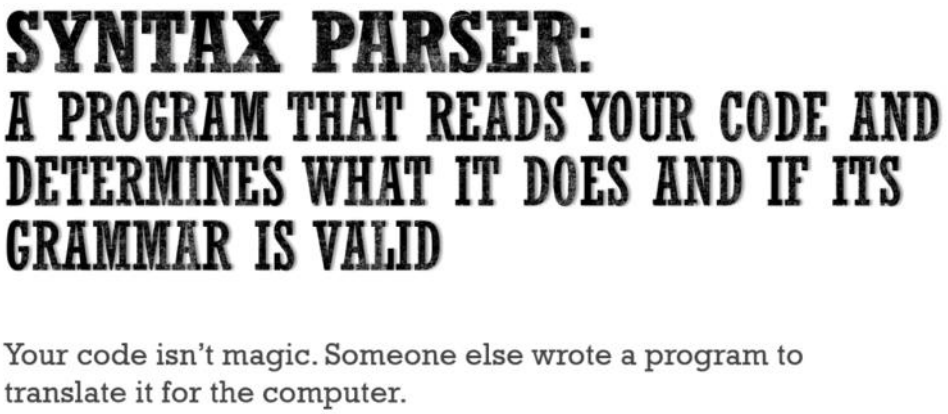
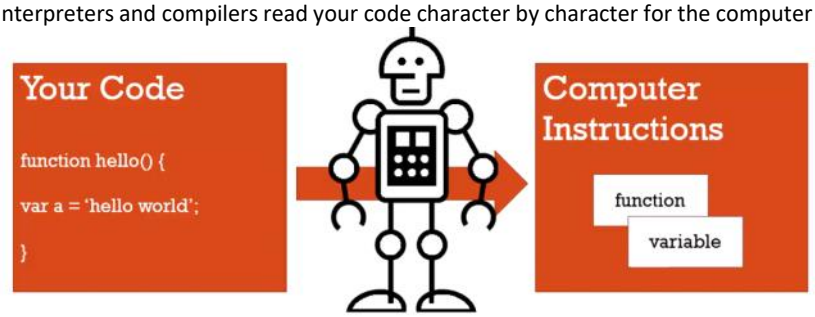
Tuesday, November 6, 2018 9:41 AM

Understanding weird parts of JS

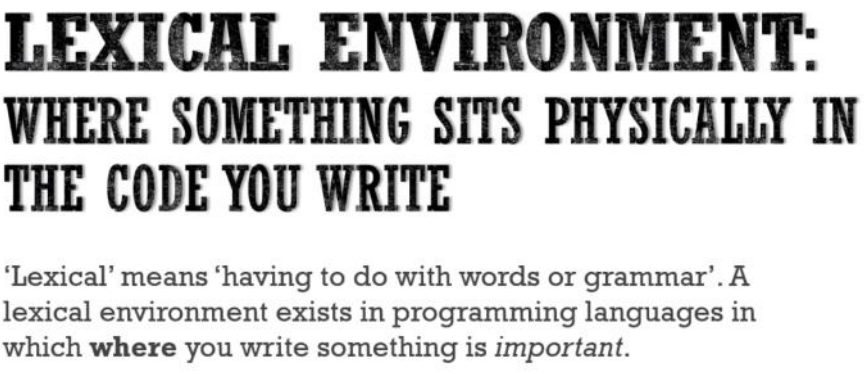
Syntax parsers

- 

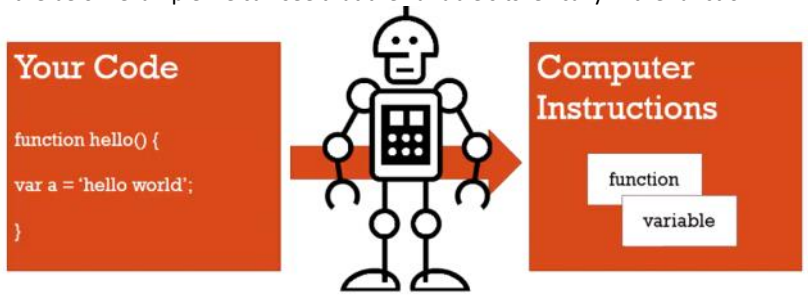
- A program that reads your code and determines what it does if it's grammar is valid
- Interpreters and compilers read your code character by character for the computer to understand.

- 

Lexical environment

- 

- In the below example we can see that the variable sits lexically in the function.

- 

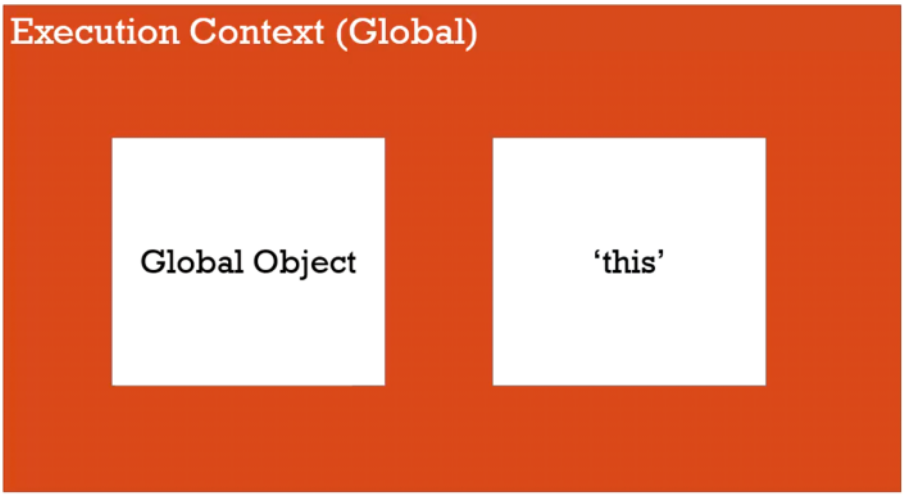
- It literally mean where it is written and what surrounds it.

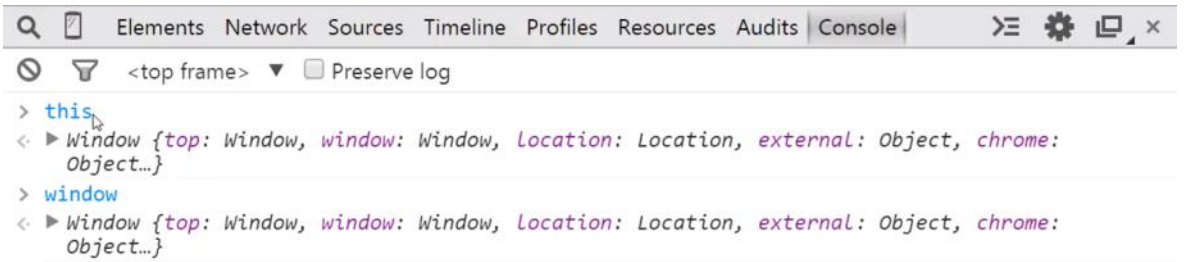
Execution context

- # EXECUTION CONTEXT:
  ## A WRAPPER TO HELP MANAGE THE CODE THAT IS RUNNING

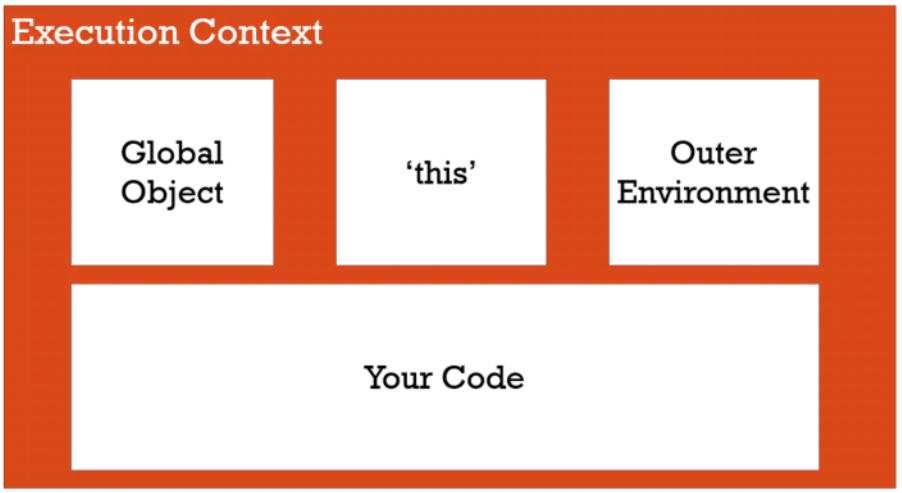
  There are lots of lexical environments. Which one is currently running is managed via execution contexts. It can contain things beyond what you've written in your code.

- The below will be created by the JS engine





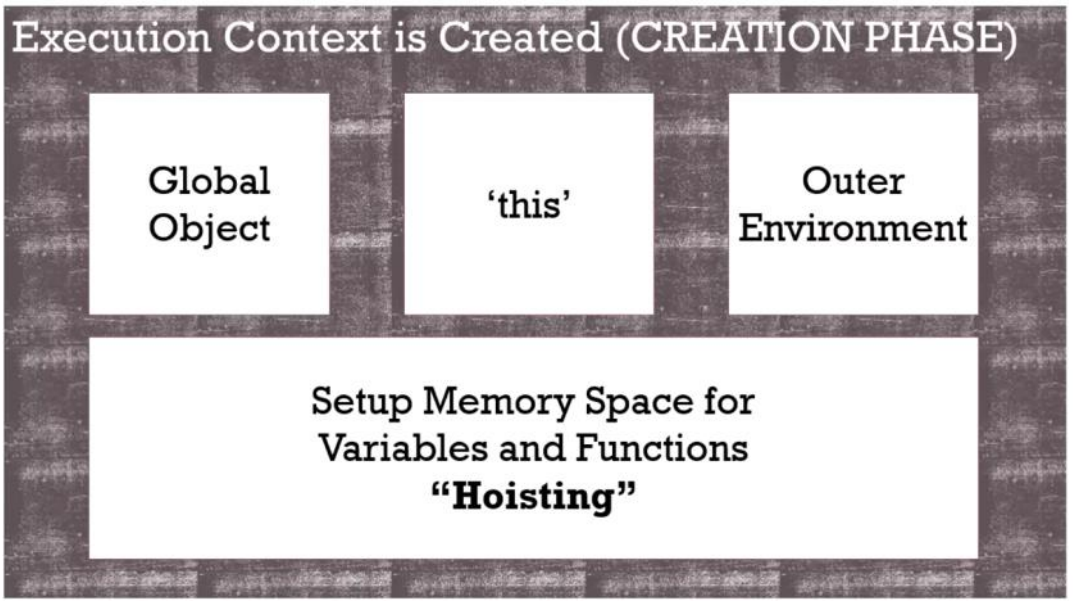
- this refers to window object at browser level.
- Global refers to that it is not inside a function.
- Below is how the js engine looks
  - In the execution context we have 'this' created along with the global object 'window'
  - The code written by us can be directly evaluated or via the global window object

- 

Creation & Hoisting

- Consider the below code
  - ```
    var a = 'Hello World!';

    function b() {
        console.log('Called b!');
    }

    b();
    console.log(a);
    ```
  - It executes and displays "Called b!" and "Hello World"
- Now we change the code a little bit, to call b() first
  - ```
    b();
    console.log(a);

    var a = 'Hello World!';

    function b() {
        console.log('Called b!');
    }
    ```
  - It executes and displays "Called b!" and undefined.
  - This is called **Hoisting**
  - As **functions are hoisted to the top** the output displays "Called b!"
  - It displays **undefined** because the **variable is declared** and a placeholder is set **but the value is not assigned by the time it executed the console.log**
- Now consider the case that line 4 in the above example is removed
  - It displays the following error "a is not defined" as "a" has never been declared
  - ```
    Called b!                                                    app.js:7
    ▶ Uncaught ReferenceError: a is not defined                  app.js:2
    ```

- In JS, the variables and functions are hoisted or moved up to the top by the js engine.
- The code in the below screenshot
  - ```
    b();
    console.log(a);

    var a = 'Hello World!';

    function b() {
        console.log('Called b!');
    }
    ```
  - Will be hoisted by the js engine as below, where we can see that the function and variable are moved to the top

-
```
1  function b() {
2      console.log('Called b!');
3  }
4  var a;
5
6  b();
7  console.log(a);
8
9  a = 'Hello World!';
10
```
    - Actually the code is not moved up but the code is translated by the js engine to setup the memory space for the variable and functions in the execution context of the creation phase
    - 

- Ex**ecution Context(Execution Phase)**
    -
```
1  function b() {
2      console.log('Called b!');
3  }
4
5  b();
6
7  console.log(a);
8
9  var a = 'Hello World!';
10
11 console.log(a);
```
    - O/p would be "Called b!", undefined, Hello World!
    - As variable declaration of a is hoisted first in the creation phase, it logs **undefined** when we console.log on line 7.
    - But in the execution phase the value "Hello World!" will be assigned to a, so it displays "**Hello World!**" when we console.log on line 11.

Single Threaded, Synchronous Execution

- Js in synchronous and single threaded
- 

SYNCHRONOUS:
ONE AT A TIME

- And in order…

Function Invocation and Execution Stack

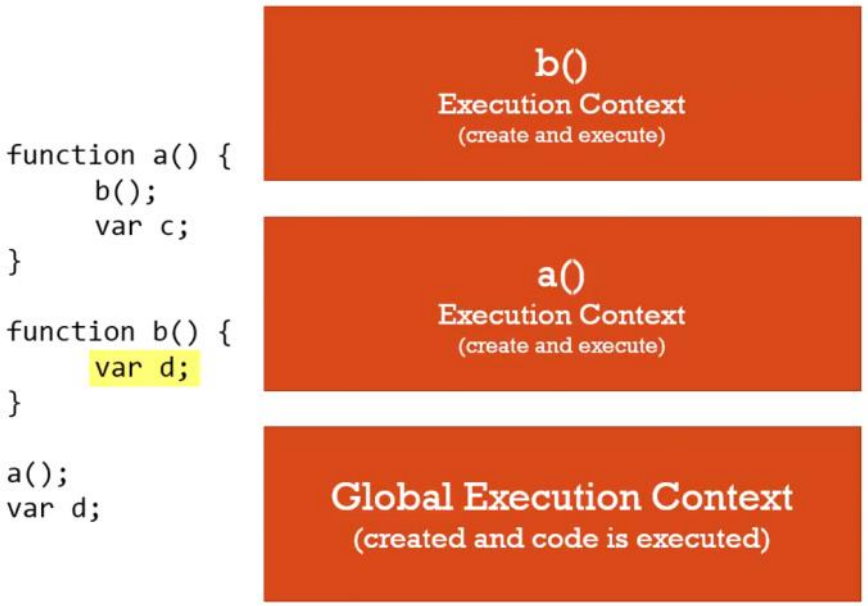
INVOCATION:
RUNNING A FUNCTION

- In Javascript, by using parenthesis ()

- Invocation means running or calling or invoking a function
- Below is how function invocation works

```
function b() {
}

function a() {
      b();
}

a();
```

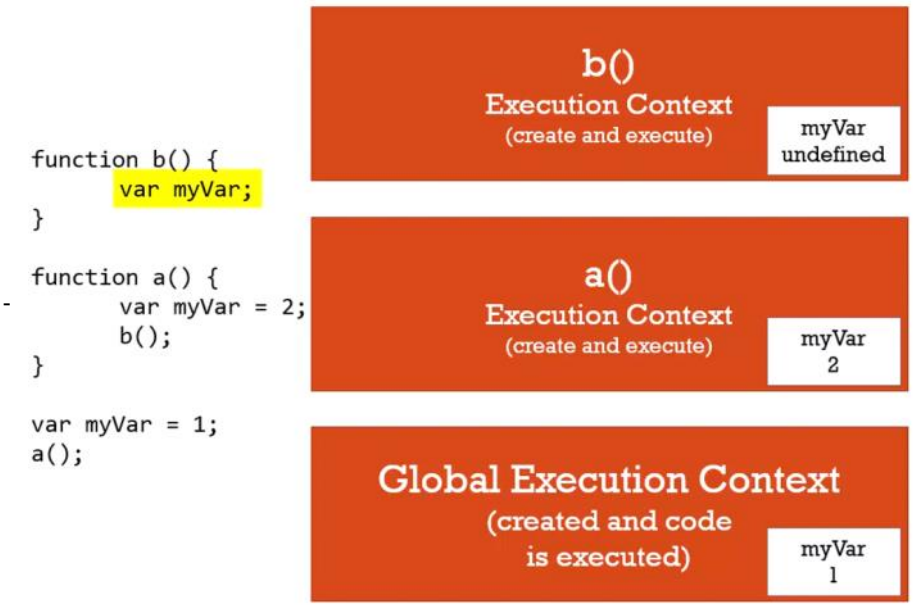
  - First Global execution context is created where the memory space is set up for a and b
  - Next when the function a is invoked, it creates an execution context and adds to the stack to execute the code line by line in function a
  - Next inside function a b() is invoked and another execution context will be created to execute the code line by line inside b()
- In the below example we will see on how variables are executed



- First Global execution context is created where the memory space is set up for a and b
- var d; will not be called yet and js is synchronous, so it creates an execution context and adds to the stack to execute the code line by line in function a and invokes function b
- Next inside function a b() is invoked and another execution context will be created to execute the code line by line inside b() where var d; will be executed
- Now the execution context of b is popped out, then goes back to function a to execute var c; and after that execution context of a is popped out

