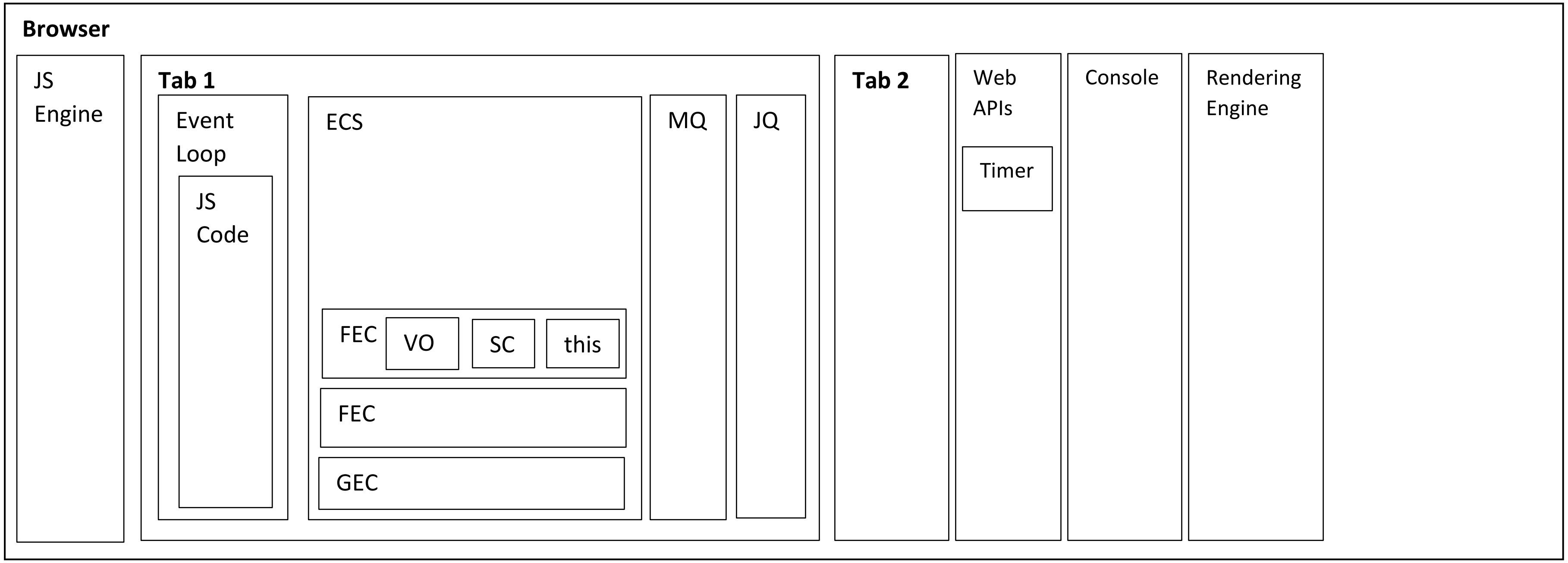
JAVASCRIPT FUNDAMENTAL CONCEPT SUMMARY

By Tam Nguyen (https://www.linkedin.com/in/tam-nguyen-a0792930)



1. **JS Engine**
    - Parser and Compiler scans through the code and executes it in each event loop
    - JavaScript is single threaded, synchronous, and non-blocking (almost IO Primitives are non-blocking)
    - Asynchronous execution occurs outside JS Engine

2. **Event Loop**
    - Single threaded, synchronous execution
    - Every Browser tab has an event loop
    - When JavaScript code file first loads in the browser, it runs inside the event loop
    - Event loop periodically checks ***Execution Context Stack (ECS,*** *aka call stack*) to push and pop ***Execution Context*** (***EC***) to/from the call stack
    - Event loop checks the ***ECS*** first. When the ***ECS*** is ***empty***, it looks into **Job Queue** (***JQ***) and then ***Message Queue*** (***MQ****)*
    - When ***JS Engine*** handles an event in ***MQ***, it creates an ***EC*** for the event handler, which has been registered to the event, and pushes it to the ***ECS*** (***ECS*** has been empty before this so this ***EC*** is the only one in ***ECS*** while the event is handled)
3. **Execution Context Stack (ECS)**
    - A LIFO stack memory where ***Global Execution Context*** *(****GEC****)* and ***Global Execution Contexts*** *(****FECs****)* are pushed into while code is executed

