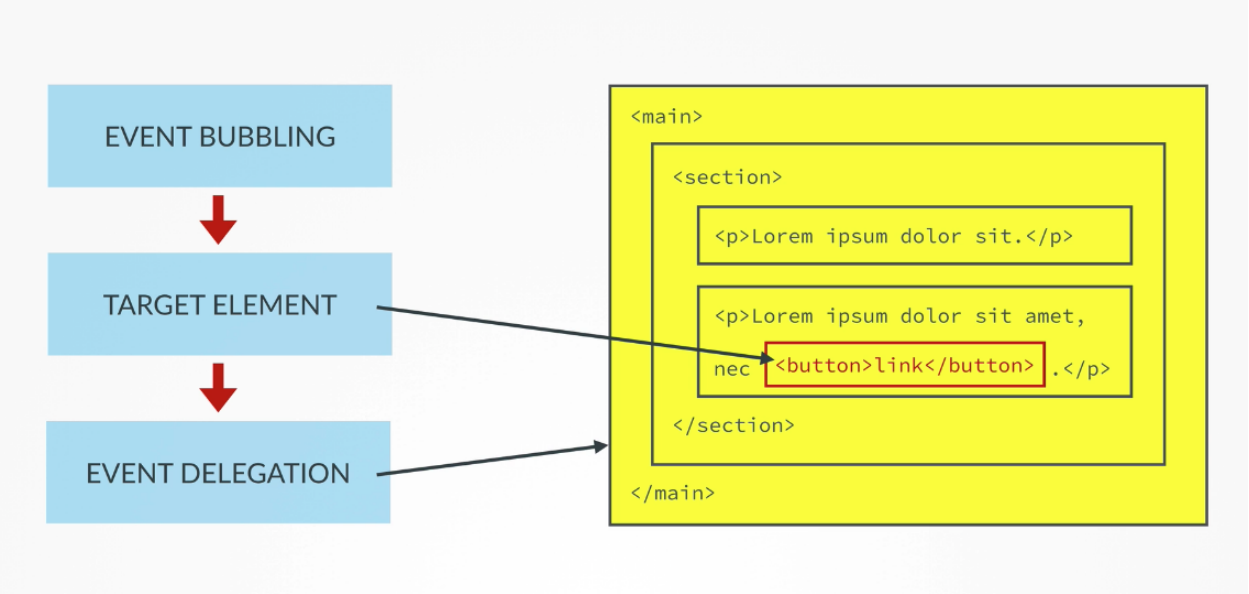
```
window.onload = function() {
      prepareEventHandlers();
};

function prepareEventHandlers() {
      ...
}

setTimeout(myFunction, 5000);
setInterval(myFunction, 5000);

var intervalHandle = setInterval(myFunction, 5000);
myImage.onclick = function() {
      clearInterval(intervalHandle);
};
```