- Now the var d; line after a() will be executed

Functions, Context and Variable environments

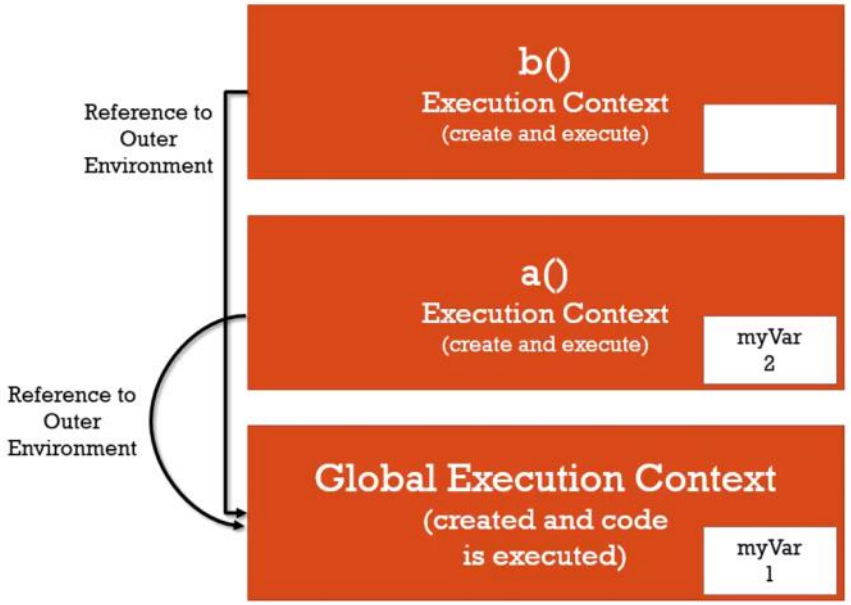


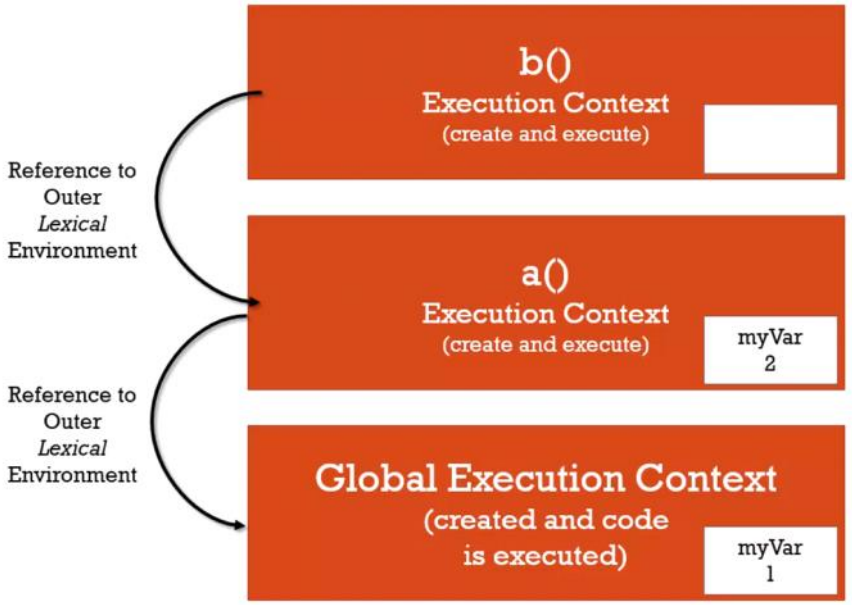
- In the above example myVar has its own value in each execution context.
- Even though they are declared three times they are distinct and unique.

The Scope chain

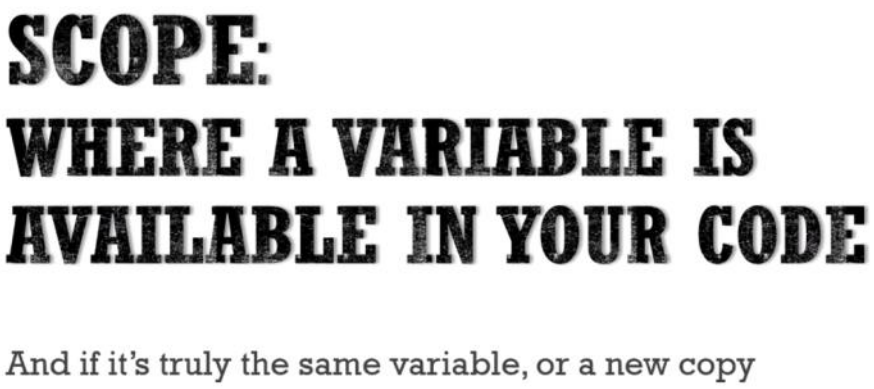
```
function b() {
    console.log(myVar);
}

function a() {
    var myVar = 2;
    b();
}

var myVar = 1;
a();
```

- Output will be "1", as myVar is not defined in the b execution context it picks up the global execution context value.



- This chain of referring to the Global execution context when the variable is not available in the function execution context is called "The Scope Chain".
- In the below example
  -
```
function a() {

    function b() {
       console.log(myVar);
    }

    var myVar = 2;
    b();
}

var myVar = 1;
a();
```
  - The output will log "2"
  - This is because b is defined inside the execution context of function a, and the lexical environment of b is constrained to the execution context of a
  - 

Scope, **ES6** and **let**

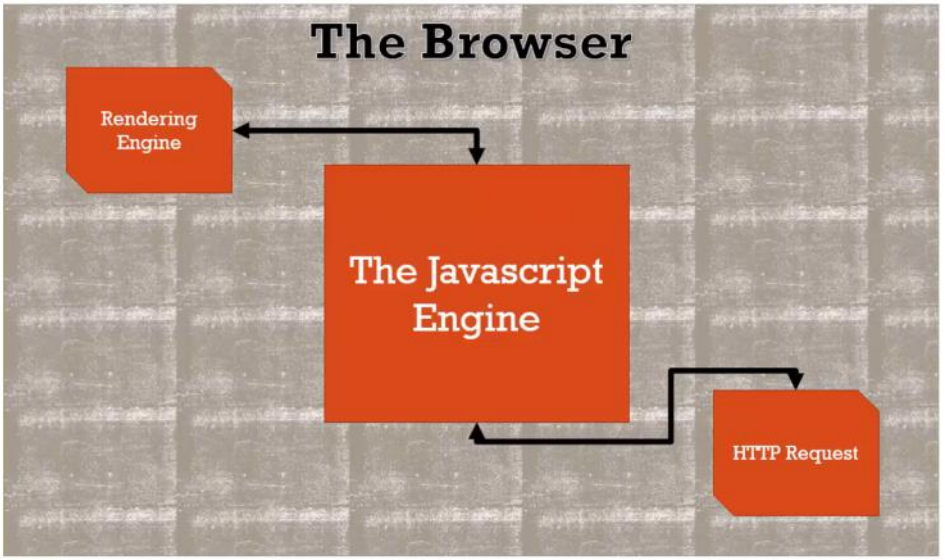
- 

- **let** -> let allows block scoping
- Ex. Below

```
if (a > b) {
    let c = true;
}
```

- Here the c is allocated the memory space, but engine won't allow us to use it before let c= true.

Asynchronous Callbacks

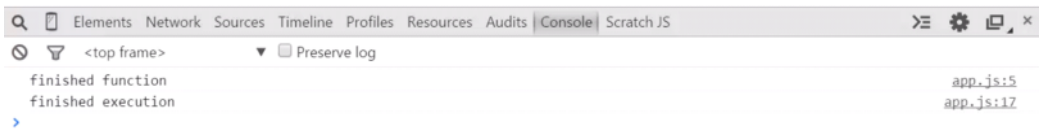


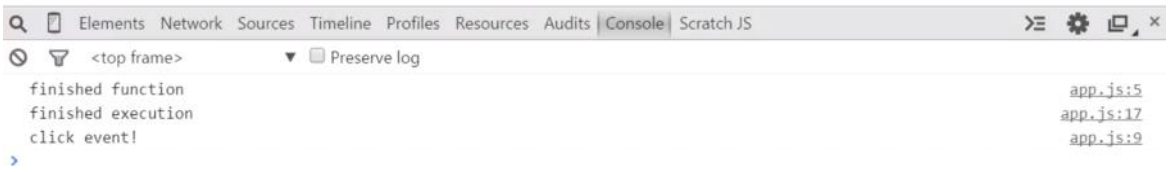
- Browser has various engines apart from js engine
- JS engine runs synchronously (one line at a time)
- But HTTP Request may run asynchronously

- 

Event Queue
- We have an Event Queue which is looked upon once the execution stack is empty
- If we have a function associated with the click event handler the function execution context will be created as below
- 

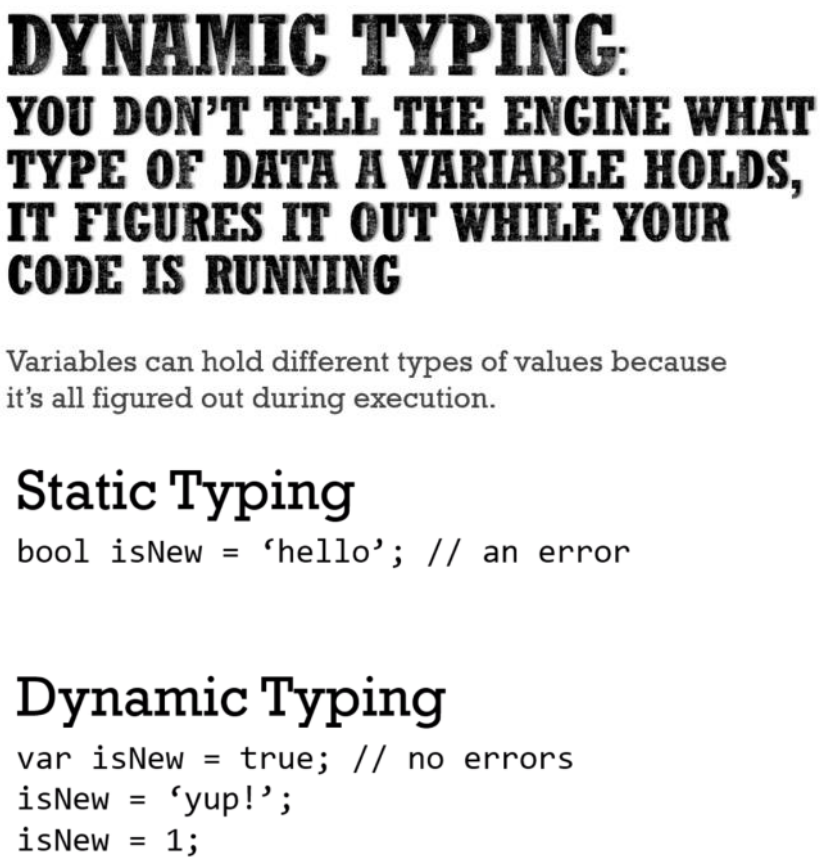
Event Queue
- Once the execution context of the click handler is removed from the stack the click in the event queue is also removed and mo ves onto the next event in the queue.
- 

Event Queue
- So the browser is putting the events in the queue asynchronously but the javascript runs synchronously only.
-
```
// long running function
function waitThreeSeconds() {
    var ms = 3000 + new Date().getTime();
    while (new Date() < ms){}
    console.log('finished function');
}

function clickHandler() {
    console.log('click event!');
}

// listen for the click event
document.addEventListener('click', clickHandler);


waitThreeSeconds();
console.log('finished execution');
```
- Output :
  - When we load the page
  - 

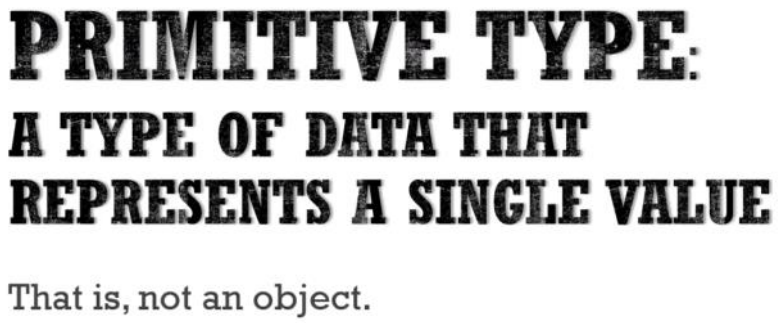
- When we click on the document
- 

- The above is because "Event Queue which is looked upon once the execution stack is empty".

Types and Javascript

- 

Primitive Types

- 

UNDEFINED

- `undefined` represents lack of existence (you shouldn't set a variable to this)

NULL

- `null` represents lack of existence (you can set a variable to this)

BOOLEAN

- `true or false`

NUMBER

- *Floating point* number (there's always some decimals). Unlike other programming languages, there's only one 'number' type…and it can make math weird.

STRING

- **a sequence of characters (both ‘ and “” can be used)**

SYMBOL

- Used in ES6 (the next version of Javascript) We won't talk about this here…

Operators

- **OPERATOR:**
  A SPECIAL FUNCTION THAT IS SYNTACTICALLY (WRITTEN) DIFFERENTLY

  Generally, operators take two parameters and return one result.

- Js translates to a function when it sees an operator as shown in the below screenshot

- ```
  var a = 3 + 4;

  function +(a, b) {
      return // add the two #s
  }
  ```

Operator Precedence and Associativity

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

- **OPERATOR PRECEDENCE:**
  WHICH OPERATOR FUNCTION GETS CALLED FIRST

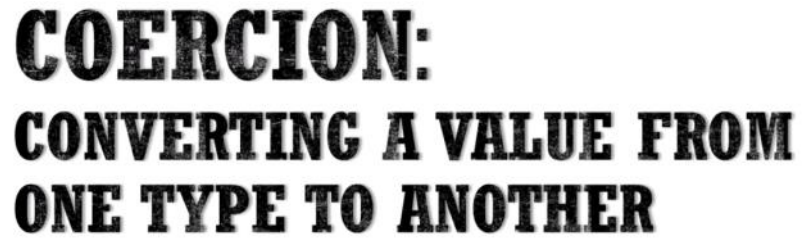
  Functions are called in order of precedence

  (HIGHER precedence wins)

- **ASSOCIATIVITY:**
  WHAT ORDER OPERATOR FUNCTIONS GET CALLED IN: LEFT-TO-RIGHT OR RIGHT-TO-LEFT

  WHEN FUNCTIONS HAVE THE *SAME* PRECEDENCE
  ```

-
```
var a = 2, b = 3, c = 4;

a = b = c;

console.log(a);
console.log(b);
console.log(c);
```
- Output : 4, 4, 4
- As equal to operator has right to left associativity

Coercion

-


This happens quite often in Javascript because it's dynamically typed.

-
```
var a = 1 + '2';
console.log(a);
```
- Output -> "12"
- First parameter gets coerced(type casted) to string

Comparison Operators

```
console.log(3 < 2 < 1);
```

- Output : true
- "<" has left-to-right associativity
- First it checks 3 < 2 which returns false, false will be coerced to number (0)
- Then it checks 0 < 1 which returns true

-
```
> Number(undefined)
< NaN
> Number(null)
< 0
```

```
> null == 0
< false
> null < 1
< true
> "" == 0
< true
```

```
"" == false
true
```

- To solve the above we have to use Strict Equality(===) or Strict Inequality(!==), which does a type check along with the valu e

```
> 3 === 3
< true
> "3" === "3"
< true
> "3" === 3
< false
```

Existence and Booleans

```
> Boolean(undefined)
< false
> Boolean(null)
< false
> Boolean("")
< false
```

```
var a;

// goes to internet and looks for a
value

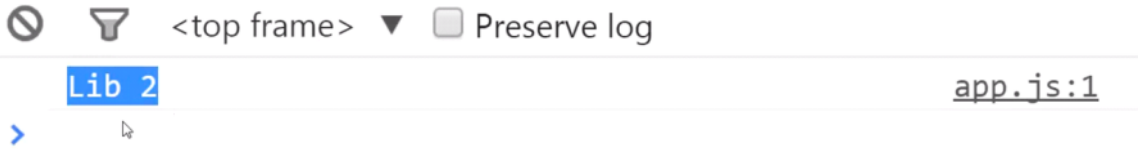
if (a) {
  console.log('Something is there.');
}
```

Default Values

```
function greet(name) {
    console.log(name);
    console.log('Hello ' + name);
}

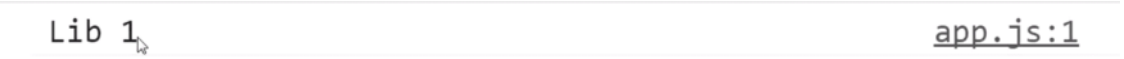
greet();
```

- Output -> undefined, Hello undefined
- To avoid the above undefined, we can provide default values for the arguments in ES6
- Also, before ES6 we can provide defaults as below

```
function greet(name) {
    name = name || '<Your name here>';
    console.log('Hello ' + name);
}

greet();
```

- Because the OR operation converts as below

```
> 0 || 1
< 1
> undefined || "hello"
< "hello"
> null || "hello"
< "hello"
```

```
function greet(name) {
    name = name || '<Your name here>';
    console.log('Hello ' + name);
}