4. **Job Queue (JQ)**
    - FIFO memory used by Promises, mechanisms to run async methods, thus also by async/await (built on top of Promises)
    - From ES6
    - Promises which resolve before current function ends will be executed right after the current function

5. **Message Queue (MQ, aka Event Queue)**
    - FIFO
    - Callback functions, callback function as the first parameter of setTimeout(), user-initialized events (click/keyboard events), fetch(http), and DOM events are queued in MQ (Event table first then pushed to MQ)

6. **IO Primitives**
    - Almost all IO primitives (WebAPIs - network requests, filesystem operations, etc.) are non-blocking

7. **Variable Object (VO, aka Activation Object)**
    - A special object in JavaScript which contains all the variables inside the function, function arguments and inner functions declarations information

8. **Lexical Environment**
    - Where something sits physically in the code of the JavaScript file

9. **Outer Environment**
    - Sits inside ***FEC***
    - Points to the parent ***EC lexically*** (physically)

10. **Scope Chain (SC):**
    - A list of all the ***VOs*** of the functions inside which the current function exists, including the ***VOs*** of ***GEC***. Scope chain also contains the current function ***VO***
    - Use ***Outer Environment Reference*** to find variables in the chain
    - Variable is searched in ***SC*** using ***Outer Environment References*** (lexically)

11. **Global Execution Context (GEC)**
    - When JavaScript code file first loads in the browser and is executed, ***JS Engine*** creates one ***GEC*** and pushes the ***GEC*** to ***ECS*** **(**there is only ONE ***GEC***)
    - While ***JS Engine*** executes the global code, every time it encounters a ***function call***, it creates a ***FEC*** for that function and pushes that ***FEC*** on top of the ***ECS***. Global code execution is paused and ***JS Engine*** executes the function whose ***FEC*** is at the top of the ***ECS***
    - ***JS Engine*** creates a special ***Global Object*** inside ***GEC***. For browser environment, this ***Global Object*** is ***Window*** object
    - All the function and variable declarations in the global scope of JavaScript file will be inside the ***Global Object*** and are accessed via dot notation (i.e. *Window.myVar*)
    - ***JS Engine*** creates a global ***this*** variable inside ***GEC*** which points to the ***Global Object***

12. **Function Execution Context (FEC)**
    - While ***JS Engine*** executes the global code, each time a function is invoked, it creates a ***FEC*** for that function and pushes that ***FEC*** on top of the ***ECS***. Global code execution is paused and ***JS Engine*** executes the function whose ***FEC*** is at the top of the ***ECS***
    - ***JS Engine*** creates an ***Outer Environment*** inside the ***FEC***
    - After the function was executed, ***JS Engine*** pops the ***FEC*** for that function out from the ***ECS*** and starts the same process for the function below this function if any or resumes the global code execution

13. **Execution Context (EC) Creation**
    - ***ECs*** (***GEC*** or ***FECs***) are created by 2 phases:

- **Creation Phase**: ***JS Engine*** scans the global code in browser (in creating ***GEC***) or the function code (in creating ***FEC***) to compile the code (This is ***JS Engine Compilation Phase*** in which ***JS Engine*** will handle only the declarations, and does not bother about the values). ***JS Engine*** creates a new ***EC object*** and performs the following tasks to update the ***EC object***
    - Creates VO
    - Creates SC
    - Initialize the value of variable ***this***
- **Execution Phase**:
    - ***JS Engine*** executes the global code (in creating ***GEC***) or the function code (in creating ***FEC***) line by line and updates the VO with the values of the variables inside the global code (in creating ***GEC***) or inside the function (in creating ***FEC***)