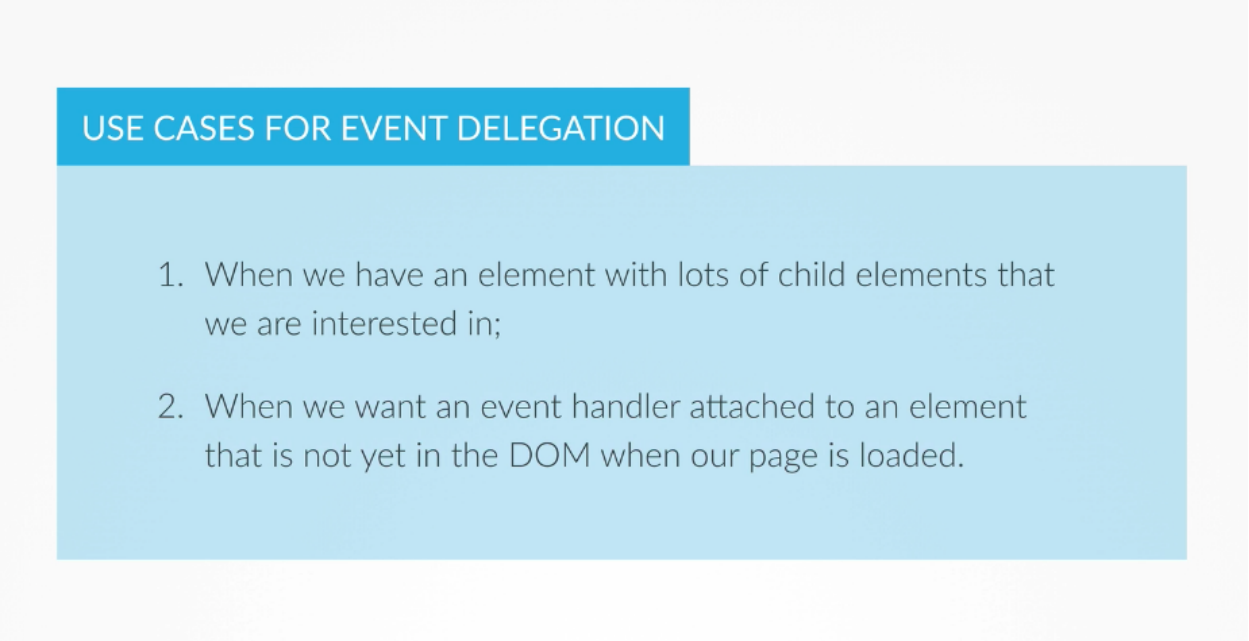
Event Delegation

Event Bubbling

Events bubble up inside the DOM tree.

Event delegation is to not set up the event handler on the original element (target element) but to attach it to a parent element.





Callback Functions

When we pass a callback function as an argument to another function, we are only passing the function definition. We are not executing the function in the parameter. In other words, we aren't passing the function with the trailing pair of executing parenthesis () like we do when we are executing a function.

Callback Functions Are Closures

When we pass a callback function as an argument to another function, the callback is executed at some point inside the containing function's body just as if the callback were defined in the containing function. This means the callback is a closure. So the callback function can access the containing functions' variables, and even the variables from the global scope

Dates

```
let today = new Date();                 // current date and time
let y2k = new Date(2000, 0, 1);         month is zero based
```

Methods

```
today.getFullYear();            YYYY
today.getMonth();               //returns 0-11
today.getDate();                // 1-31 day of month
today.getDay();                 // 0-6 of week. 0 = sunday
today.getHours();               // 0-23
today.getTime();                        // milliseconds since
1/1/1970
```

```
today.toLocaleDateString("en-US", options);
// where const options = { weekday: 'long', year:
'numeric', month: 'long', day: 'numeric' };
```

Comparison

```
let date1 = new Date(2000, 0, 1);
let date2 = new Date(2000, 0, 1);

date1 == date2;
// returns false because they different objects

date1.getTime() == date2.getTime();
// returns true
```

Prototypes & Inheritance

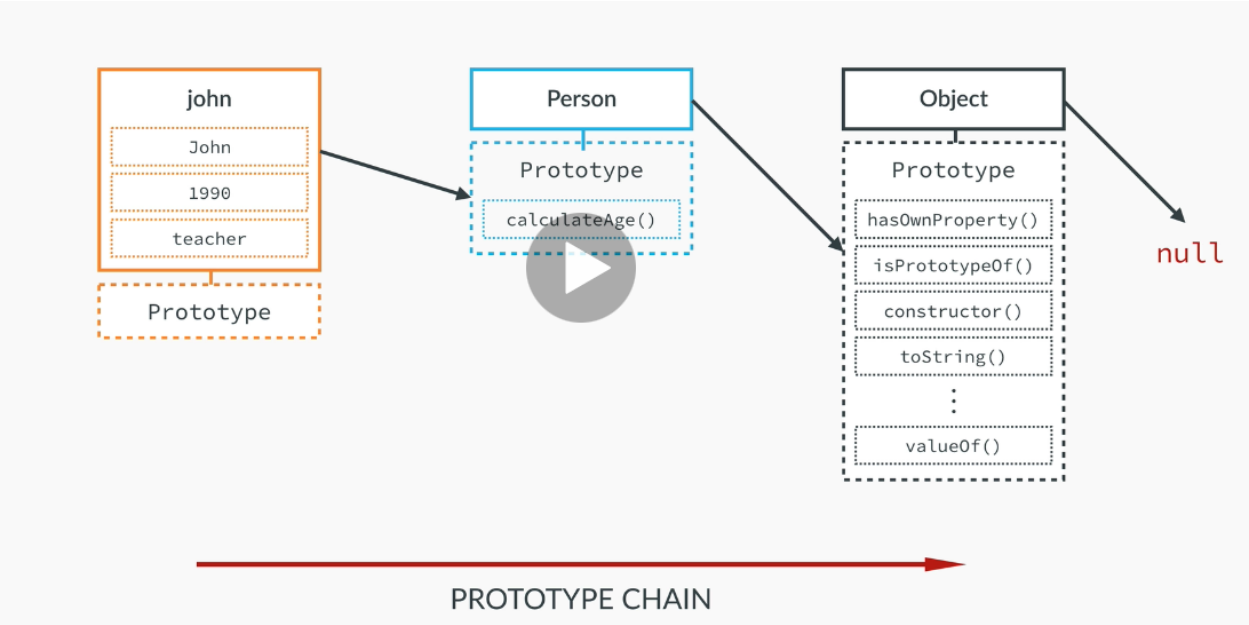
JavaScript is a prototype-based language. Inheritance made possible through prototype property that every object has. To provide inheritance, objects can have a **prototype property object**, which acts as a template object that it inherits methods and properties from.