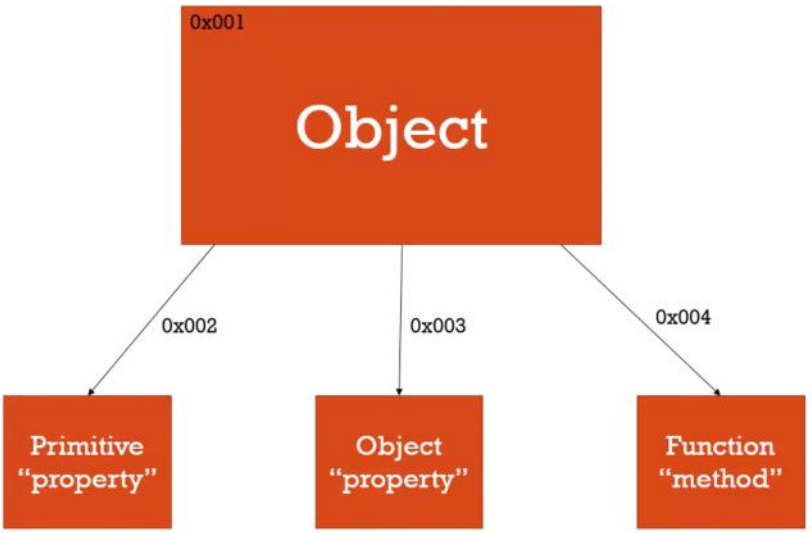
greet('Tony');
greet();
```

- Output is as below

```
Hello Tony                                  app.js:3
Hello <Your name here>                      app.js:3
```

- Another example on how defaults help when different libraries use the same variable name

```
<html>
    <head>

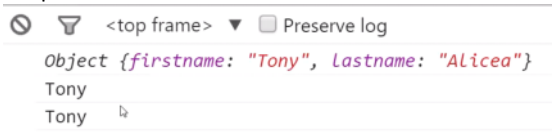
    </head>
    <body>
        <script src="lib1.js"></script>
        <script src="lib2.js"></script>
        <script src="app.js"></script>
    </body>
</html>
```

- Lib1.js
  - ```
    var libraryName = "Lib 1";
    ```
- Lib2.js
  - ```
    var libraryName = "Lib 2";
    ```
- App.js
  - ```
    console.log(libraryName);
    ```
    ```

- Output ->
  - 

- Generally the code stack upon one another when three different files are used.
- Also in production it is generally bundled into a single file
- So, the variable is stored in a window object in the global execution when lib1.js is executed.
- To avoid the above, we can update lib2.js as below
  - ```
    window.libraryName = window.libraryName || "Lib 2";
    ```
- The above checks if the variable is already created in the Global execution, if it is then it uses the same or it uses the va lue mentioned in lib2.js
- Now output will be
  - 

Object and Functions

- Objects are name value pairs which has properties and methods and those can also be name and value pairs.
- 

- The core object will have some address in the computer's memory and it will have references to the properties and methods tha t are connected to it which are also sitting the computer's memory as shown in the above screenshot.
- To access those objects and its properties and methods, we can do it in the following way
  - ```
    var person = new Object();

    person["firstname"] = "Tony";
    person["lastname"] = "Alicea";

    var firstNameProperty = "firstname";

    console.log(person);
    console.log(person[firstNameProperty]);

    console.log(person.firstname);
    ```
  - Output -
  - 

  - Computed Member Access -> []
  - Member Access -> .

```
13
14 person.address = new Object();
15 person.address.street = "111 Main St.";
16 person.address.city = "New York";
17 person.address.state = "NY";
18
19 console.log(person.address.street);
20 console.log(person.address.city);
21 console.log(person["address"]["state"]);
22
```

- Output-

```
Alicea                                  app.js:12
111 Main St.                            app.js:19
New York                                app.js:20
NY                                      app.js:21
```

- **Object Literals**
  - Object can also be written as below

```
1 var person = {};
2 console.log(person);
```

  - The above is a short hand for creating a new object
  - Js engine when it parses the syntax and comes across the curly braces, it assumes we are creating an object

```
1 var person = { firstname: 'Tony', lastname: 'Alicea' };
2 console.log(person);
```

  - Output ->
  - *Object {firstname: "Tony", lastname: "Alicea"}*

```
1  var person = {
2      firstname: 'Tony',
3      lastname: 'Alicea',
4      address: {
5          street: '111 Main St.',
6          city: 'New York',
7          state: 'NY'
8      }
9  };
10
11 console.log(person);
12
```

  - Output-

```
                                                                    app.js:11
▼ Object {firstname: "Tony", lastname: "Alicea", address: Object}
  ▼ address: Object
      city: "New York"
      state: "NY"
      street: "111 Main St."
    ▶ __proto__: Object
    firstname: "Tony"
    lastname: "Alicea"
  ▶ __proto__: Object
```

- Faking Namespaces

- 


```
var greet = 'Hello!';
var greet = 'Hola!';

console.log(greet);

var english = {};
var spanish = {};

english.greet = 'Hello!';
spanish.greet = 'Hola!';

console.log(english)
```

```
Hola!                                    app.js:4
Object {greet: "Hello!"}                 app.js:12
```

- JSON and Object Literals
  - JSON (JavaScript Object Notation)
  - The below explains the difference between object literals and json string

```
var objectLiteral = {
    firstname: 'Mary',
    isAProgrammer: true
}

console.log(JSON.stringify(objectLiteral));

var jsonValue = JSON.parse('{ "firstname": "Mary", "isAProgrammer": true }');

console.log(jsonValue);
```

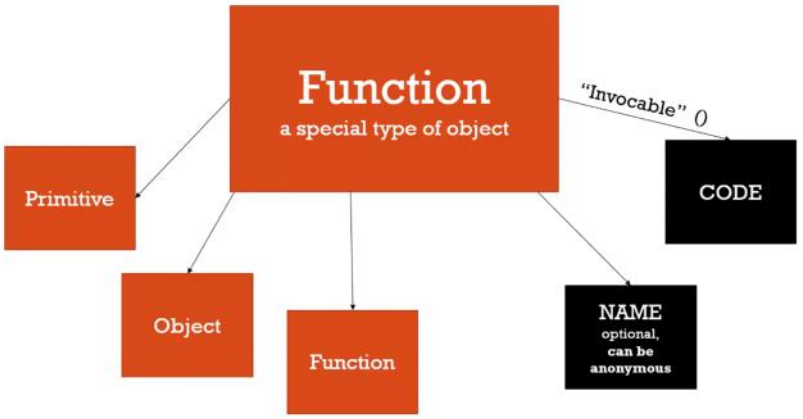
  - Output:

```
{"firstname":"Mary","isAProgrammer":true}
Object {firstname: "Mary", isAProgrammer: true}
```

- **Functions are objects in Javascript**

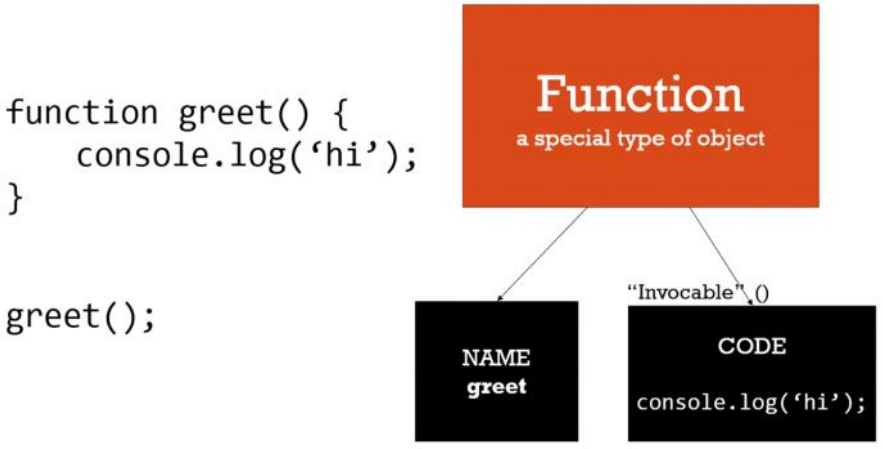
- **FIRST CLASS FUNCTIONS: EVERYTHING YOU CAN DO WITH OTHER TYPES YOU CAN DO WITH FUNCTIONS.**

  Assign them to variables, pass them around, create them on the fly.

- 


- ```
  function greet() {
      console.log('hi');
  }

  greet.language = 'english';
  console.log(greet.language);
  ```

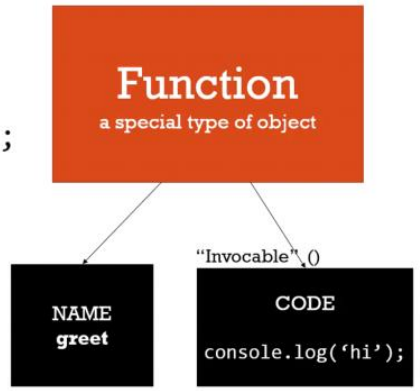
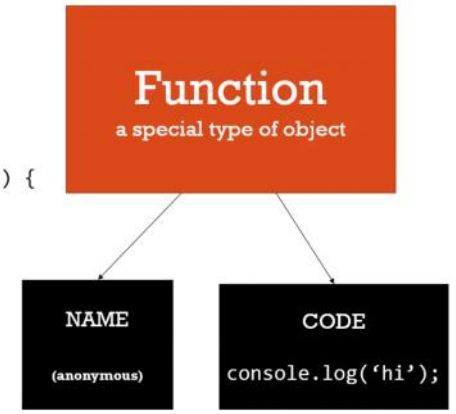
- Output: "english"

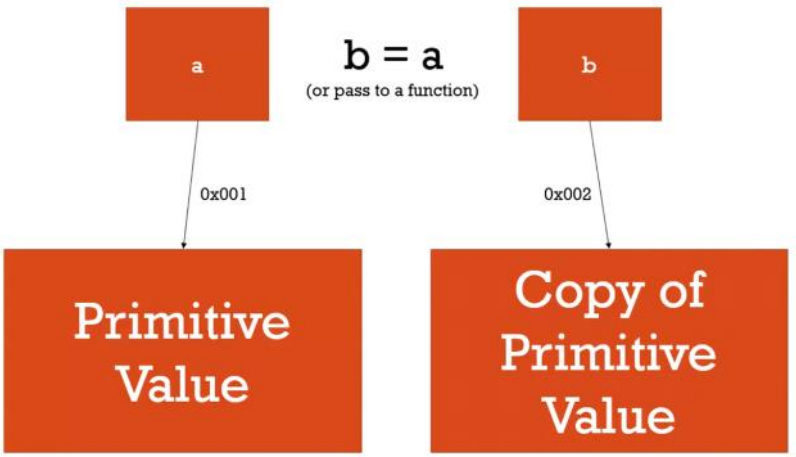
- 

- **Functional Statements and Functional Expressions**
  ```

- **EXPRESSION:**
  A UNIT OF CODE THAT RESULTS IN A VALUE

  It doesn't have to save to a variable.
- Expression is something which returns a value.

```
> a = 3;
<- 3
> 1 + 2;
<- 3
> a = { greeting: 'hi' };
<- Object {greeting: "hi"}
```

- Statements are as below

```
if (a === 3) {

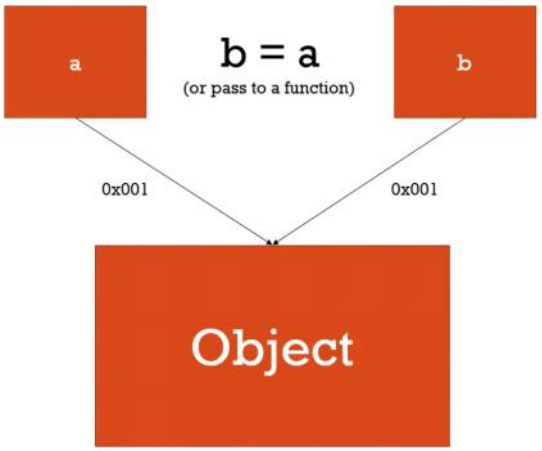
}
```

- Functional Statement
  - For this function, we will have an object created in Global Execution Context of Creation Phase
  - Object has a name called "greet"
  - It has code which can be invoke through the name followed by parenthesis. Ex greet()

```
greet();

function greet() {
    console.log('hi');
}
```

```
function greet() {
    console.log('hi');
}

greet();
```



- Functional Expression
  - Here in this case, the object is put in memory and function is referenced via that address.
  - It does not look at the name but it uses the reference, so we call it an anonymous function.
  - Here the code is invoked by using the object name followed by parenthesis. Ex. anonymousGreet()

```
var anonymousGreet = function() {
    console.log('hi');
}
```




```
var anonymousGreet = function() {
    console.log('hi');
}

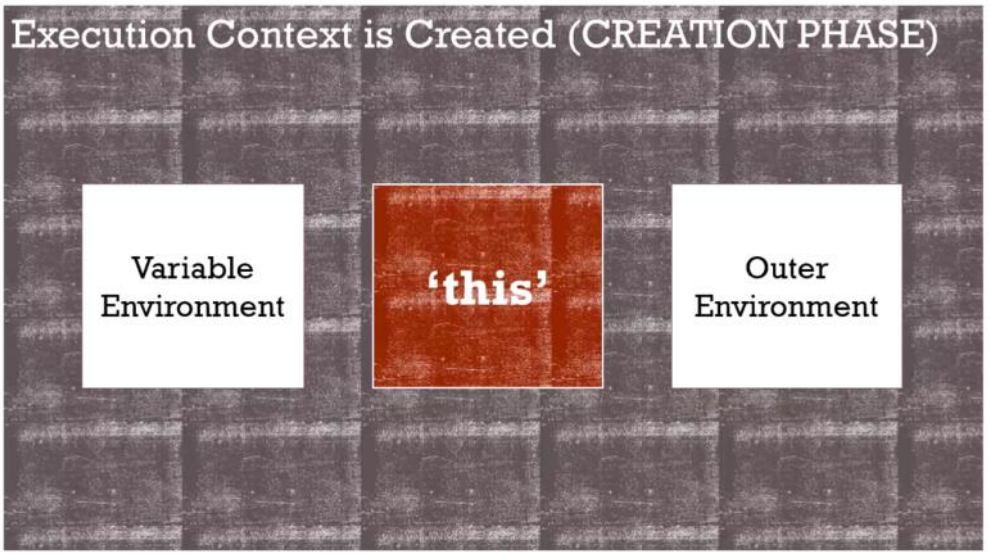
anonymousGreet();
```

```
13 function log(a) {
14     console.log(a);
15 }
16
17 log(function() {
18     console.log('hi');
19 });
20
```

- Output :

```
function () {                                   app.js:14
    console.log('hi');
}
```

```
13 function log(a) {
14     a();
15 }
16
17 log(function() {
18     console.log('hi');
19 });
```

- Output:

```
hi                                              app.js:18
```

- **By Value** vs **By Reference**



- If we set a variable with a primitive value(String, Boolean, Number)
- So the variable will have a address location where the primitive value sits in memory.
- Reference here is just to a location in memory.
- If we assign this variable to another variable / pass it to a function.

- The new variable points to a new address, a new location in memory.
- Ex. If var a = 3, and if we set var b = a;, then a and b will have value as 3 but will be different address locations and b will have a copy of 3 in its address location in memory.
- This approach is called "**By Value**"
-
```
// by value (primitives)
var a = 3;
var b;

b = a;
a = 2;

console.log(a);
console.log(b);
```
  - Output ->
  - 2 app.js:8
  - 3 app.js:9
  - As values are stored in two different locations, even if the value of a is changed it does not affect the value of b.

- Now If we follow the above for an object
- 

- When a variable is created with an object and assigned to a new variable/passed to a function, it does not create a new address location in memory instead it refers to the same address location in memory when the first variable is created.
- No copy of the object is created. Instead two names point to the same address.
- This is called "**By Reference**".
- All objects interact "**By Reference**"
- 

-
```
// by reference (all objects (including functions))
var c = { greeting: 'hi' };
var d;

d = c;
c.greeting = 'hello'; // mutate

console.log(c);
console.log(d);
```
  - Output ->

```
Object {greeting: "hello"}                    app.js:18
Object {greeting: "hello"}                    app.js:19
```

- Both "c" and "d" display the same as they point to the same location in memory.