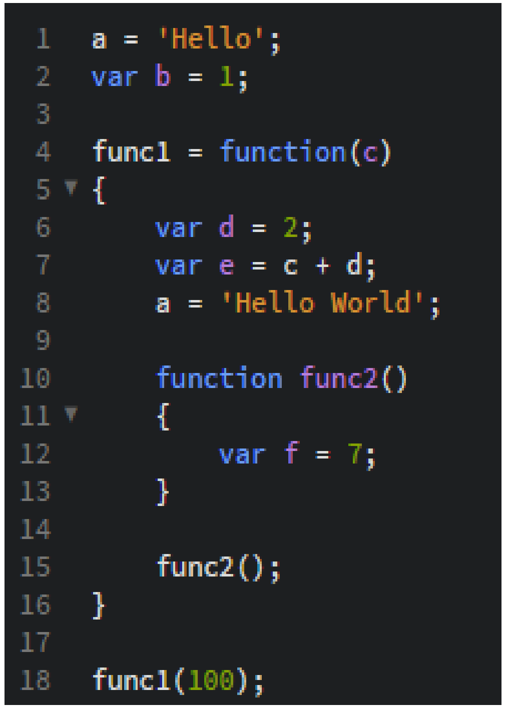
14. JS Engine Scanning Code

- When ***JS Engine*** encounters a function definition while scanning code, it will create a new **property** by the name of the function.
- This property points to the function definition in the HEAP memory (Function definitions are stored in HEAP memory, not in the ***ECS***)

15. Defer A Function Execution Until ECS Is Empty

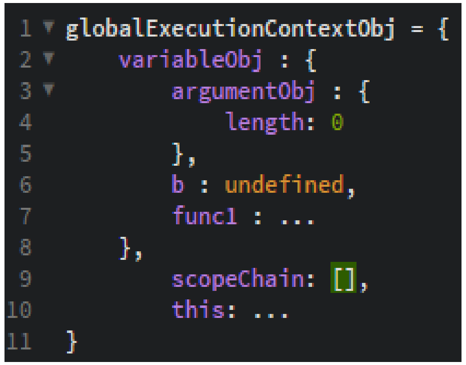
- Use JavaScript built-in method ***setTimeout***(()=>{}, 1000)
- When ***setTimeout()*** is invoked, ***JS Engine*** starts a timer with timeout value set in the second parameter of the ***setTimeout()***
- When the timeout is reached, ***JE Engine*** puts the call back function (second parameter of the ***setTimeout()***) at the back of ***MQ***
- When ***ECS*** is empty, ***Event Loop*** will look at the ***MQ*** and pushes the callback execution context into the ***ECS***

16. Example on EC Creation



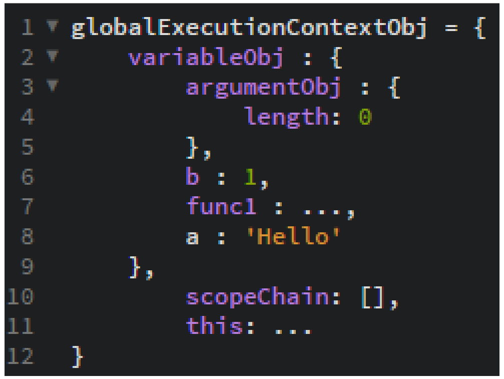
```
1   a = 'Hello';
2   var b = 1;
3
4   func1 = function(c)
5 ▼ {
6       var d = 2;
7       var e = c + d;
8       a = 'Hello World';
9
10      function func2()
11 ▼    {
12          var f = 7;
13      }
14
15      func2();
16  }
17
18  func1(100);
```

- When the code file is loaded in browser, ***JS Engine*** performs **GEC creation phase** with the details as below:
  - ○ creates a new ***GEC*** object ***globalExecutionContextObj***
  - ○ scans through the code line by line and updates ***globalExecutionContextObj*** as below:
    - ▪ Line 1: not variable declaration or function declaration → ***JS Engine*** does nothing
    - ▪ Line 2: variable declaration → ***JS Engine*** creates a property named ***b*** in ***globalExecutionContextObj*** and initializes it with value ***undefined***
    - ▪ Line 4: function declaration → ***JS Engine*** stores the function declaration in HEAP, and creates a property named ***func1*** pointing to the location of the function declaration in HEAP
    - ▪ Line 18: not function declaration → ***JS Engine*** does nothing



```
globalExecutionContextObj = {
    variableObj : {
        argumentObj : {
            length: 0
        },
        b : undefined,
        func1 : ...
    },
        scopeChain: [],
        this: ...
}
```