The prototype property of an object is where we put methods and properties that we want other objects to inherit.

When a certain method or property is called, the search starts in the object itself, and if it cannot be found, the search moves on to the object's prototype. This continues until the method is found. We want to reiterate that the methods and properties are **not** copied from one object to another in the prototype chain. They are *accessed* by *walking up the chain* as described above.

Note that there is a distinction between **an object's prototype** (available via Object.getPrototypeOf(obj), or via the deprecated __proto__ property) and **the prototype property on the constructor function**. The former is the property on each instance, and the latter is the property on the constructor.

Note that Array.prototype is the actual object while __proto__ is a reference to the property object.



Function Constructor Pattern

Step 1: Formalizing object with a function constructor

```
function Player(fname, lname, age) {
      this.firstName = fname;
      this.lastName = lname;
      this.age = age;
}
```

Step 2: Adding methods

```
Player.prototype.name = function() {
      return `${this.fisrtName} ${this.lastName}`;
};
```

Or

```
let personPrototype = {
      calculateAge: function () {

      }
}
```

```
let john = Object.create(personPrototype);
john.name = 'John';
john.age = 23;

let jane = Object.create(personPrototype,
{
      name: { value: 'Jane'}
});
```

Creating an object by calling its constructor

```
var fred = new Player("Fred", "Johnston", 32);
fred.hasOwnProperty('firstName');

fred instanceof Player
```

The **new** keyword does the following things:

1. Creates a blank, plain JavaScript object;
2. Links (sets the constructor of) the newly created object to another object by setting the other object as its parent prototype;
3. Passes the newly created object from *Step 1* as the this context;
4. Returns this if the function doesn't return an object.

Class

```
class Person {
      constructor(name) {
            this._name = name;
      }

      get name() {
            return this._name.toUpperCase();
      }

      set name(newName) {
            this._name = newName;
      }
```

```
        calculateAge() {
            return new Date().getFullYear() - this.age;
        }
    }
```

Note that class definitions are not hoisted. Moreover, we cannot add properties to classes only methods.

Subclass

```
    class Athlete extends Person {
        constructor(name, medals) {
            super(name);
            this.medals = medals;
        }

        wonAMedal() {
            this.medals++;
        }
    }
```

Or in classical:

```
    var Athlete = function(name, job, medals) {
        Person.call(this, name);
        this.medals = medals;
    }

    Athlete.prototype = Object.create(Person.prototype);
    Athlete.prototype.constructor = Athlete;
```

Primitives vs Objects

```
let age = 27;
let obj = {
    name: 'Jane',
    city: 'Vancouver'
}

function change(a,b) {
```

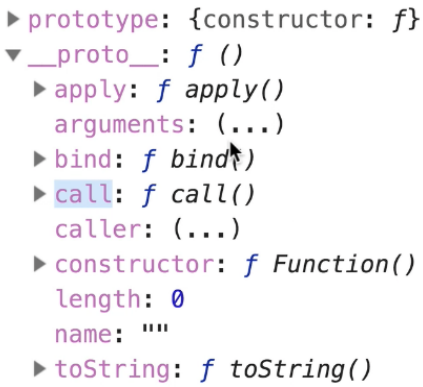
```
        a = 30;
        b.city = 'Seattle'
}

change(age, obj);
```
// age remains unchanged while obj.city has been changed to 'Seattle'

Functions

A function is an instance of the Function type.