```
// by reference (even as parameters)
function changeGreeting(obj) {
    obj.greeting = 'Hola'; // mutate
}

changeGreeting(d);
console.log(c);
console.log(d);
```

- Output ->

```
Object {greeting: "hello"}                    app.js:18
Object {greeting: "hello"}                    app.js:19
Object {greeting: "Hola"}                     app.js:27
Object {greeting: "Hola"}                     app.js:28
```

```
// equals operator sets up new memory space (new address)
c = { greeting: 'howdy' };
console.log(c);
console.log(d);
```

- Output ->

```
Object {greeting: "howdy"}                    app.js:32
Object {greeting: "Hola"}                     app.js:33
```

- **c** display "howdy" but **d** still displays "Hola", as the object assigned to c using equal to operator sets up a new memory space
- This is a special case where "By Reference" does not work for an object.

- **Objects, Functions and 'this'**



- Variable Environment -> Environment to know where the variable created inside the function sits
- Outer Environment -> Environment to know what the lexical environment of the variables is, if it is not found in its current scope, through scope chain it finds out until the global environment is reached.
- Along with the above, we have 'this' context also created which refers to different objects based on the how the function is invoked.
- Below are some examples of 'this' in different contexts

```
console.log(this);
```

- Output ->

```
▶ Window                                      app.js:1
```

-
```
function a() {
    console.log(this);
}

a();
```
  - Output
  -
```
app.js:2
▼Window {top: Window, window: Window, location: Location, external: Obj
    Infinity: Infinity
  ▶ AnalyserNode: function AnalyserNode() { [native code] }
  ▶ ApplicationCache: function ApplicationCache() { [native code] }
  ▶ ApplicationCacheErrorEvent: function ApplicationCacheErrorEvent() {
  ▶ Array: function Array() { [native code] }
  ▶ ArrayBuffer: function ArrayBuffer() { [native code] }
  ▶ Attr: function Attr() { [native code] }
  ▶ Audio: function HTMLAudioElement() { [native code] }
```
-
```
function a() {
    console.log(this);
}

var b = function() {
    console.log(this);
}

a();
b();
```
  - Output
  -
```
app.js:2
▼Window {top: Window, window: Window, location: Location, external: Obj
    Infinity: Infinity
  ▶ AnalyserNode: function AnalyserNode() { [native code] }
  ▶ ApplicationCache: function ApplicationCache() { [native code] }
  ▶ ApplicationCacheErrorEvent: function ApplicationCacheErrorEvent() {
  ▶ Array: function Array() { [native code] }
  ▶ ArrayBuffer: function ArrayBuffer() { [native code] }
  ▶ Attr: function Attr() { [native code] }
  ▶ Audio: function HTMLAudioElement() { [native code] }
```
-
```
var c = {
    name: 'The c object',
    log: function() {
        console.log(this);
    }
}

c.log();
```
  - Output ->
  - In this context, the 'this' key word points out to the object that contains it
  - Ex.
    -
```
var c = {
    name: 'The c object',
    log: function() {
        this.name = 'Updated c object';
        console.log(this);
    }
}

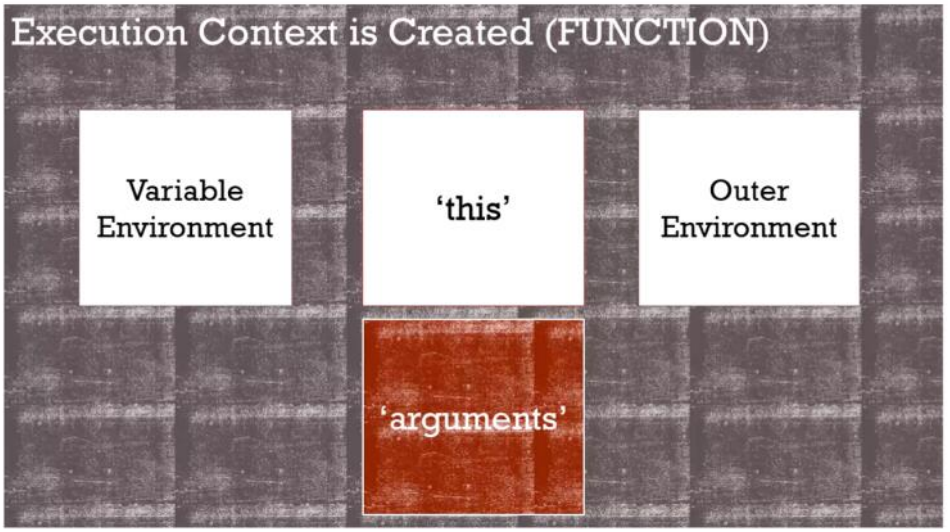
c.log();
```
    - Output ->
    -
```
▼ Object {name: "Updated c object", log: function}   app.js:20
```
  - So by the above example, we can see that the object properties can be mutated by using the this keyword with the property name, as this points to the current object it is contained in.

```
var c = {
    name: 'The c object',
    log: function() {
        this.name = 'Updated c object';
        console.log(this);

        var setname = function(newname) {
            this.name = newname;
        }
        setname('Updated again! The c object');
        console.log(this);
    }
}

c.log();
```

- Output ->

```
▶ Object {name: "Updated c object", log: function}        app.js:20
▶ Object {name: "Updated c object", log: function}        app.js:26
```

- In the above example, the name property of the c object is not updated through the setname function
- If we check the window object, it would have created the name property with value as "Updated again! The c object".
- This is because the internal function when it's execution context is created, the this keyword points to the global object.
- To overcome the above we can use the below code, where we assign the 'this' to another variable 'self'

```
var c = {
    name: 'The c object',
    log: function() {
        var self = this;

        self.name = 'Updated c object';
        console.log(self);

        var setname = function(newname) {
            self.name = newname;
        }
        setname('Updated again! The c object');
        console.log(self);
    }
}
```

- Output ->

```
▶ Object {name: "Updated c object", log: function}        app.js:22
▶ Object {name: "Updated again! The c object", log: function}   app.js:28
```

- **Arrays - Collections of anything**
  - Array can be collection of any types and can initialized in any of the following ways:
    - var arr = []; var arr2 = new Array(); var arr3 = [1, 2, 3];

```
var arr = [
    1,
    false,
    {
        name: 'Tony',
        address: '111 Main St.'
    },
    function(name) {
        var greeting = 'Hello';
        console.log(greeting + name);
    },
    "hello"
];

console.log(arr);
arr[3](arr[2].name)
```

- Output ->

```
▶ [1, false, Object, function, "hello"]   app.js:15
HelloTony                                 app.js:10
```

- **Arguments and Spread**
  - 

  - **ARGUMENTS: THE PARAMETERS YOU PASS TO A FUNCTION**

    Javascript gives you a keyword of the same name which contains them all.
  -
```
function greet(firstname, lastname, language) {

    console.log(firstname);
    console.log(lastname);
    console.log(language);
    console.log('-------------');

}

greet();
greet('John');
greet('John', 'Doe');
greet('John', 'Doe', 'es');
```
  - We can call a function which needs arguments by without passing the arguments also.
  - It displays undefined for the arguments not passed.
  - This is taken care by js engine, as soon as it sees the arguments accepted by the function it creates those variables sets them up in the memory.
  - In modern js(ES6), we have the below option to set default when the argument is not passed to the function
    -
```
function greet(firstname, lastname, language = 'en') {
```
    - But this is not yet available to all the browsers and it does not work on old browsers as well.
    - So, we can use the following to provide defaults if the argument is not provided

```
function greet(firstname, lastname, language) {

    language = language || 'en';

    console.log(firstname);
    console.log(lastname);
    console.log(language);
    console.log('-------------');

}

greet();
greet('John');
greet('John', 'Doe');
greet('John', 'Doe', 'es');
```

- **Spread Operator** (...)

```
function greet(firstname, lastname, language, ...other) {

    language = language || 'en';

    if (arguments.length === 0) {
        console.log('Missing parameters!');
        console.log('-------------');
        return;
    }

    console.log(firstname);
    console.log(lastname);
    console.log(language);
    console.log(arguments);
    console.log('arg 0: ' + arguments[0]);
    console.log('-------------');

}
```

```
greet();
greet('John');
greet('John', 'Doe');
greet('John', 'Doe', 'es', '111 main st', 'new york');
```

- **Function Overloading**

```
function greet(firstname, lastname, language) {

    language = language || 'en';

    if (language === 'en') {
        console.log('Hello ' + firstname + ' ' + lastname);
    }

    if (language === 'es') {
        console.log('Hola ' + firstname + ' ' + lastname);
    }

}

greet('John', 'Doe', 'en');
greet('John', 'Doe', 'es');
```

- Output ->

```
Hello John Doe                    app.js:6
Hola John Doe                     app.js:10
```

- What if we want to have two functions for english and spanish
- One pattern is to create different function for english and spanish, we have other approaches too for this function overloading

```
15 function greetEnglish(firstname, lastname) {
16     greet(firstname, lastname, 'en');
17 }
18
19 function greetSpanish(firstname, lastname) {
20     greet(firstname, lastname, 'es');
21 }
22
23 greetEnglish('John', 'Doe');
24 greetSpanish('John', 'Doe');
```

- **Syntax Parsers**
  - This is checked character by character using a set of rules checking for what is the valid syntax, and if the character that it is anticipates does not match it throws an error.

- **Automatic Semicolon Insertion**
  - Semicolon are optional in core javascript.
  - When you write a return statement without a semicolon, js engine reads character by character and it knows what the language expects and it knows what the syntax looks like. So, if it sees we are finishing a line that is a carriage return(when we hit enter on keyboard). Carriage return is an invisible character but it is a character so the syntax sees it and recognizes the carriage return and as it realizes the line finished it automatically assigns a semi-colon.

```
1  function getPerson() {
2
3      return
4      {
5          firstname: 'Tony'
6      }
7
8  }
9
10 console.log(getPerson());
```

  - Output ->

```
<top frame>  Preserve log
undefined                         app.js:10
```

  - The above results in "undefined" because of the automatic semicolon insertion in return statement .
  - Because of the carriage return after the end the "return" keyword.
  - It recognizes as in the below code and returns undefined.

```
1  function getPerson() {
2
3      return;
4      {
5          firstname: 'Tony'
6      }
7
8  }
9
10 console.log(getPerson());
```

  - To prevent the above we have to modify the code as below

```
function getPerson() {

    return {
        firstname: 'Tony'
    }

}

console.log(getPerson());
```

- So this helps the js engine recognize that there is a space and an object literal after the return keyword and returns the object without automatically insertion of semicolon.

- **Whitespace**
  - 

  - JS is liberal about whitespaces in the code. Below is an example
  - ```
    var
        // first name of the person
        firstname,

        // last name of the person
        lastname,

        // the language
        // can be 'en' or 'es'
        language;

    var person = {
        firstname: 'John',
        lastname: 'Doe'
    }

    console.log(person);
    ```
  - JS ignores all the whitespaces and comments.

- Immediately Invoked Function Expressions(IIFEs)
  - Below are examples of function statement and function expression
  - ```
    // function statement
    function greet(name) {
        console.log('Hello ' + name);
    }
    greet();

    // using a function expression
    var greetFunc = function(name) {
        console.log('Hello ' + name);
    };
    greetFunc();
    ```
  - Inside the function expression, we use an anonymous function assigned to a variable in memory and will be invoked using the variable

with parenthesis.

- Below is an example of IIFE
  - ```
    // using an Immediately Invoked Function Expression (IIFE)
    var greeting = function(name) {
        console.log('Hello ' + name);
    }();
    ```
  - In the above code, the anonymous function is immediately invoked.
- ```
  // using an Immediately Invoked Function Expression (IIFE)
  var greeting = function(name) {
      return 'Hello ' + name;
  }('John');
  console.log(greeting);
  ```
  - Output->
  - "Hello John"
- Whenever a statement starts with a "function" keyword, the syntax parser expects a function statement with a name and it can't be an anonymous function.
  - Ex.
  - ```
    function(name) {

        return 'Hello ' + name;

    }
    ```
  - It returns an error

    ```
    Uncaught SyntaxError: Unexpected token (
    ```
    - >
- Whenever a statement is written inside parenthesis, it will not return an error as it reads that as an expression.
  - ```
    (3+4)*2;
    (function(name) {

        return 'Hello ' + name;

    });
    ```
  - ```
    (function(name) {

        return 'Hello ' + name;

    });
    ```
  - It can be self-invoked as below, hence it can be called as an IIFE or anonymous function
  - ```
    (function(name) {

        var greeting = 'Hello';
        console.log(greeting + ' ' + name);

    }());
    ```
  - Ex.
  - ```
    var firstname = 'John';

    (function(name) {

        var greeting = 'Inside IIFE: Hello';
        console.log(greeting + ' ' + name);

    }(firstname));
    ```
    - Output ->
    - ```
      Inside IIFE: Hello John                                    app.js:27
      ```
- IIFE can be invoked inside or outside the parenthesis.

```
22 var firstname = 'John';
23
24 (function(name) {
25
26     var greeting = 'Inside IIFE: Hello';
27     console.log(greeting + ' ' + name);
28
29 }(firstname));
```

```
22 var firstname = 'John';
23
24 (function(name) {
25
26     var greeting = 'Inside IIFE: Hello';
27     console.log(greeting + ' ' + name);
28
29 })(firstname); // IIFE
```

```
(function(name) {

    var greeting = 'Hello';
    console.log(greeting + '
' + name);

}('John'));
```

- Behind the scenes,
- First the global execution context is created, as the function is anonymous and has no name it just takes the parenthesis.
- Second it sees the argument "John" in the parenthesis and the function gets invoked, and it creates an function execution context with the variable greeting.
- So, this approach will have nothing in the global execution context and everything sits in the function execution context.

```
var greeting = 'Hola';

(function(name) {

    var greeting = 'Hello';
    console.log(greeting + '
' + name);

}('John'));
```

- In the above, we can see we have two variables with the same name
- One sits in the global execution context
- Another in the IIFE execution context.
- Variable inside IIFE does not conflict with the variable in the global execution context
- Hence, widely used in frameworks and libraries.

- Any reference required by the anonymous function can be passed as an argument to the anonymous function

```
// IIFE
(function(global, name) {

    var greeting = 'Hello';
    console.log(greeting + ' ' + name);

}(window, 'John')); // IIFE

console.log(greeting);
```

- In the above code, we can send the global window object to the anonymous function
- As we know that object work by reference, any changes in the object in the anonymous function will reflect in the global execution as well.

```
// IIFE
(function(global, name) {

    var greeting = 'Hello';
    global.greeting = 'Hello';
    console.log(greeting + ' ' + name);

}(window, 'John')); // IIFE

console.log(greeting);
```

- **Closures**

```
function greet(whattosay) {

    return function(name) {
        console.log(whattosay + ' ' + name);
    }

}

greet('Hi')('Tony');
```

- Output -> "Hi Tony"

```
function greet(whattosay) {

    return function(name) {
        console.log(whattosay + ' ' + name);
    }

}

var sayHi = greet('Hi');
sayHi('Tony');
```

- Output -> "Hi Tony"

```
function greet(whattosay) {

   return function(name) {
      console.log(whattosay + ' '
+ name;
   }

}

var sayHi = greet('Hi');
sayHi('Tony');
```

- Initially the Global execution context is created
- It executes the variable sayHi and invokes greet function
- Then the greet function execution context is created
- It has whattosay variable created with 'Hi' value.
- Once the return statement in greet function is executed, the greet function execution context is popped out of the stack.
- All the variable memory spaces created in the function execution context will be cleared out once the function execution context is popped out of the stack. This process is called "**Garbage Collection**".
- But here as shown in the below screenshot, the variable is not garbage collected

```
function greet(whattosay) {

   return function(name) {
      console.log(whattosay + ' '
+ name;
   }

}

var sayHi = greet('Hi');
sayHi('Tony');
```

- Next it invokes the sayHi function in the global execution context, which creates a sayHi function execution context

```
function greet(whattosay) {

   return function(name) {
      console.log(whattosay + ' '
+ name;
   }

}

var sayHi = greet('Hi');
sayHi('Tony');
```

- When the sayHi function is invoked and executes the below highlighted code

```
function greet(whattosay) {

   return function(name) {
       console.log(whattosay + ' '
+ name;
   }

}

var sayHi = greet('Hi');
sayHi('Tony');
```

- When it sees the whattosay variable, the js engine goes up the scope chain until it finds the variable in the outer lexical environment.
- Since it did not find inside the sayHi execution context, it goes upto the next in the scope chain the greet function execution context.
- Even though the greet function execution context is popped out of the stack, the sayHi execution context still has a reference to the variables in the memory space of the outer environment as in the below screenshot.

```
function greet(whattosay) {

   return function(name) {
       console.log(whattosay + ' '
+ name;
   }

}

var sayHi = greet('Hi');
sayHi('Tony');
```

- So, the phenomena of closing in all the variables the function is supposed to have to access to even though the execution context is popped out of the stack is called Closure.
- Closure is feature of js language, it gives access to the outer environment variables irrespective on whether the outer function is still running or not.

```
function buildFunctions() {

    var arr = [];

    for (var i = 0; i < 3; i++) {

        arr.push(
            function() {
                console.log(i);
            }
        )

    }

    return arr;
}

var fs = buildFunctions();

fs[0]();
fs[1]();
fs[2]();
```

- Output ->

```
3                                                    app.js:9
3                                                    app.js:9
3                                                    app.js:9
```

```
function buildFunctions() {

    var arr = [];

    for (var i = 0; i < 3; i++) {
        arr.push(function() {
            console.log(i);
        });
    }

    return arr;

}

var fs = buildFunctions();

fs[0]();
fs[1]();
fs[2]();
```

- First the global execution context is created with fs and invoking of buildFunctions.
- `var fs = buildFunctions();`
- Once the above is executed, the buildFunctions execution context is created with variables "i" and "arr".
- So the for loop runs and pushes the function into the array "arr" with the console.log statement.
- But the console.log will not run while being pushed into the array.
- We are just creating a new function object and pushing it into the array.
- Loop runs until i is 2, then when i is incremented and has become 3 and it checks i < 3 and exits out of the for loop.
- So by the time the function is executed, i is 3 and arr is created with 3 function objects.

```
function buildFunctions() {

    var arr = [];

    for (var i = 0; i < 3; i++) {
       arr.push(function() {
            console.log(i);
       });
    }

    return arr;

}

var fs = buildFunctions();

fs[0]();
fs[1]();
fs[2]();
```

- Now the buildFunctions execution context is popped out of the stack, while the variable i and arr will be still hanging around in the memory.
- Next line fs[0](); will be executed and invoked.
- Now it checks for variable i in the execution context, when it does not find in the current function execution context, it goes to the next execution context in the scope chain to find the variable "i".
- It finds i as 3, which is in the memory of buildFunctions execution context.