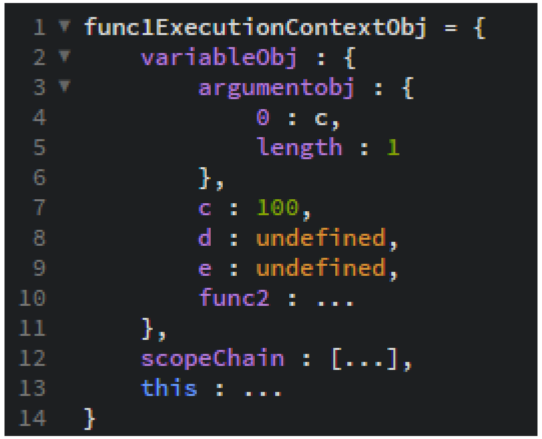
- After the ***GEC creation phase***, ***JS Engine*** performs ***GEC execution phase*** with the details as below:
  - ○ executes the code line by line and updates ***globalExecutionContextObj*** as below:
    - ▪ Line 1: there is no property named ***a*** in ***variableObj*** of ***globalExecutionContextObj*** → **JS Engine** adds property ***a*** to ***variableObj*** and set its value to *'Hello'*
    - ▪ Line 2: **JS Engine** updates the value of property named ***b*** in ***globalExecutionContextObj*** to *1*
    - ▪ Line 4: function declaration → **JS Engine** does not do anything and moves to line 18



```
globalExecutionContextObj = {
    variableObj : {
        argumentObj : {
            length: 0
        },
        b : 1,
        func1 : ...,
        a : 'Hello'
    },
        scopeChain: [],
        this: ...
}
```

- There is a function call at line 18 so ***JS Engine*** performs ***FEC creation phase*** with the details as below:
  - ○ creates a new ***FEC*** object ***func1ExecutionContextObj***
  - ○ scans through the func1 code line by line and updates ***func1ExecutionContextObj*** as below:

- Line 4: there is an argument named ***c*** → ***JS Engine*** adds ***c*** into ***argumentObj*** of ***func1ExecutionContextObj***, and then add a property named ***c*** and initializes it value to 100
- Line 6: variable declaration → ***JS Engine*** creates a property named ***d*** in ***func1ExecutionContextObj*** and initializes it with value ***undefined***
- Line 7: variable declaration → ***JS Engine*** creates a property named ***e*** in ***func1ExecutionContextObj*** and initializes it with value ***undefined***
- Line 8: not variable declaration or function declaration → ***JS Engine*** does nothing
- Line 10: function declaration → ***JS Engine*** stores the function declaration in HEAP, and creates a property named ***func2*** pointing to the location of the function declaration in HEAP
- Line 15: not function declaration → ***JS Engine*** does nothing

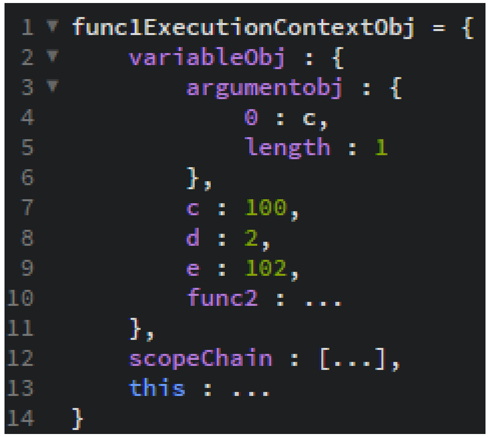


```
func1ExecutionContextObj = {
    variableObj : {
        argumentobj : {
            0 : c,
            length : 1
        },
        c : 100,
        d : undefined,
        e : undefined,
        func2 : ...
    },
    scopeChain : [...],
    this : ...
}
```

- After the ***FEC creation phase*** for func1, ***JS Engine*** performs ***FEC execution phase*** with the details as below:
  - executes the code line by line and updates ***func1ExecutionContextObj*** as below:
    - Line 6: ***JS Engine*** updates value of property named ***d*** in ***func1ExecutionContextObj*** to *2*
    - Line 7: ***JS Engine*** updates value of property named ***e*** in ***func1ExecutionContextObj*** to *102*
    - Line 8: ***a*** is not a property of ***func1ExecutionContextObj*** nor a variable declaration → ***JS Engine*** looks into ***globalExecutionContextObj*** with the help of scope chain (***lexically***) and checks if a property with the name ***a*** exists in it. Because ***a*** already exists in ***globalExecutionContextObj***, its value will be updated to 'Hello World' (In case ***a*** does not exist

in ***globalExecutionContextObj***, ***JE Engine*** will create a property named ***a*** in ***func1ExecutionContextObj*** and will initialize it with value ‘*Hello World’*