call, apply, and bind

These methods are methods of `Function` prototype:



call

`call` method allows us to call a function while binding its `this` keyword to our object of choice:

```
function add(c, d) {
      return this.a + this.b + c + d;
}

add.call({a: 1, b: 3}, 5, 7);
```

apply

`apply` method works exactly like `call` expect we can pass an array as its second parameter whose members are used as the arguments in the function call:

```
add.apply({a: 1, b: 3}, [10, 20]);
```

bind

`bind` works like the `call` method with the difference that the function is not going to be executed, but creates a copy of the original function with the same body and scope permanently bound to the first argument of bind, regardless of how the function is being used. We can also pass the original function parameters as second parameters to bind functions (carrying technique to pre-set some arguments).

```
function f() {
  return this.a;
}

var g = f.bind({a: 'azerty'});
console.log(g()); // azerty

var h = g.bind({a: 'yoo'}); // bind only works ONCE!
console.log(h()); // azerty

var o = {a: 37, f: f, g: g, h: h};
console.log(o.f(), o.g(), o.h()); // 37, azerty, azerty
```

Pass by value vs pass by reference

pass by value

If the variable that holds value of a primitive value type is passed to a function as an argument its value cannot be changed inside the function.

```
function myFunction(parameterA){
```

```
        parameterA = 20;
        return parameterA;
}

const a = 10;
console.log(myFunction(a));
console.log(a);
```

pass by reference

If the variable that holds value of a primitive value type is passed to a function as an argument its value cannot be changed inside the function.

```
function myFunction(parameterArray){
        parameterArray.push(20);
        console.log(parameterArray);
        return parameterArray;
}

const myArr = [10, 5];
console.log(myArr);
console.log(myFunction(myArr));
```

IIFE (Immediately Invoked Function Expression)

By putting a snippet inside of (), we tell the JavaScript parser that it should be treated as an expression and not a statement. We can use this to invoke a function as follows:

```
(function() {
        console.log("Hello World"!);
})();
```

We can use IIFE to make sure that the global object is not polluted.

Architecting a solution

1. Decide on objects (look for nouns);

2. Decide on methods to be added to objects' prototypes

Module Patterns

To encapsulate the x variable and add function:

```
const budgetController = (function() {
      let x = 34;

      let add = function (a) {
            return x + a;
      }
      return {
            publicMethod: function(b) {
                  console.log(add(b));
            }
      };
})();
```

Debugger Statement

Debugger Steps

Add a 'debugger' statement in your function
Call the function manually
At the terminal, run 'node inspect index.js'
To continue execution of the file, press 'c' then 'enter'
To launch a 'repl' session, type 'repl' then 'enter'
To exit the 'repl', press Control + C

ES6

Destructuring

Arrays

```
const [name, age] = ['John', 29];

const [age, retirement] = calcAgeRetirement(1990);
// where calcAgeRetirement returns an array
```

Objects

```
const {firstName, lastName} = obj;
// firstName and lastName should match the actual property name of the object

const {firstName: a, lastName: b} = obj;


const {firstName = "John", lastName = "Smith"} = obj;

const {firstName = "John", lastName = "Smith",
children: { first, second }} = obj;
```

Arrays

```
const boxesArray = Array.from(boxes);
// to convert boxes to an array
```

Remember that we cannot use continue or break with forEach and map loops. However in ES6 there is a new loop:

```
for (const cur of boxesArray) {

}

ages.find(cur => cur >= 18);
ages.findIndex(cur => cur >= 18);
```

The spread operator

1. Flat out an array

```
function add(a, b, c, d) {
     return a + b + c + d;
}

ages = [12, 33, 89, 4];
const sum = function add(...ages);

const newArray = [];
newArray.push(1);
newArray.push(...ages);
```

2. Join two arrays or objects

```
[...a,...b]
// this creates a brand new array

let obj1 = {
     firstName: "John",
     lastName: "Smith"
}


let obj2 = {
     ...obj1,
     age: 32
}
```

Rest parameters

```
function isFullAge(...years) {
     years.forEach(e => (2016 - e) >= 18);
}

isFullAge(1990, 1980, 2001);
// To convert parameters to an array
```

Default parameters

```
function Person(firstName, nationality = 'Canadian') {
```