```
function buildFunctions() {

    var arr = [];

    for (var i = 0; i < 3; i++) {
       arr.push(function() {
            console.log(i);
       });
    }

    return arr;

}

var fs = buildFunctions();

fs[0]();
fs[1]();
fs[2]();
```

- It moves on to the next once fs[0]() execution context is done

```
function buildFunctions() {

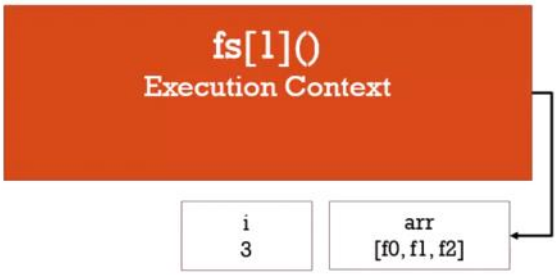
    var arr = [];

    for (var i = 0; i < 3; i++) {
       arr.push(function() {
             console.log(i);
       });
    }

    return arr;

}

var fs = buildFunctions();

fs[0]();
fs[1]();
fs[2]();
```




```
function buildFunctions() {

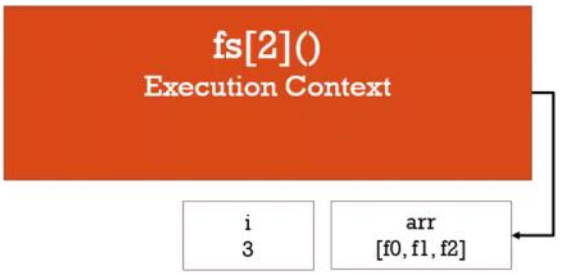
    var arr = [];

    for (var i = 0; i < 3; i++) {
       arr.push(function() {
             console.log(i);
       });
    }

    return arr;

}

var fs = buildFunctions();

fs[0]();
fs[1]();
fs[2]();
```



- To fix the above, we need to write the code in the following way using an IIFE, passing i to the IIFE

```
function buildFunctions2() {

    var arr = [];

    for (var i = 0; i < 3; i++) {
        arr.push(
            (function(j) {
                return function() {
                    console.log(j);
                }
            }(i))
        )

    }

    return arr;
}

var fs2 = buildFunctions2();

fs2[0]();
fs2[1]();
```

- The above highlighted code runs an IIFE which returns the function which consoles the variable
  - Output ->

```
0                                   app.js:32
1                                   app.js:32
2                                   app.js:32
```

- **Function factories**
  - ```
    function makeGreeting(language) {

        return function(firstname, lastname) {

            if (language === 'en') {
                console.log('Hello ' + firstname + ' ' + lastname);
            }

            if (language === 'es') {
                console.log('Hola ' + firstname + ' ' + lastname);
            }

        }

    }

    var greetEnglish = makeGreeting('en');
    var greetSpanish = makeGreeting('es');

    greetEnglish('John', 'Doe');
    greetSpanish('John', 'Doe');
    ```
    - Output ->
    - ```
      Hello John Doe                    app.js:6
      Hola John Doe                     app.js:10
      ```
  - The above is a function factory.
  - makeGreeting is a function which accepts language as an argument and returns the inner function to the outer function makeGreeting.
  - Language will be trapped in the closure and will be used in the inner function, even though the execution context of the outer function is popped out of the stack, the language variable reference will still be available in memory and the inner function can use it.
  - Two variables greetEnglish and greetSpanish which calls makeGreeting function with the respective languages passed.
  - Each variable will have it's own execution context of makeGreeting function, creates a new memory space and it will hold the inner function
  - ```
    function makeGreeting(language) {

        return function(firstname,
    lastname) {

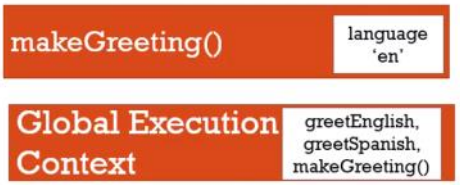
            if (language === 'en') {
                console.log('Hello ' +
    firstname + ' ' + lastname);
            }

            if (language === 'es') {
                console.log('Hola ' +
    firstname + ' ' + lastname);
            }

        }

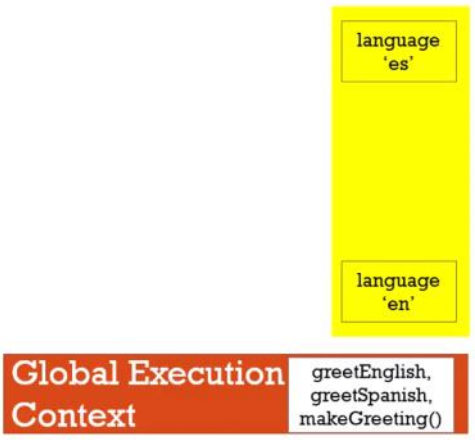
    }

    var greetEnglish = makeGreeting('en');
    var greetSpanish = makeGreeting('es');

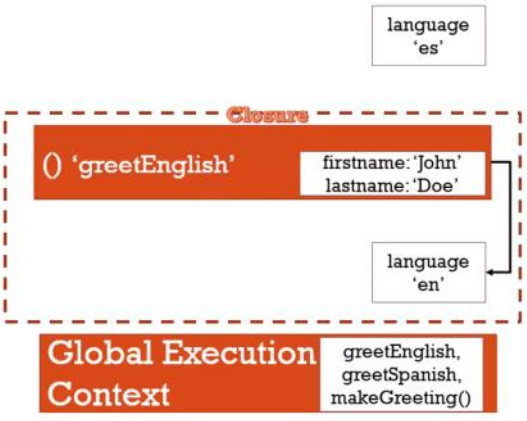
    greetEnglish('John', 'Doe');
    greetSpanish('John', 'Doe');
    ```
    

    ```

```
function makeGreeting(language) {

    return function(firstname,
lastname) {

        if (language === 'en') {
            console.log('Hello ' +
firstname + ' ' + lastname);
        }

        if (language === 'es') {
            console.log('Hola ' +
firstname + ' ' + lastname);
        }

    }

}

var greetEnglish = makeGreeting('en');
var greetSpanish = makeGreeting('es');

greetEnglish('John', 'Doe');
greetSpanish('John', 'Doe');
```

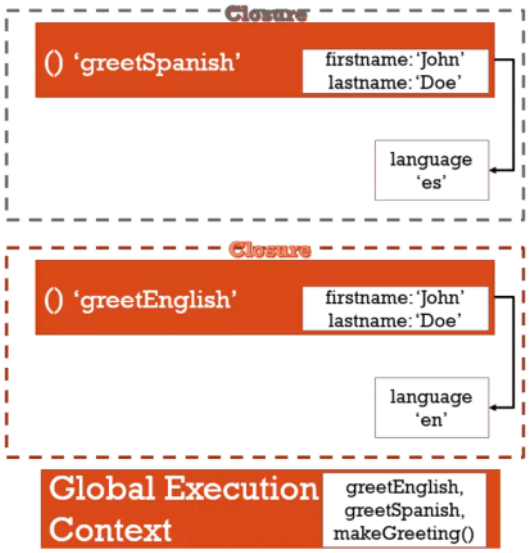

```
function makeGreeting(language) {

    return function(firstname,
lastname) {

        if (language === 'en') {
            console.log('Hello ' +
firstname + ' ' + lastname);
        }

        if (language === 'es') {
            console.log('Hola ' +
firstname + ' ' + lastname);
        }

    }

}

var greetEnglish = makeGreeting('en');
var greetSpanish = makeGreeting('es');

greetEnglish('John', 'Doe');
greetSpanish('John', 'Doe');
```




```
function makeGreeting(language) {

    return function(firstname,
lastname) {

        if (language === 'en') {
            console.log('Hello ' +
firstname + ' ' + lastname);
        }

        if (language === 'es') {
            console.log('Hola ' +
firstname + ' ' + lastname);
        }

    }

}

var greetEnglish = makeGreeting('en');
var greetSpanish = makeGreeting('es');

greetEnglish('John', 'Doe');
greetSpanish('John', 'Doe');
```



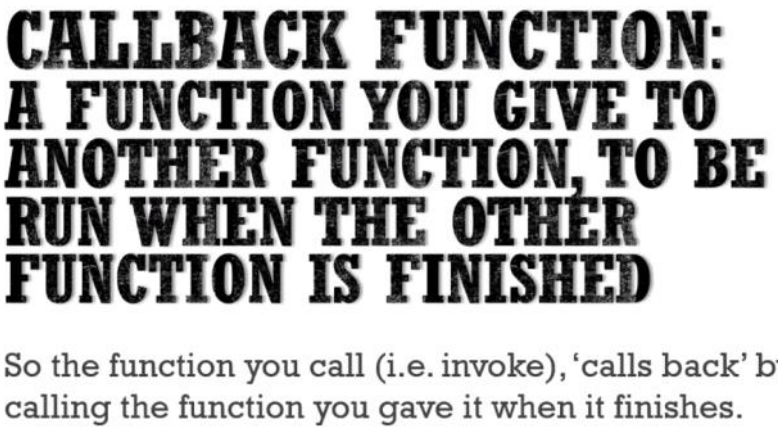
```
function makeGreeting(language) {

    return function(firstname,
lastname) {

        if (language === 'en') {
            console.log('Hello ' +
firstname + ' ' + lastname);
        }

        if (language === 'es') {
            console.log('Hola ' +
firstname + ' ' + lastname);
        }

    }

}

var greetEnglish = makeGreeting('en');
var greetSpanish = makeGreeting('es');

greetEnglish('John', 'Doe');
greetSpanish('John', 'Doe');
```



- **Closures and callbacks**

```
function sayHiLater() {

    var greeting = 'Hi!';

    setTimeout(function() {

        console.log(greeting);

    }, 3000);

}

sayHiLater();

// jQuery uses function expressions and first-class functions!
//$("button").click(function() {
//
//});
```

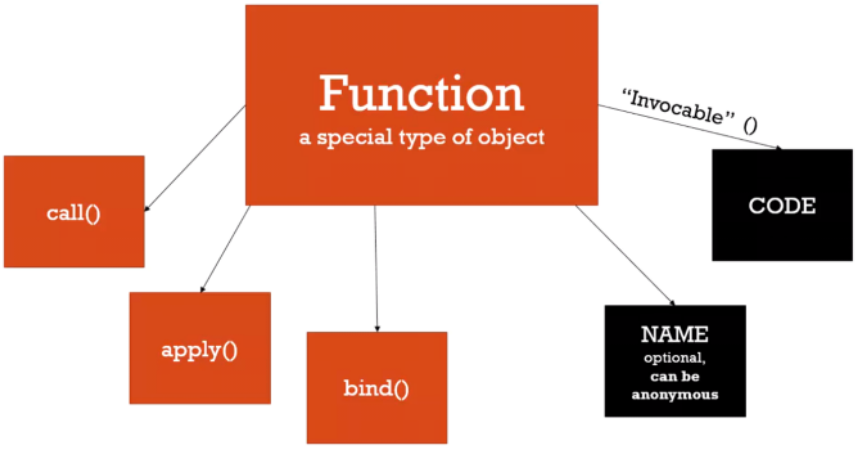
  - The output "Hi!" is logged after 3 seconds.
  - This is because, even though the sayHiLater function execution context is popped out of the stack, the greeting variable will still be in the memory so that setTimeout function can use it.
  - setTimeout is an example of function expression and first-class functions.
  - The parenthesis after setTimeout keyword indicates a function expression.
  - The same implementation is used in jQuery click functions
  - setTimeout here is a callback function for the function expression.



```
function tellMeWhenDone(callback) {

    var a = 1000; // some work
    var b = 2000; // some work

    callback(); // the 'callback', it runs the function I give it!

}

tellMeWhenDone(function() {

    console.log('I am done!');

});

tellMeWhenDone(function() {

    console.log('I am done!');

});
```

- Above is an example of callback function.

- **call(), apply() and bind()**



- A function has the properties as mentioned above
- Below is an illustration of call(), apply() and bind()
- Consider the below example for bind()

```
var person = {
    firstname: 'John',
    lastname: 'Doe',
    getFullName: function() {

        var fullname = this.firstname + ' ' + this.lastname;
        return fullname;

    }
}

var logName = function(lang1, lang2) {

    console.log('Logged: ' + this.getFullName());

}

logName();
```

- Here, when logName() is invoked, it returns the following error as "this" refers to the global execution context in the logName function which is undefined.

```
Uncaught TypeError: undefined is not a function        app.js:14
```

- So, what if we control the context of this to what it points to
- To achieve the above we can the code as the following using bind
- ```
  var person = {
      firstname: 'John',
      lastname: 'Doe',
      getFullName: function() {

          var fullname = this.firstname + ' ' + this.lastname;
          return fullname;

      }
  }

  var logName = function(lang1, lang2) {

      console.log('Logged: ' + this.getFullName());

  }

  var logPersonName = logName.bind(person);

  logPersonName();
  ```
- Output ->
- ```
  Logged: John Doe                                    app.js:14
  ```
- Above can also be written as in the below screenshot and call logName() directly.
- ```
  var logName = function(lang1, lang2) {

      console.log('Logged: ' + person.getFullName());

  }.bind(person);
  ```
  (person, logPersonName)
- Bind is a function object method and is always available inside it.
- We are using the logName as an object and calling the bind method of that object.
- Here we are binding the person object as the "this" variable.
- The bind method returns a new function, so it actually makes a copy if the logName function and sets this new object name function in the copy
- So whenever the execution context is created, the js engine sees that the function was created using the bind object method and sets up that "this" variable should point to person object.

- Example for call()
- ```
  var logName = function(lang1, lang2) {

      console.log('Logged: ' + this.getFullName());
      console.log('Arguments: ' + lang1 + ' ' + lang2);
      console.log('-----------');

  }

  var logPersonName = logName.bind(person);
  logPersonName('en');

  logName.call(person, 'en', 'es');
  ```
- Output ->
- ```
  Logged: John Doe                                    app.js:14
  Arguments: en undefined                             app.js:15
  -----------                                         app.js:16
  Logged: John Doe                                    app.js:14
  Arguments: en es                                    app.js:15
  -----------                                         app.js:16
  ```
- Call actually executes/invokes the function without making a copy but bind makes a copy of the function.
- The first argument is to set the context of "this" variable which is set to person in the above case
- The first argument is followed by the remaining arguments the logName function requires.

- Example for apply()
  ```

```
var logName = function(lang1, lang2) {

    console.log('Logged: ' + this.getFullName());
    console.log('Arguments: ' + lang1 + ' ' + lang2);
    console.log('-----------');

}

var logPersonName = logName.bind(person);
logPersonName('en');

logName.call(person, 'en', 'es');
logName.apply(person, ['en', 'es']);
```

- Output ->

```
Logged: John Doe                    app.js:14
Arguments: en es                    app.js:15
-----------                         app.js:16
Logged: John Doe                    app.js:14
Arguments: en es                    app.js:15
-----------                         app.js:16
```

- Apply also does the same as call
- The only difference is the arguments, as we can see above it passes the arguments as an array
- We can also use an IIFE to apply as in the below screenshot

```
(function(lang1, lang2) {

    console.log('Logged: ' + this.getFullName());
    console.log('Arguments: ' + lang1 + ' ' + lang2);
    console.log('-----------');

}).apply(person, ['es', 'en']);
```

- Scenarios in which apply can be used

```
// function borrowing
var person2 = {
    firstname: 'Jane',
    lastname: 'Doe'
}

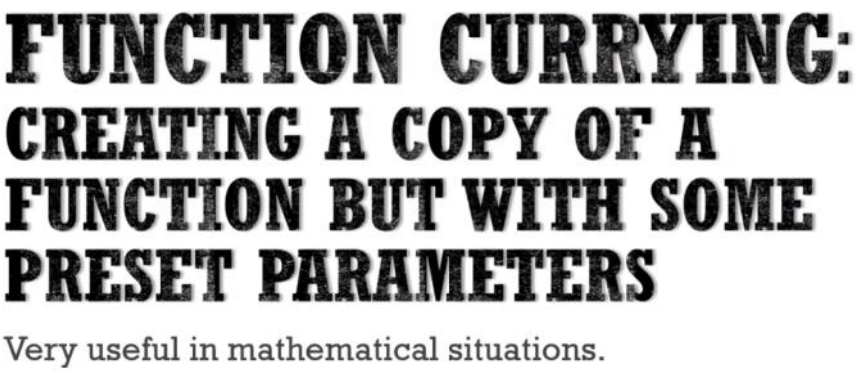
console.log(person.getFullName.apply(person2));
```

  - We created a new object called person2
  - For the person getFullName method we can set the context of this to be pointed to person2 object and display the full name of the properties in person2 object.

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

var multipleByTwo = multiply.bind(this, 2);
multipleByTwo(4);
```

  - In the above example when we bind the multiply method, it creates a copy of the function and permanently sets the value of a to 2.
  - When multipleByTwo is called it sets the value of b to 4.
  - Output -> 8

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

var multipleByTwo = multiply.bind(this, 2, 2);
console.log(multipleByTwo(5));
```

  - Output -> 4
  - a and b values are set to 2 permanently when the bind is called.
  - It doesn't matter on what we pass arguments again when we call multipleByTwo.