        - Line 10: function declaration → ***JS Engine*** does not do anything and moves to line 15

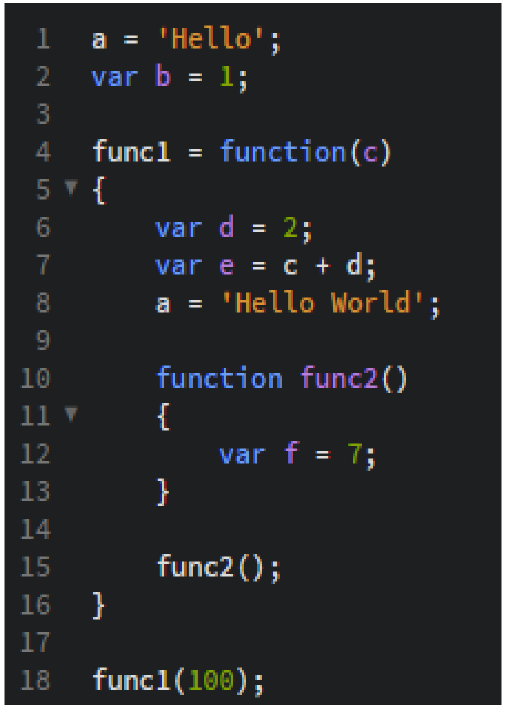


```
func1ExecutionContextObj = {
    variableObj : {
        argumentobj : {
            0 : c,
            length : 1
        },
        c : 100,
        d : 2,
        e : 102,
        func2 : ...
    },
    scopeChain : [...],
    this : ...
}
```

- There is a function call at line 15 so ***JS Engine*** performs ***FEC creation phase*** with the details as below:
    - ○ creates a new ***FEC*** object ***func2ExecutionContextObj***
    - ○ scans through the func2 code line by line and updates ***func2ExecutionContextObj*** as below:
        - Line 10: ***JS Engine*** does nothing
        - Line 12: variable declaration → ***JS Engine*** creates a property named ***f*** in ***func2ExecutionContextObj*** and initializes it with value ***undefined***

- After the ***FEC creation phase*** for func2, ***JS Engine*** performs ***FEC execution phase*** with the details as below:
    - ○ executes the code line by line and updates ***func2ExecutionContextObj*** as below:
        - Line 10: ***JS Engine*** updates the value of property named ***f*** in ***func2ExecutionContextObj*** to *7*

17. Scopes



```
a = 'Hello';
var b = 1;

func1 = function(c)
{
    var d = 2;
    var e = c + d;
    a = 'Hello World';

    function func2()
    {
        var f = 7;
    }

    func2();
}

func1(100);
```

- ***func2ExecutionContextObj*** has access to all the variables and functions defined in ***func1ExecutionContextObj*** and in the ***globalExecutionContextObj*** using the scope chain

- ***func1ExecutionContextObj*** has access to all the variables and functions in ***globalExecutionContextObj***. However, it does not have access to those of ***func2ExecutionContextObj***

- ***globalExecutionContextObj*** does not have access to variables or functions of ***func1ExecutionContextObj*** and ***func2ExecutionContextObj***

18. Hoisting

- The 2 phases of ***Execution Context*** creation (creation phase and execution phase) explains how hoisting is working well (see 14.)
- The ***EC*** creation phase setups memory space for variables and functions → Hoisting

19. Let vs Var

- Var → variable can be used after ***EC creation phase***
- Let → variable can not be used until when code is executed in ***EC execution phase***
- Let → **block scope**. If defined inside a block ({}), even for loop, scope is inside the block. For a loop, a new variable is created for each time the loop is running

20. Closure

- A function still has references to variables to which the current ***FEC*** is supposed to have access and which are referenced to by the ***Scope Chain*** from the current ***EC*** even when the ***FECs*** where those variables are defined are no longer in the ***ECS*** (see 9 and 10)
- ***JS Engine*** creates the closures