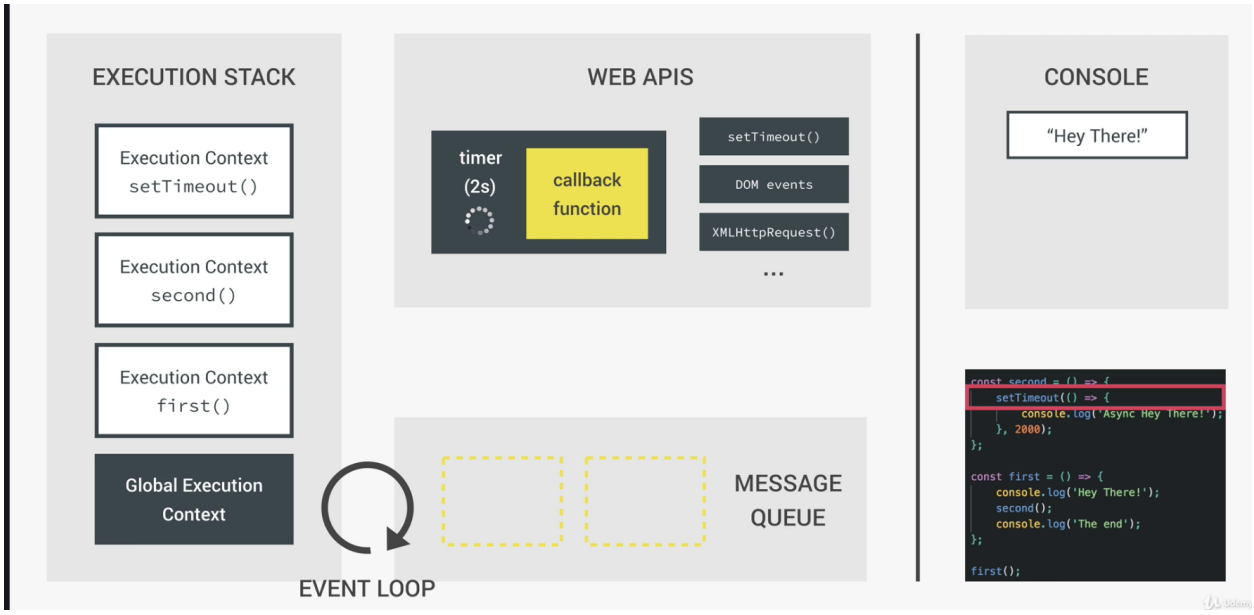
```
}
```

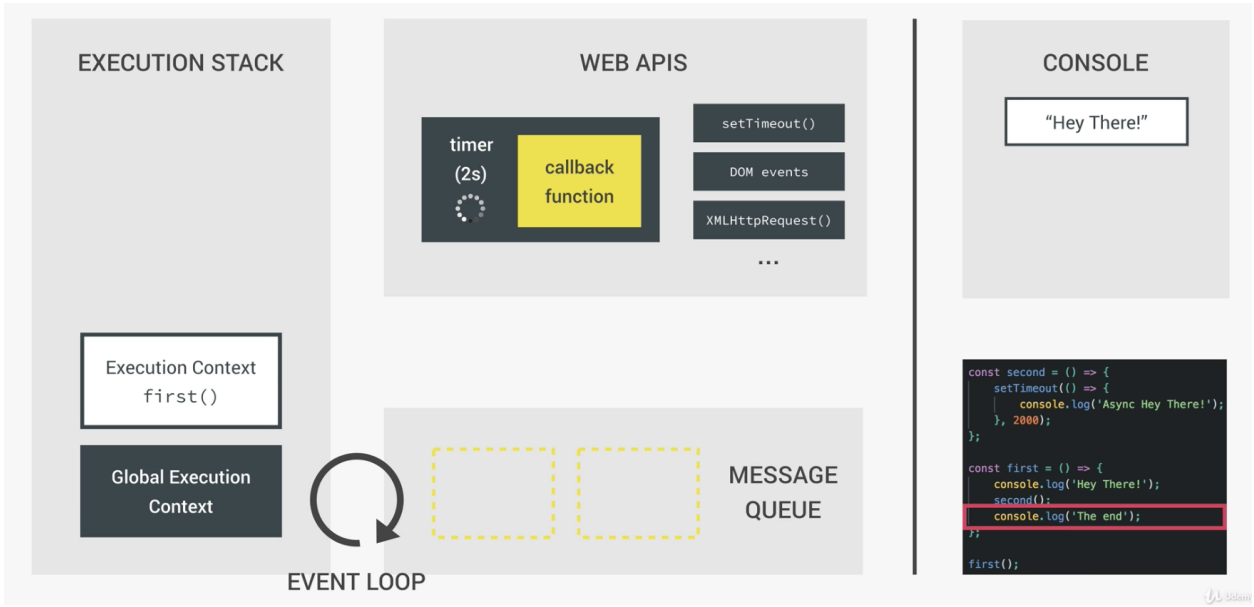
Map

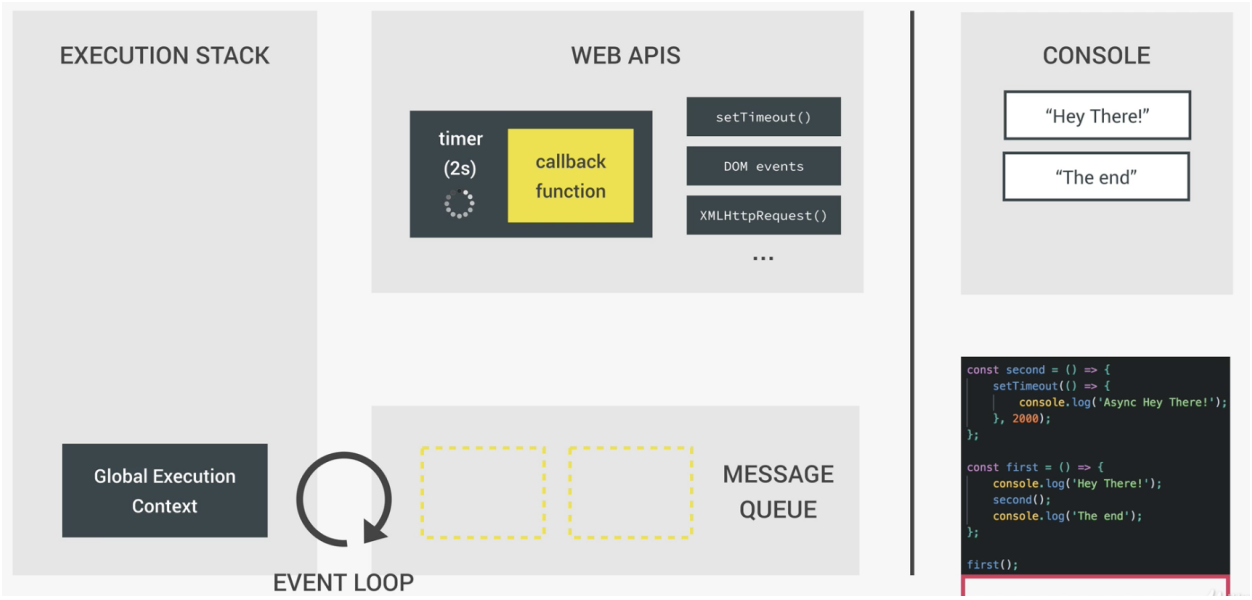
```
const question = new Map();
question.set(<key>,<value>);
// where key can be string, number, or boolean

question.get(<key>);
question.size

if (question.has(<key>)) {
     question.delete(<key>);
}

question.clear();

question.forEach((value, key) => {


});

for (let [key, value] of question.entries()) {

}
```