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

var multipleByTwo = multiply.bind(this);
console.log(multipleByTwo(5, 2));
```

- Output -> 10

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

var multipleByTwo = multiply.bind(this, 2);
console.log(multipleByTwo(4));

var multipleByThree = multiply.bind(this, 3);
console.log(multipleByThree(4));
```

- Output -> 8, 12
- The above method pre-setting the value of a is called "**Function Currying**"



- **Functional Programming**
  - Examples

```
var arr1 = [1,2,3];
console.log(arr1);

var arr2 = [];
for (var i=0; i < arr1.length; i++) {

    arr2.push(arr1[i] * 2);

}

console.log(arr2);
```

    - Output ->

```
[1, 2, 3]                                   app.js:2
[2, 4, 6]                                   app.js:11
```

  - The above can be written using functional programming,
    - Here we create a function(mapForEach) which accepts an array and a fn object as shown in the below screenshot
    - Call the function(mapForEach) which sends an array (arr1) and a function which points out the code to manipulate on what to do for each item

```
function mapForEach(arr, fn) {

    var newArr = [];
    for (var i=0; i < arr.length; i++) {
        newArr.push(
            fn(arr[i])
        )
    };

    return newArr;
}

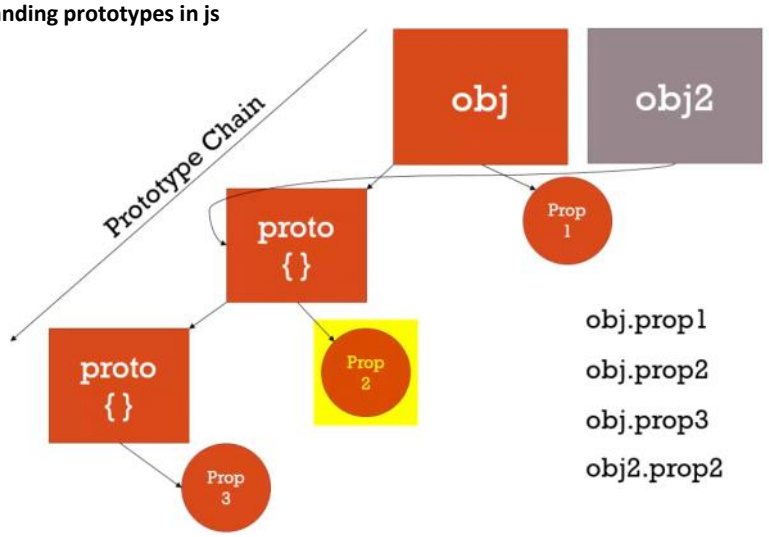
var arr1 = [1,2,3];
console.log(arr1);

var arr2 = mapForEach(arr1, function(item) {
   return item * 2;
});
```

- So the same function can be used for different mathematical operations as in the below example

```
var arr1 = [1,2,3];
console.log(arr1);

var arr2 = mapForEach(arr1, function(item) {
   return item * 2;
});
console.log(arr2);

var arr3 = mapForEach(arr1, function(item) {
   return item > 2;
});
console.log(arr3);
```

  - Output ->

```
[1, 2, 3]                    app.js:14
[2, 4, 6]                    app.js:19
[false, false, true]         app.js:24
```

- Example for passing two arguments when the mapForEach accepts only 1, this can be achieved using bind where we can bind the limiter value permanently to 1 as in the below example

```
var checkPastLimit = function(limiter, item) {
    return item > limiter;
}
var arr4 = mapForEach(arr1, checkPastLimit.bind(this, 1));
console.log(arr4);
```

  - Output ->
  - `[false, true, true]` app.js:33

- Above can be modified further as shown below

```
var checkPastLimitSimplified = function(limiter) {
    return function(limiter, item) {
        return item > limiter;
    }.bind(this, limiter);
};

var arr5 = mapForEach(arr1, checkPastLimitSimplified(2));
console.log(arr5);
```

- We can learn the functional programming by going the following libraries
  - Loadash
  - Underscore js
    - Examples

```
var arr6 = _.map(arr1, function(item) { return item * 3 });
console.log(arr6);

var arr7 = _.filter([2,3,4,5,6,7], function(item) { return item % 2 === 0;
});
console.log(arr7);
```

- Object Oriented JS and Prototypal Inheritance
  - **INHERITANCE:**
    ONE OBJECT GETS ACCESS TO THE PROPERTIES AND METHODS OF ANOTHER OBJECT.

    CLASSICAL INHERITANCE

    Verbose.

    - friend
      protected
      private
      interface
    - Classical Inheritance is the way inheritance is implemented in C#, Java etc..

    PROTOTYPAL INHERITANCE

    Simple.

    - flexible
      extensible
      easy to understand

- **Understanding prototypes in js**
  - 

  - Object has properties and methods.
  - Objects also has hidden properties and methods.

- All objects including functions has a prototype property. It is simply a reference to another object called "proto".
- If proto has an object property called prop2, when we call obj.prop2
  - The property prop2 is looked upon obj first, as it is not available in obj, it looks in the proto object and returns as soon as it finds it
  - Each proto is an object by itself and can contain a prototype
  - It goes by the top-to-bottom approach to find the properties for the obj.
  - The above method is called "Prototype chain".
  - Even obj2 can have the same prototype as obj. Objects in js can share prototypes.
- Example for prototype inheritance
  - ```
    var person = {
        firstname: 'Default',
        lastname: 'Default',
        getFullName: function() {
            return this.firstname + ' ' + this.lastname;
        }
    }

    var john = {
        firstname: 'John',
        lastname: 'Doe'
    }

    // don't do this EVER! for demo purposes only!!!
    john.__proto__ = person;
    console.log(john.getFullName());
    console.log(john.firstname);
    ```
    - Output -> ***John Doe***
  - Even if john object does not have the method getFullName() still it can access the method because of the prototypal inheritance.
  - First the method is checked in john object, if it is not found it goes to the person object to find the method
  - ```
    var jane = {
        firstname: 'Jane'
    }

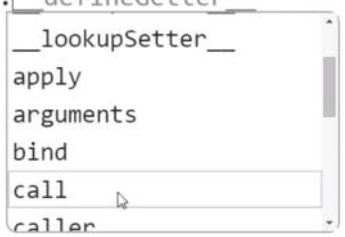
    jane.__proto__ = person;
    console.log(jane.getFullName());

    ```
    - Output -> ***Jane Default***
    - As lastname is not defined in jane object, it goes down till the root object until it finds lastname
    - As person object has lastname, it uses that and print "***Jane Default***"

- **Everything is an object in js**
  - Below is an example
  - ```
    var a = {};
    var b = function() { };
    var c = [];
    ```
  - ```
    > a.__proto__
    < Object {}
    ```
    - a.__proto__ gives us the base object
    - a.__proto__ has toString property which are only available for a base object
    - ```
      > a.__proto__.__defineGetter__
      hasOwnProperty
      isPrototypeOf
      propertyIsEnumerable
      toLocaleString
      toString
      valueOf
      ```
      ```

```
> b.__proto__
function Empty() {}
```

- b.__proto__ gives us an empty function object which is the prototype of every function.
- b.__proto__ has call, bind and apply methods

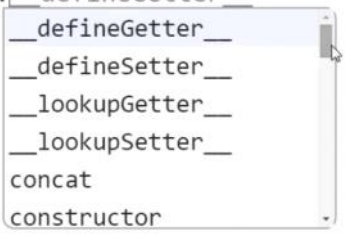
```
> b.__proto__.__defineGetter__
```




```
> b.__proto__.__proto__
Object {}
```

- c.__proto__ gives us an empty array

```
> c.__proto__
[]
> c.__proto__.__defineGetter__
```



- The prototype has methods like push, pop, map etc..
- All arrays has this prototype which the js engine assigns automatically

```
> c.__proto__
[]
> c.__proto__.__proto__
Object {}
```

- The prototype of an prototype of an array is an object

- **Reflection and Extend**
  - 

  - Below is an example of reflection in js

```
var person = {
    firstname: 'Default',
    lastname: 'Default',
    getFullName: function() {
        return this.firstname + ' ' + this.lastname;
    }
}

var john = {
    firstname: 'John',
    lastname: 'Doe'
}

// don't do this EVER! for demo purposes only!!!
john.__proto__ = person;
```

- Now if we want to list down all the properties of the object, we write the below for .. In code

```
for (var prop in john) {
    console.log(prop + ': ' + john[prop]);
}
```

- Output ->

```
firstname: John
lastname: Doe
getFullName: function () {
        return this.firstname + ' ' + this.lastname;
    }
```

- The above code displayed the methods along with the properties. Generally the methods are available only in the prototypes.
- This is because the above code grabs all the methods and properties not only on the object but on the prototype also.
- But to display only the properties on the object, we need to write a condition

```
for (var prop in john) {
    if (john.hasOwnProperty(prop)) {
        console.log(prop + ': ' + john[prop]);
    }
}
```

- Output ->

```
firstname: John                                    app.js:19
lastname: Doe                                      app.js:19
```

- Using Underscore js we can extend the objects as in the below example

```
var jane = {
    address: '111 Main St.',
    getFormalFullName: function() {
        return this.lastname + ', ' + this.firstname;
    }
}

var jim = {
    getFirstName: function() {
        return firstname;
    }
}

_.extend(john, jane, jim);
```

  - The above combines the objects
  - In this case all objects gets combined to john object
  - Output ->

```
app.js:38
▼ Object {firstname: "John", lastname: "Doe", address: "111 Main St.
  ", getFormalFullName: function, getFirstName: function…}
    address: "111 Main St."
    firstname: "John"
  ▶ getFirstName: function () {
  ▶ getFormalFullName: function () {
    lastname: "Doe"
  ▶ __proto__: Object
```

- Below is an internal implementation on how the above code works in underscore js

```
97  // An internal function for creating assigner functions.
98  var createAssigner = function(keysFunc, undefinedOnly) {
99    return function(obj) {
00      var length = arguments.length;
01      if (length < 2 || obj == null) return obj;
02      for (var index = 1; index < length; index++) {
03        var source = arguments[index],
04            keys = keysFunc(source),
05            l = keys.length;
06        for (var i = 0; i < l; i++) {
07          var key = keys[i];
08          if (!undefinedOnly || obj[key] === void 0) obj[key] =
source[key];
09        }
10      }
11      return obj;
12    };
13  };
```

- **Building Objects**
  - **Function Constructors and keyword "new"**
  - 


```
1 function Person() {
2
3     this.firstname = 'John';
4     this.lastname = 'Doe';
5
6 }
7
8 var john = new Person();
9 console.log(john);
```

- Output ->

```
Person {firstname: "John", lastname: "Doe"}                app.js:9
```

- New keyword is an operator and is just another way of creating an object in js.

Javascript Operator Precedence

Precedence	Operator type	Associativity	Individual operators
19	Grouping	n/a	(…)
18	Member Access	left-to-right	… . …
	Computed Member Access	left-to-right	… […]
	new (with argument list)	n/a	new … (…)
17	Function Call	left-to-right	… (…)
	new (without argument list)	right-to-left	new …
16	Postfix Increment	n/a	… ++
	Postfix Decrement	n/a	… --

- As soon as the js engine recognized new
  - It creates an empty object
  - Then it invokes/calls the function
  - When the function is called, we know that the global execution context is created with a variable called "***this***"
  - In this case, when it finds the variable with new keyword, it changes to what the "**this**" variable points to.
  - The new variable created will point out to the new empty object.
  - And the properties of the person object and copied to the newly created object.
  - The new keyword returns the js object created and assigned properties.
  -
```
function Person() {

    console.log(this);
    this.firstname = 'John';
    this.lastname = 'Doe';
    console.log('This function is invoked.');

}

var john = new Person();
console.log(john);
```
    - Output ->
    -
```
Person {}                                                    app.js:3
This function is invoked.                                    app.js:6
Person {firstname: "John", lastname: "Doe"}                  app.js:11
```
  - But if the function has an explicit return statement is returns that instead of the object. As the js engine's way of returning the object is interfered.
    -
```
function Person() {

    console.log(this);
    this.firstname = 'John';
    this.lastname = 'Doe';
    console.log('This function is invoked.');

    return { greeting: 'i got in the way' };

}

var john = new Person();
console.log(john);
```
    - Output ->
    -
```
Person {}                                                    app.js:3
This function is invoked.                                    app.js:6
Object {greeting: "i got in the way"}                        app.js:13
```
  - The function which is specifically used to construct an object is called "**Function Constructor**"

```
function Person(firstname, lastname) {

    console.log(this);
    this.firstname = firstname;
    this.lastname = lastname;
    console.log('This function is invoked.');

}

var john = new Person('John', 'Doe');
console.log(john);

var jane = new Person('Jane', 'Doe');
console.log(jane);
```

- Output ->

```
Person {}                                                    app.js:3
This function is invoked.                                    app.js:6
Person {firstname: "John", lastname: "Doe"}                  app.js:11
Person {}                                                    app.js:3
This function is invoked.                                    app.js:6
Person {firstname: "Jane", lastname: "Doe"}                  app.js:14
```

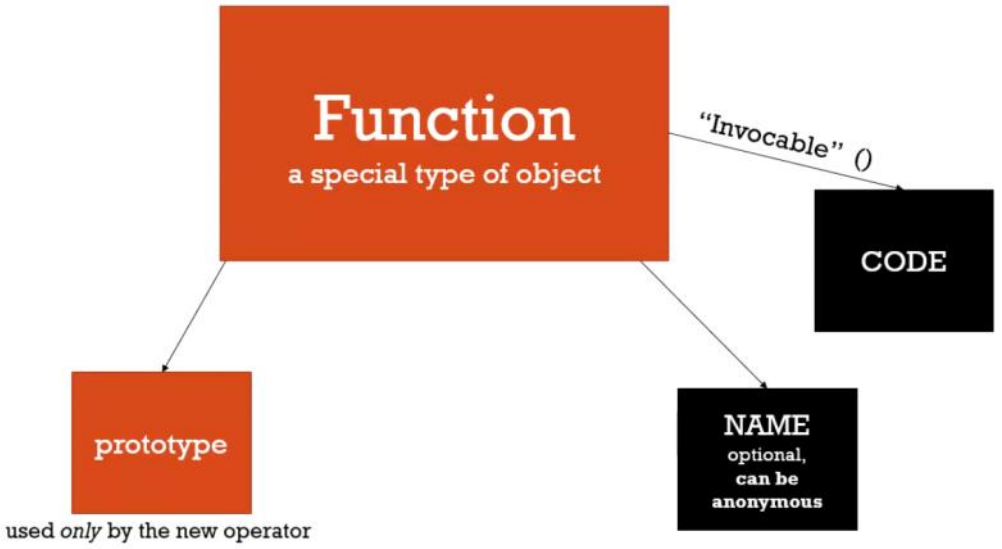
- **Function Constructors and .prototype**
  - When we use a function constructor the prototype will already be set
    - Example

```
function Person(firstname, lastname) {

    console.log(this);
    this.firstname = firstname;
    this.lastname = lastname;
    console.log('This function is invoked.');

}

var john = new Person('John', 'Doe');
console.log(john);

var jane = new Person('Jane', 'Doe');
console.log(jane);
```

- Output ->

```
Person {}                                                    app.js:3
This function is invoked.                                    app.js:6
Person {firstname: "John", lastname: "Doe"}                  app.js:11
Person {}                                                    app.js:3
This function is invoked.                                    app.js:6
Person {firstname: "Jane", lastname: "Doe"}                  app.js:14
john.__proto__
Person {}
```

  - Every function has the below
    - The prototype is always available in the function object, but is used by only the new operator

- 

- __proto__ is the prototype of any object's created in js and not the function's prototype.
-
```
function Person(firstname, lastname) {

    console.log(this);
    this.firstname = firstname;
    this.lastname = lastname;
    console.log('This function is invoked.');

}

Person.prototype.getFullName = function() {
    return this.firstname + ' ' + this.lastname;
}

var john = new Person('John', 'Doe');
console.log(john);

var jane = new Person('Jane', 'Doe');
console.log(jane);
```
- So, the above code works as explained below
- Above is a function constructor implemented and a method is added to its prototype using Person.prototype
- When john object is created using a new operator with the function constructor, an empty object will be created with the properties and methods of Person copied to the newly created empty object.
- So when the prototype is created, john points out to Person.prototype and its prototype __proto__
- Same happens with Jane object also.
  - Output ->
  -
```
▶ Person {getFullName: function}                                    app.js:3
This function is invoked.                                           app.js:6
▶ Person {firstname: "John", lastname: "Doe", getFullName: function}  app.js:15
▶ Person {getFullName: function}                                    app.js:3
This function is invoked.                                           app.js:6
▶ Person {firstname: "Jane", lastname: "Doe", getFullName: function}  app.js:18
```
  -
```
> john.getFullName()
< "John Doe"
```
- ***"The reason why methods always sit in the prototype is that the properties created inside an object uses up memory space, but the methods in prototype if they are added to the properties in an object, then every object has to allocate space for the method which consumes a lot of memory space. If 1000 objects are created 1000 methods will be created for each object. But if added in prototype we have the method only once even if there are 1000 objects. So efficient ways of adding methods to a function constructor is to a prototype."***

- **Dangerous Aside - 'new' and Functions**
  -
```
function Person(firstname, lastname) {

    console.log(this);
    this.firstname = firstname;
    this.lastname = lastname;
    console.log('This function is invoked.');

}

Person.prototype.getFullName = function() {
    return this.firstname + ' ' + this.lastname;
}

var john = Person('John', 'Doe');
console.log(john);

var jane = Person('Jane', 'Doe');
console.log(jane);
```
  - If the function constructor is not initiated with a new operator, the function is still executed but returns an undefined object.
  - And if we try to access the methods of the function we get an error as in the below screenshot.
  -
```
undefined                                                    app.js:15
                                                             app.js:3
▶ Window {top: Window, window: Window, location: Location, external:
  Object, chrome: Object…}
This function is invoked.                                    app.js:6
undefined                                                    app.js:18
▶ Uncaught TypeError: Cannot read property                   app.js:24
  'getFormalFullName' of undefined
```
  - Convention is to make the function constructors in capital(Ex. Person) so that we identify it as a function constructor and add a new operator to it.