21. Object

- Name value pair collection
- Contains ***Primitive*** properties, ***Object*** properties, and ***methods*** (functions)
- Has references to properties and functions
- Has a special/hidden property named__ ***proto__***, used for prototypal inheritance
- Everything in JavaScript (i.e. functions, arrays) except for ***Primitives*** are objects
- Functions have three special properties - ***name*** (optional; not used if a function is anonymous), ***code*** (invocable), and ***prototype*** (only used when used with ***new*** operator); and has 3 special methods - ***bind()***, ***call()***, and ***apply()***
- While ***Primitive*** values are referenced **by value** (copy value), ***Objects*** are referenced to **by reference**
- JavaScript base object is named ***Object*** and is at the bottom of the inheritance tree (all other objects inherit ***Object***). Base object has no ***__proto__*** property

22. This Keyword

- When a function is attached to an object (by default, defined inside an object), ***this*** points to the object inside which the function locates
- For ***GEC***, ***this*** points to ***Window*** object
- To attach an object to a function, use bind (create new function by copying), call (i.e. func1(obj1, param1, param2)), or apply (i.e. func1(obj1, [param1, param2])) → change where ***this*** variable points to

23. Object Inheritance

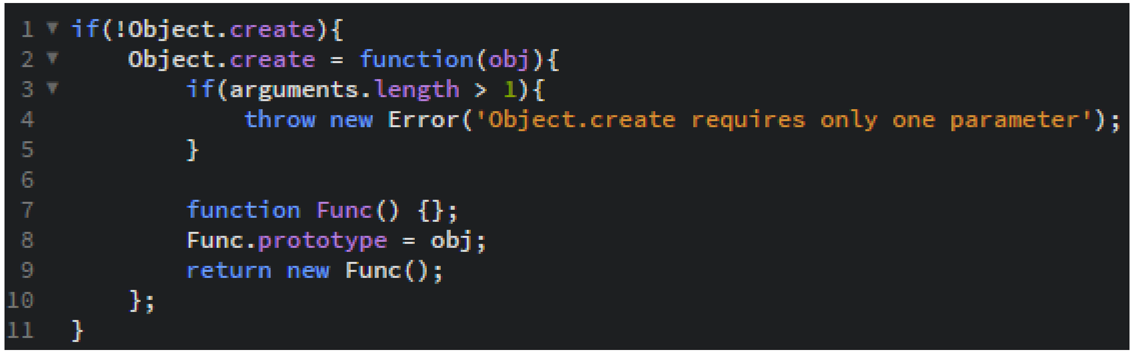
- Prototypal inheritance
- Property ***__proto__*** of ***Object*** (at the bottom of inheritance tree) references to another object. This referenced object has its own ***__proto__*** and this ***__proto__*** references to another object and so on → creating a ***Prototype Chain*** for inheritance staring from **Object**
- Property and Method accessibility (inheritance)
- Two or more objects are allowed to have their ***__proto__*** properties pointing to the same object
- Object B inherits object A → ***EC*** creates ***this*** pointing to B

24. Function Constructor

- A normal function used to create a new object
- Naming convention: first character in upper case to differentiate function constructors to normal functions
- Used together with ***new*** keyword (i.e. *var myVar = new Car()* where *Car* is function constructor)
- How ***new*** operator works:
  - Empty object is created
  - Function constructor is invoked → ***EC*** is created with ***this*** pointing to the previous mentioned empty object
  - As long as the function constructor does not return anything, ***JS Engine*** returns this empty object to the assigned variable (*i.e. myVar* in this example)
- Inheritance: prototype property (named ***prototype***) of the function constructor can be used to set the methods and properties to which ***__proto__*** property of the above-mentioned empty object (created by ***new*** operator) references (i.e. *Car.prototype.getColor = function(){}*). This setting can be done anytime, before or after the ***new*** is applied to the function constructor

25. Polyfill Object Creation

- Polyfill: code which adds a feature that ***JS Engine*** does not have
- We can add polyfill for Object.create(obj) if the browser does not have it to have an easy way to create an object from another object with full inheritance



```
if(!Object.create){
    Object.create = function(obj){
        if(arguments.length > 1){
            throw new Error('Object.create requires only one parameter');
        }

        function Func() {};
        Func.prototype = obj;
        return new Func();
    };
}
```

26. Asynchronous Programming

- Via
    - ○ Callbacks (***MQ mechanism)***
    - ○ Promises (producers/consumers) from ES6 (***JQ mechanism***) → consumers are blocking
    - ○ async/await (for promise consumers) from ES7 (***JQ mechanism***) → await is blocking