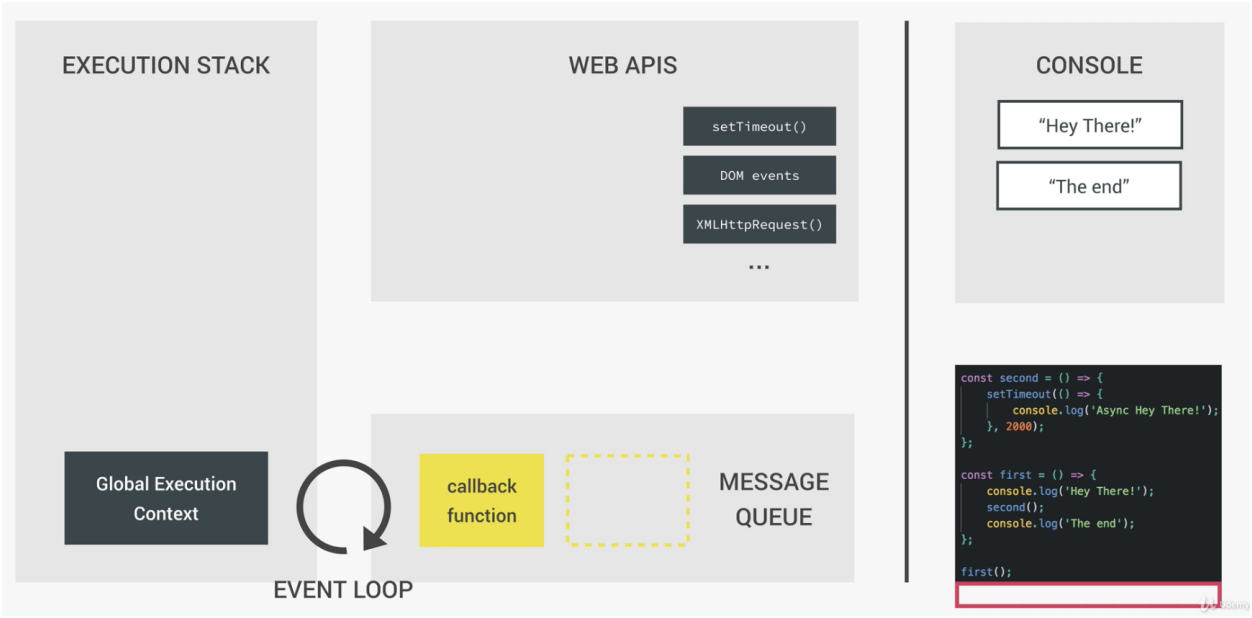
Asynchronous JavaScript

Some functions like `setTimeout()` don't belong to JavaScript runtime, they belong to WEB API. They are just made available in JavaScript runtime. Whenever they are being called a timer gets created with the callback function and gets pushed into the WEB API:

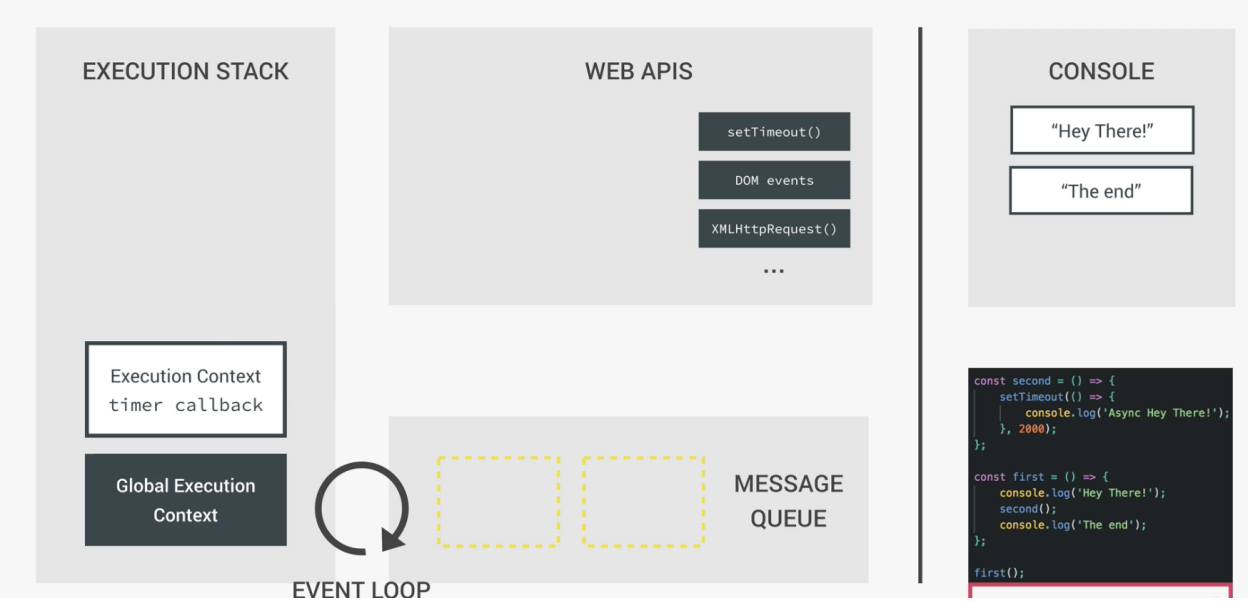

EXECUTION STACK
Execution Context
setTimeout()
Execution Context
second()
Execution Context
first()
Global Execution Context
WEB APIS
timer
(2s)
callback function
setTimeout()
DOM events
XMLHttpRequest()
...
CONSOLE
"Hey There!"
MESSAGE QUEUE
EVENT LOOP
const second = () => {
setTimeout(() => {
console.log('Async Hey There!');
}, 2000);
};
const first = () => {
console.log('Hey There!');
second();
console.log('The end');
};
first();


EXECUTION STACK
Execution Context
first()
Global Execution Context
WEB APIS
timer
(2s)
callback function
setTimeout()
DOM events
XMLHttpRequest()
...
CONSOLE
"Hey There!"
MESSAGE QUEUE
EVENT LOOP
const second = () => {
setTimeout(() => {
console.log('Async Hey There!');
}, 2000);
};
const first = () => {
console.log('Hey There!');
second();
console.log('The end');
};
first();