- **Built-in Function Constructors**
  - Function constructors are used to create objects which contain primitives not the primitives.
  - Examples
  - Number
    -
```
> var a = new Number(3)
undefined
> a
Number {[[PrimitiveValue]]: 3}
```
    - And the built-in Number function constructor has methods in its prototype
    -
```
Elements  Ne  propertyIsEnumerable  Profiles  »
              toExponential
<top frame>   toFixed
> var a = new Numbe  toLocaleString
undefined             toPrecision
> a                   toString
Number {[[Primiti     valueOf
> Number.prototype.__defineGetter__
```

```
> var a = new Number(3)
undefined
> a
Number {[[PrimitiveValue]]: 3}
> a.toFixed(2)
"3.00"
>
```

- Example of primitive inside an object
  - a.toFixed(2)

- String

```
> var a = new String("John")
undefined
> String.prototype.indexOf('o')
-1
> String.prototype.indexOf("Jo")
-1
```

- Example of primitive inside an object
  - "John".length
  - The above is equivalent to new String("John").length

```
String.prototype.isLengthGreaterThan = function(limit) {
    return this.length > limit;
}

console.log("John".isLengthGreaterThan(3));
```

  - Output -> true
  - In the above code primitive string ("John") is converted to John object which returns an object and in its prototype the isLengthGreaterThan method is added

- **Dangerous Aside - Built-in function constructors**

```
> var a = 3
undefined
> var b = new Number(3)
undefined
> a == b
true
> a === b
false
```

- When type checked using "===" it returns false because a is an primitive and b is an object which contains the primitive.

- **Dangerous Aside - Arrays and for..in**

```
var arr = ['John', 'Jane', 'Jim'];

for (var prop in arr) {
    console.log(prop + ': ' + arr[prop]);
}
```

- Output ->

```
0: John                                    app.js:4
1: Jane                                    app.js:4
2: Jim                                     app.js:4
```

```
Array.prototype.myCustomFeature = 'cool!';

```

-
```
Array.prototype.myCustomFeature = 'cool!';

var arr = ['John', 'Jane', 'Jim'];

for (var prop in arr) {
    console.log(prop + ': ' + arr[prop]);
}
```
  - Output->
    ```
    0: John                                   app.js:6
    1: Jane                                   app.js:6
    2: Jim                                    app.js:6
    myCustomFeature: cool!                    app.js:6
    ```
  - So instead of using for..in we have to use the standard for loop

- **Object.create and Pure Prototypal Inheritance**
  -
```
var person = {
    firstname: 'Default',
    lastname: 'Default',
    greet: function() {
        return 'Hi ' + this.firstname;
    }
}

var john = Object.create(person);
console.log(john);
```
    - Output->
      ```
                                                                app.js:10
      ▼ Object {firstname: "Default", lastname: "Default", greet: function}
        ▼ __proto__: Object
          firstname: "Default"
          ▶ greet: function () {
          lastname: "Default"
          ▶ __proto__: Object
      ```
    -
      ```
      > john.greet()
      < "Hi Default"
      ```
  -
```
var person = {
    firstname: 'Default',
    lastname: 'Default',
    greet: function() {
        return 'Hi ' + this.firstname;
    }
}

var john = Object.create(person);
john.firstname = 'John';
john.lastname = 'Doe';
console.log(john);
```
    - Output->
      ```
                                                                app.js:12
      ▼ Object {firstname: "John", lastname: "Doe", greet: function}
          firstname: "John"
          lastname: "Doe"
        ▼ __proto__: Object
          firstname: "Default"
          ▶ greet: function () {
      ```

```
app.js:12
▼ Object {firstname: "John", lastname: "Doe", greet: function}
    firstname: "John"
    lastname: "Doe"
  ▼ __proto__: Object
      firstname: "Default"
    ▶ greet: function () {
      lastname: "Default"
    ▶ __proto__: Object

> john.greet()
< "Hi John"
```

- The above is Pure prototypal inheritance.
- Here we just make objects, create new objects from them pointing to other objects as their prototype.
- If we define a new object, we create a new object and we simply override, hide properties and methods on those created objects by setting the values of those properties and methods on the new objects themselves.
- Here we can mutate and change the prototype on the fly.
- 

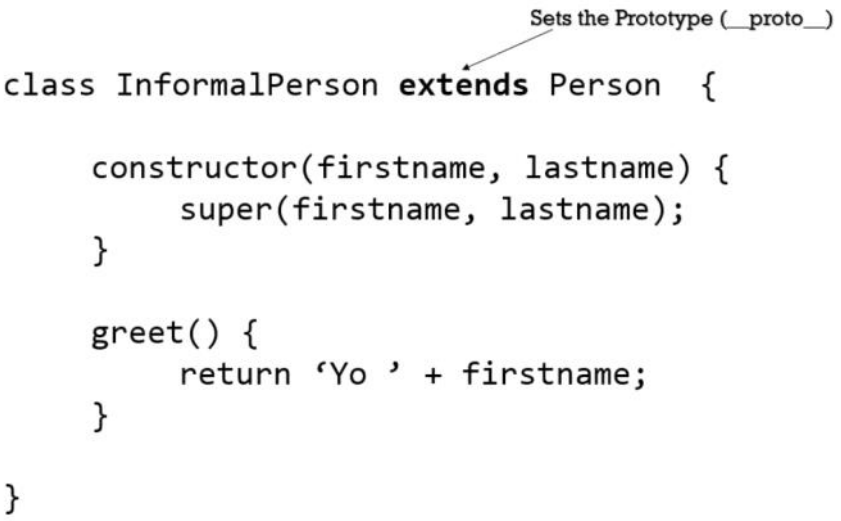
- If object.create is not available in the older browsers which has older js engines, we use something called as polyfills.
- We can check if the js engine has the feature and if it doesn't have we use polyfills.
- ```
  if (!Object.create) {
    Object.create = function (o) {
      if (arguments.length > 1) {
        throw new Error('Object.create implementation'
        + ' only accepts the first parameter.');
      }
      function F() {}
      F.prototype = o;
      return new F();
    };
  }
  ```
- We have an empty constructor F()
- The function F prototype is assigned the passed object.
- The Object.create returns the function constructor with the new operator.

- **ES6 and Classes**
  - Example of class declaration in ES6 with properties and methods
  ```

```
class Person {

    constructor(firstname, lastname) {
        this.firstname = firstname;
        this.lastname = lastname;
    }

    greet() {
        return 'Hi ' + firstname;
    }

}

var john = new Person('John', 'Doe');
```

- Class is also an object in js.



- Odds and Ends (typeof and instanceof)

```
var a = 3;
console.log(typeof a);

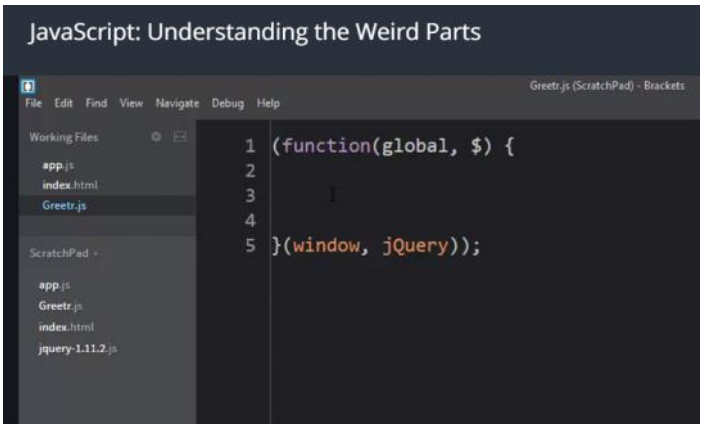
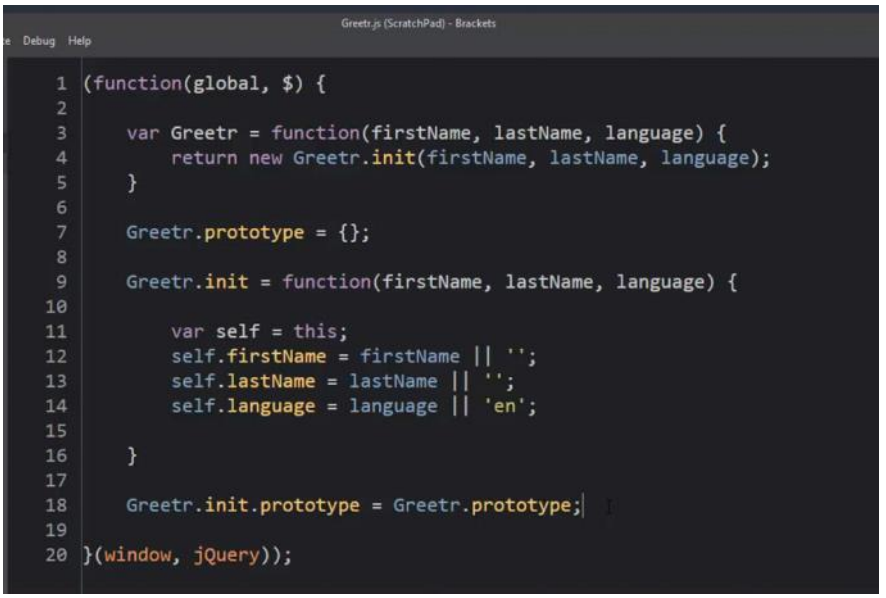
var b = "Hello";
console.log(typeof b);

var c = {};
console.log(typeof c);

var d = [];
console.log(typeof d); // weird!
console.log(Object.prototype.toString.call(d)); // better!

function Person(name) {
    this.name = name;
}

var e = new Person('Jane');
console.log(typeof e);
console.log(e instanceof Person);

console.log(typeof undefined); // makes sense
```

  - Output ->

```
number                                  app.js:2
string                                  app.js:5
object                                  app.js:8
object                                  app.js:11
[object Array]                          app.js:12
object                                  app.js:19
true                                    app.js:20
```

```
23 console.log(typeof null); // a bug since, like, forever...
```

- Output ->
  - `object` app.js:23

```
25 var z = function() { };
26 console.log(typeof z);
```

- Output ->
  - `function` app.js:26

- Uses of using "use strict"

```
1  function logNewPerson() {
2      "use strict";
3
4      var person2;
5      persom2 = {};
6      console.log(persom2);
7  }
8
9  var person;
10 persom = {};
11 console.log(persom);
12 logNewPerson();
```

- Output ->

```
Object {}                                   app.js:11
▶ Uncaught ReferenceError: persom2 is not defined   app.js:5
>
```

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

- Creating our own library called 'Greetr'

- 

- 

- To use the Greetr as below without using the new keyword we need to write as above
  - ```
    var g = G$(firstname, lastname, language)
    ```
  - So Greetr creates the new constructor of init
  - And we also make sure that the Greetr prototype points out to the Greetr init prototype.
    ```

Design Patterns

Wednesday, November 6, 2019 9:15 PM

Design Pattern

The academic definition is a general, reusable solution to a commonly-occurring problem within a given context in software design. In simple words, it's a way that has been defined as a proper approach to resolve common problems in code.

Types

- Creational → Create new things
- Structural → Structure your code
- Behavioral → Use for behaviors in code

- **Creational Design Patterns**
  - As the name suggests, these patterns are for handling object creational mechanisms. A creational design pattern basically solves a problem by controlling the creation process of an object.
  - Examples: **Constructor Pattern**, **Factory Pattern**, **Prototype Pattern**, and **Singleton Pattern**.
  -

Creational	Based on the concept of creating an object.
Class	
Factory Method	This makes an instance of several derived classes based on interfaced data or events.
Object	
Abstract Factory	Creates an instance of several families of classes without detailing concrete classes.
Builder	Separates object construction from its representation, always creates the same type of object.
Prototype	A fully initialized instance used for copying or cloning.
Singleton	A class with only a single instance with global access points.

- **Structural Design Patterns**
  - These patterns are concerned with class and object composition. They help structure or restructure one or more parts without affecting the entire system. In other words, they help obtain new functionalities without tampering with the existing ones.
  - Examples: **Adapter Pattern**, **Composite Pattern**, **Decorator Pattern**, **Façade Pattern**, **Flyweight Pattern**, and **Proxy Pattern**.

Structural	Based on the idea of building blocks of objects.
Class	
Adapter	Match interfaces of different classes therefore classes can work together despite incompatible interfaces.
Object	
Adapter	Match interfaces of different classes therefore classes can work together despite incompatible interfaces.
Bridge	Separates an object's interface from its implementation so the two can vary independently.
Composite	A structure of simple and composite objects which makes the total object more than just the sum of its parts.
Decorator	Dynamically add alternate processing to objects.
Facade	A single class that hides the complexity of an entire subsystem.
Flyweight	A fine-grained instance used for efficient sharing of information that is contained elsewhere.
Proxy	A place holder object representing the true object.

- **Behavioral Design Patterns**
  - These patterns are concerned with improving communication between dissimilar objects.
  - Examples: **Chain of Responsibility Pattern**, **Command Pattern**, **Iterator Pattern**, **Mediator Pattern**, **Observer Pattern**, **State Pattern**, **Strategy Pattern**, and **Template Pattern**

Behavioral	Based on the way objects play and work together.
Class	
Interpreter	A way to include language elements in an application to match the grammar of the intended language.
Template Method	Creates the shell of an algorithm in a method, then defer the exact steps to a subclass.
Object	
Chain of Responsibility	A way of passing a request between a chain of objects to find the object that can handle the request.
Command	Encapsulate a command request as an object to enable, logging and/or queuing of requests, and provides error-handling for unhandled requests.
Iterator	Sequentially access the elements of a collection without knowing the inner workings of the collection.
Mediator	Defines simplified communication between classes to prevent a group of classes from referring explicitly to each other.
Memento	Capture an object's internal state to be able to restore it later.
Observer	A way of notifying change to a number of classes to ensure consistency between the classes.
State	Alter an object's behavior when its state changes.
Strategy	Encapsulates an algorithm inside a class separating the selection from the implementation.
Visitor	Adds a new operation to a class without changing the class.

Functions as first-class citizens

- It means that when functions can be treated like a variable, meaning they can be passed as arguments to other functions as well or can be assigned as a value to a variable or even return in a function, it means the function is a first-class citizen and JavaScript does first-class citizen functions.

```
const calc = () => {
    return 4 * 3;
}

let aNumber = calc();
```

- **aNumber** is a first-class citizen function.

Callback function

- In its simplest terms, a callback function is a function that is called inside of another function. In other words, whenever you pass a function in the arguments and run it inside this function you're doing the callback pattern.

```
const calc = () => {
    return 4 * 3;
}

const printCalc = (callback) => {
    console.log(callback());
}

printCalc(calc);
```

<u>Creational Design Patterns</u>

Prototype/Class Design Pattern

- This pattern is an **object-based creational design pattern**. In this, we use a sort of a "skeleton" of an existing object to create or instantiate new objects.
- This pattern is specifically important and beneficial to JavaScript because it utilizes prototypal inheritance instead of a classic object-oriented inheritance. Hence, it plays to JavaScript's strength and has native support.
- The prototype, or class pattern, allows us to define a blueprint for a specific type of item and then reuse it by creating a new object from that class.

```
class Car {
    constructor(doors, engine, color) {
        this.doors = doors;
        this.engine = engine;
        this.color = color;
    }
}

const civic = new Car(4, 'V6', 'grey');

console.log(civic);
```

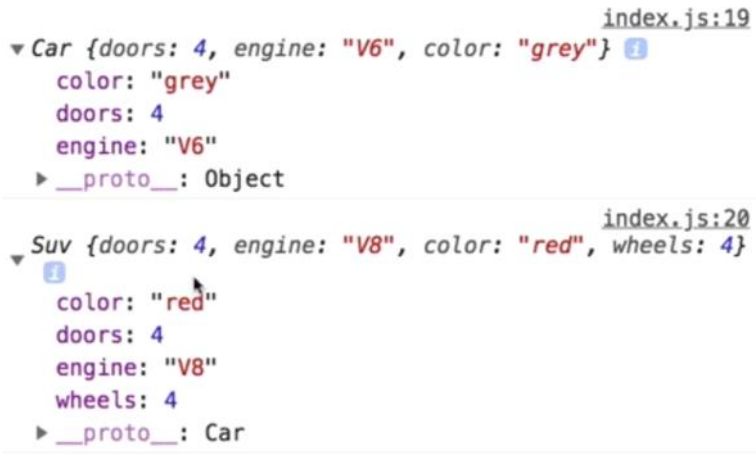
```
index.js:11
▼Car {doors: 4, engine: "V6", color: "grey"}
   color: "grey"
   doors: 4
   engine: "V6"
  ▼__proto__:
    ▶constructor: class Car
    ▶__proto__: Object
```

```
// using Object.create as was recommended by ES5 standard
const car = {
  noOfWheels: 4,
  start() {
    return 'started';
  },
  stop() {
    return 'stopped';
  },
};

// Object.create(proto[, propertiesObject])

const myCar = Object.create(car, { owner: { value: 'John' } });

console.log(myCar.__proto__ === car); // true
```