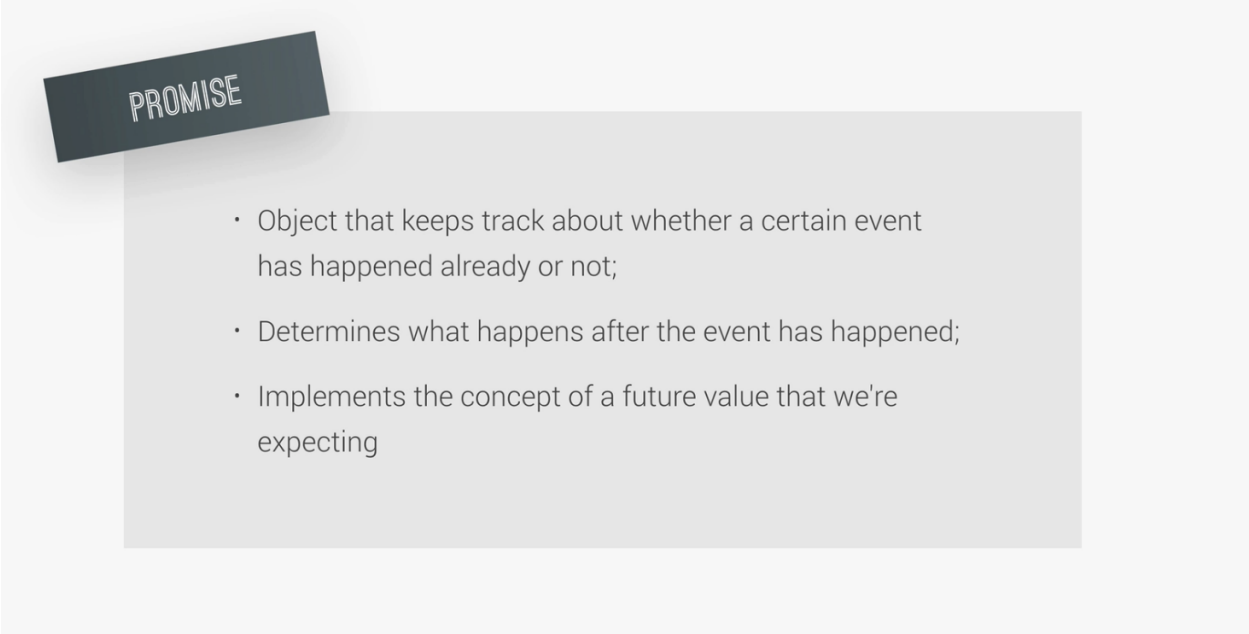


The job of event loop is to constantly monitor message queue and Execution Stack and push the first callback function to the Execution Stack as soon as the Execution Context is empty.



Promise

A promise is an object representing the eventual completion or failure of an asynchronous operation. It will eventually get resolved or rejected with a returned data or an error.



Creation

```
let pr = new Promise((resolve, reject) => {
   setTimeout(() => {
       resolve([423, 564, 213]);
   }, 5000);
});
```

Consuming

```
pr.then((value) => {
   console.log(value);
}).catch(() => {

});
```

How to chain

```
pr1.then(() => {
      return pr2;
}).then(() => {})
```

Implicit return:

```
pr1.then(() => pr2)
```

Async/Await

As soon as we annotate a function with an `async` keyword that function is going to return a promise. If the function returns a value, the promise will be resolved with that value. If the function throws an exception, the promise will be rejected. Moreover, it can be consumed like a standard promise using `then` keyword.

Now let's assume that `getRecipe` is a function that returns a promise. However, we can only use `await` keyword inside a function that has been flagged with `async`. So, what we do is creating a new function `getRecipeAsyncAwait` and flag it with `async`:

```
async function getRecipeAsyncAwait() {
      const ids = await getRecipe();
}
```

Or

```
const getRecipeAsyncAwait = async () => {
      const ids = await getRecipe();
}
```

Now we can call `getRecipeAsyncAwait` like a non async function

```
getRecipeAsyncAwait();
```

Memoization

Memoization is a specific form of caching that involves caching the return value of a function based on its parameter.

Currying

Currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluation of sequence of functions each with a single argument. It is useful to create utility functions.

```
const multiply = (a,b) => a * b;
const multiplyBy = (a) => (b) => a * b;
```