Constructor Design Pattern

- This is a **class-based creational design pattern**. Constructors are special functions that can be used to instantiate new objects with methods and properties defined by that function.
- Similar to the class prototype pattern, the constructor pattern is one step further from the class pattern, where we leverage a class created to initiate a new extended class from it.
- It is not one of the classic design patterns. In fact, it is more of a basic language construct than a pattern in most object-oriented languages. But in JavaScript, objects can be created on the fly without any constructor functions or “class” definition. Therefore, I think it is important to lay down the foundation for other patterns to come with this simple one.
- Constructor pattern is one of the most commonly used patterns in JavaScript for creating new objects of a given kind.

```
class Car {
    constructor(doors, engine, color) {
        this.doors = doors;
        this.engine = engine;
        this.color = color;
    }
}

class Suv extends Car {
    constructor(doors, engine, color) {
        super(doors, engine, color);
        this.wheels = 4;
    }
}

const civic = new Car(4, 'V6', 'grey');
const cx5 = new Suv(4, "V8", 'red');

console.log(civic);
console.log(cx5);
```



-

```
// traditional Function-based syntax
function Hero(name, specialAbility) {
  // setting property values
  this.name = name;
  this.specialAbility = specialAbility;

  // declaring a method on the object
  this.getDetails = function() {
    return this.name + ' can ' + this.specialAbility;
  };
}

// ES6 Class syntax
class Hero {
  constructor(name, specialAbility) {
    // setting property values
    this._name = name;
    this._specialAbility = specialAbility;

    // declaring a method on the object
    this.getDetails = function() {
      return `${this._name} can ${this._specialAbility}`;
    };
  }
}

// creating new instances of Hero
const IronMan = new Hero('Iron Man', 'fly');

console.log(IronMan.getDetails()); // Iron Man can fly
```

-

Singleton Design Pattern

- Singleton is a **special creational design pattern** in which only one instance of a class can exist. It works like this — if no instance of the singleton class exists then a new instance is created and returned, but if an instance already exists, then the reference to the existing instance is returned.
- Preventing our class from creating more than one instance of the blueprint we've defined.

```
let instance = null;

class Car {
    constructor(doors, engine, color) {
        if (!instance) {
            this.doors = doors;
            this.engine = engine;
            this.color = color;
            instance = this;
        } else {
            return instance;
        }

    }
}

const civic = new Car(4, 'V6', 'grey');
const honda = new Car(2, 'V4', 'red');

console.log(civic);
console.log(honda);
```

```
index.js:27
▸ Car {doors: 4, engine: "V6", color: "grey"}
index.js:28
▸ Car {doors: 4, engine: "V6", color: "grey"}
```

- A perfect real-life example would be that of mongoose (the famous Node.js ODM library for MongoDB). It utilizes the singleton pattern.
- In the below example, we have a Database class that is a singleton.
- First, we create an object mongo by using the new operator to invoke the Database class constructor. This time an object is instantiated because none already exists. The second time, when we create the mysql object, no new object is instantiated but instead, the reference to the object that was instantiated earlier, i.e. the mongo object, is returned.

```
class Database {
  constructor(data) {
    if (Database.exists) {
      return Database.instance;
    }
    this._data = data;
    Database.instance = this;
    Database.exists = true;
    return this;
  }

  getData() {
    return this._data;
  }

  setData(data) {
    this._data = data;
  }
}

// usage
const mongo = new Database('mongo');
console.log(mongo.getData()); // mongo

const mysql = new Database('mysql');
console.log(mysql.getData()); // mongo
```

Factory Pattern

- Factory pattern is another **class-based creational pattern**. In this, we provide a generic interface that delegates the responsibility of object instantiation to its subclasses.
- This pattern is frequently used when we need to manage or manipulate collections of objects that are different yet have many similar characteristics.
- In this example, we create a factory class named BallFactory that has a method that takes in parameters, and, depending on the parameters, it delegates the object instantiation responsibility to the respective class. If the type parameter is "football" or "soccer" object instantiation is handled by Football class, but if it is "basketball" object instantiation is handled by Basketball class.

```
class BallFactory {
  constructor() {
    this.createBall = function(type) {
      let ball;
      if (type === 'football' || type === 'soccer') ball = new Football();
      else if (type === 'basketball') ball = new Basketball();
      ball.roll = function() {
        return `The ${this._type} is rolling.`;
      };

      return ball;
    };
  }
}

class Football {
  constructor() {
    this._type = 'football';
    this.kick = function() {
      return 'You kicked the football.';
    };
  }
}

class Basketball {
  constructor() {
    this._type = 'basketball';
    this.bounce = function() {
      return 'You bounced the basketball.';
    };
  }
}
```

-

```

// creating objects
const factory = new BallFactory();

const myFootball = factory.createBall('football');
const myBasketball = factory.createBall('basketball');

console.log(myFootball.roll()); // The football is rolling.
console.log(myBasketball.roll()); // The basketball is rolling.
console.log(myFootball.kick()); // You kicked the football.
console.log(myBasketball.bounce()); // You bounced the basketball.
```

-

```
class Car {
    constructor(doors, engine, color) {
        this.doors = doors;
        this.engine = engine;
        this.color = color;
    }
}

class carFactory {
    createCar(type) {
        switch(type) {
            case 'civic':
                return new Car(4, 'V6', 'grey')
            case 'honda':
                return new Car(2, 'V4', 'red')
        }
    }
}

const factory = new carFactory();
const honda = factory.createCar('honda');

console.log(honda);
```

```
                                                    index.js:23
▼ Car {doors: 2, engine: "V4", color: "red"}
    color: "red"
    doors: 2
    engine: "V4"
  ▶ __proto__: Object
```

Abstract Factory

- The abstract factory pattern where you abstract the factories and are able to create multiple factories, classes, etcetera. So in our car example, this would be a car company overseeing multiple factories.

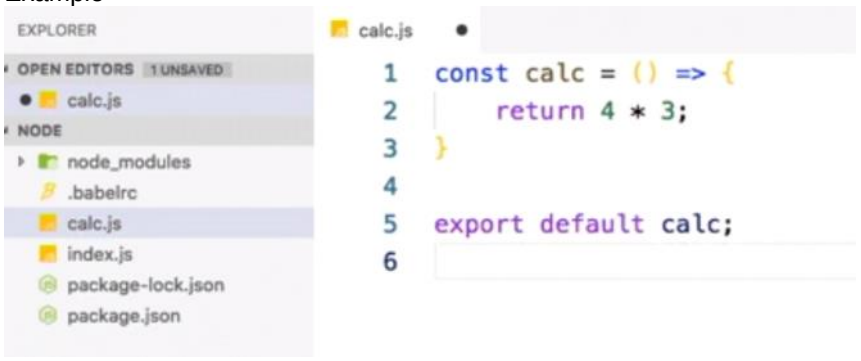
```
class Car {
    constructor(doors, engine, color) {
        this.doors = doors;
        this.engine = engine;
        this.color = color;
    }
}

class carFactory {
    createCar(type) {
        switch(type) {
            case 'civic':
                return new Car(4, 'V6', 'grey')
            case 'honda':
                return new Car(2, 'V4', 'red')
        }
    }
}

class Suv {
    constructor(doors, engine, color) {
        this.doors = doors;
        this.engine = engine;
        this.color = color;
    }
}

class suvFactory {
    createSuv(type) {
        switch(type) {
            case 'cx5':
                return new Car(4, 'V6', 'grey')
            case 'sante fe':
                return new Car(2, 'V4', 'red')
        }
    }
}

const carFactory = new carFactory();
const suvFactory = new suvFactory();

const autoManufacturer = (type, model) => {
    switch(type) {
        case 'car':
            return carFactory.createCar(model);
        case 'suv':
            return suvFactory.createSuv(model);
    }
}

const cx5 = autoManufacturer('suv', 'cx5');

console.log(cx5);
```

```
index.js:53
Car {doors: 4, engine: "V6", color: "grey"}
    color: "grey"
    doors: 4
    engine: "V6"
  ▶ __proto__: Object
```

Structural Design Patterns

Module Pattern

- Whenever you encapsulate a block of code into a singular function or pure function as it is sometimes referred to, you're creating a module.
- The idea behind using module is to organize your code in pure functions so if you have code to debug, it is much easier to find where the error is.
- We often use modules too with the keyword, import or export, so when we compile our code, we only use the code we need.
- Example



Mixins Patterns

- Mixins are a great way to mix functions and instances of a class after they have been created. In other words, you could use Mixins to add interesting functions to the car class we created earlier.

```
class Car {
    constructor(doors, engine, color) {
        this.doors = doors;
        this.engine = engine;
        this.color = color;
    }
}

class CarFactory {
    createCar(type) {
        switch(type) {
            case 'civic':
                return new Car(4, 'V6', 'grey')
            case 'honda':
                return new Car(2, 'V4', 'red')
        }
    }
}

let carMixin = {
    revEngine() {
        console.log(`The ${this.engine} is doing Vroom Vroom!`)
    }
}

const carFactory = new CarFactory();
```

```
let carMixin = {
    revEngine() {
        console.log(`The ${this.engine} is doing Vroom Vroom!`);
    }
}

const carFactory = new CarFactory();

const autoManufacturer = (type, model) => {
    switch(type) {
        case 'car':
            return carFactory.createCar(model);
        case 'suv':
            return suvFactory.createSuv(model);
    }
}

Object.assign(Car.prototype, carMixin);

const honda = autoManufacturer('car', 'honda');

honda.revEngine();
```

```
The V4 is doing Vroom Vroom!                    index.js:22
>
```

Adapter Pattern

- This is a **structural design pattern** where the interface of one class is translated into another. This pattern lets classes

work together that could not otherwise because of incompatible interfaces.

- This pattern is often used to create wrappers for new refactored APIs so that other existing old APIs can still work with them. This is usually done when new implementations or code refactoring (done for reasons like performance gains) result in a different public API, while the other parts of the system are still using the old API and need to be adapted to work together.
- In the below example,
  - We have an old API, i.e. *OldCalculator* class, and a new API, i.e. *NewCalculator* class.
  - The *OldCalculator* class provides an operation method for both addition and subtraction, while the *NewCalculator* provides separate methods for addition and subtraction.
  - The Adapter class *CalcAdapter* wraps the *NewCalculator* to add the operation method to the public-facing API while using its own addition and subtraction implementation under the hood.
  -

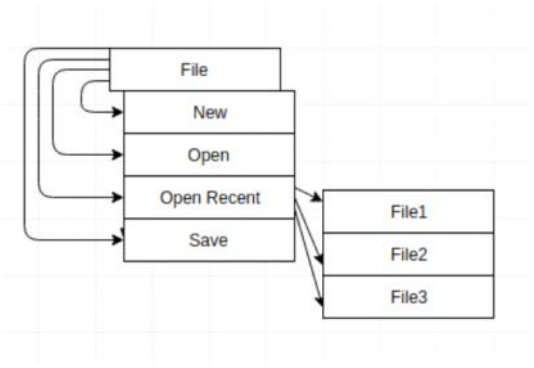
```
// old interface
class OldCalculator {
  constructor() {
    this.operations = function(term1, term2, operation) {
      switch (operation) {
        case 'add':
          return term1 + term2;
        case 'sub':
          return term1 - term2;
        default:
          return NaN;
      }
    };
  }
}

// new interface
class NewCalculator {
  constructor() {
    this.add = function(term1, term2) {
      return term1 + term2;
    };
    this.sub = function(term1, term2) {
      return term1 - term2;
    };
  }
}
```

```
// Adapter Class
class CalcAdapter {
  constructor() {
    const newCalc = new NewCalculator();

    this.operations = function(term1, term2, operation) {
      switch (operation) {
        case 'add':
          // using the new implementation under the hood
          return newCalc.add(term1, term2);
        case 'sub':
          return newCalc.sub(term1, term2);
        default:
          return NaN;
      }
    };
  }
}

// usage
const oldCalc = new OldCalculator();
console.log(oldCalc.operations(10, 5, 'add')); // 15

const newCalc = new NewCalculator();
console.log(newCalc.add(10, 5)); // 15

const adaptedCalc = new CalcAdapter();
console.log(adaptedCalc.operations(10, 5, 'add')); // 15;
```

Composite Pattern

- This is a **structural design pattern** that composes objects into tree-like structures to represent whole-part hierarchies.

- In this pattern, each node in the tree-like structure can be either an individual object or a composed collection of objects. Regardless, each node is treated uniformly.
-



A Multi-level Menu Structure

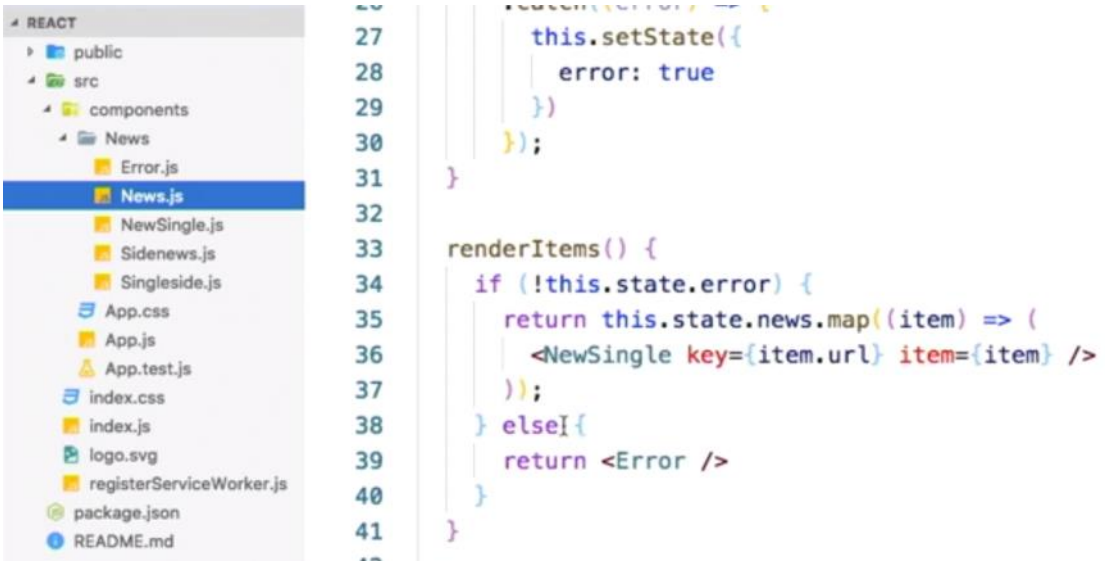
- Above is an example of a multi-level menu.
- Each node can be a distinct option, or it can be a menu itself, which has multiple options as its child.
- A node component with children is a composite component, while a node component without any child is a leaf component.
- In the below example,
  - We create a base class of Component that implements the common functionalities needed and abstracts the other methods needed.
  - The base class also has a static method that utilizes recursion to traverse a composite tree structure made with its subclasses.
  - Then we create two subclasses extending the base class — Leaf that does not have any children and Composite that can have children—and hence have methods handling adding, searching, and removing child functionalities. The two subclasses are used to create a composite structure—a tree, in this case.
  -

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

  getNodeName() {
    return this._name;
  }

  // abstract methods that need to be overridden
  getType() {}

  addChild(component) {}

  removeChildByName(componentName) {}

  removeChildByIndex(index) {}

  getChildByName(componentName) {}

  getChildByIndex(index) {}

  noOfChildren() {}

```

```
  static logTreeStructure(root) {
    let treeStructure = '';
    function traverse(node, indent = 0) {
      treeStructure += `${'--'.repeat(indent)}${node.getNodeName()}\n`;
      indent++;
      for (let i = 0, length = node.noOfChildren(); i < length; i++) {
        traverse(node.getChildByIndex(i), indent);
      }
    }

    traverse(root);
    return treeStructure;
  }
}

class Leaf extends Component {
  constructor(name) {
    super(name);
    this._type = 'Leaf Node';
  }

  getType() {
    return this._type;
  }

  noOfChildren() {
    return 0;
  }
}

class Composite extends Component {
  constructor(name) {
    super(name);
    this._type = 'Composite Node';
    this._children = [];
  }

  getType() {
    return this._type;
  }

  addChild(component) {
    this._children = [...this._children, component];
  }

  removeChildByName(componentName) {
    this._children = [...this._children].filter(component => component.getNodeName() !== compon
  }

  removeChildByIndex(index) {
    this._children = [...this._children.slice(0, index), ...this._children.slice(index + 1)];
  }

  getChildByName(componentName) {
    return this._children.find(component => component.name === componentName);
  }
```

```
  getChildByIndex(index) {
    return this._children[index];
  }

  noOfChildren() {
    return this._children.length;
  }
}

// usage
const tree = new Composite('root');
tree.addChild(new Leaf('left'));
const right = new Composite('right');
tree.addChild(right);
right.addChild(new Leaf('right-left'));
const rightMid = new Composite('right-middle');
right.addChild(rightMid);
right.addChild(new Leaf('right-right'));
rightMid.addChild(new Leaf('left-end'));
rightMid.addChild(new Leaf('right-end'));

// log
console.log(Component.logTreeStructure(tree));
/*
root
--left
--right
----right-left
----right-middle
------left-end
------right-end
----right-right
*/
```

Façade Pattern

- Facade is basically the pattern of hiding away complexity by creating a facade for the complex code.
- This is a **structural design pattern** that is widely used in the JavaScript libraries. It is used to provide a unified and simpler, public-facing interface for ease of use that shields away from the complexities of its consisting subsystems or subclasses.
- If you're a react developer or building components, you have been using facades every day.
- When you are building a component in any framework, you code the complexity of this component into a module or file and then leverage a simple line to render this component into your code.
- In the below example <NewsSingle /> is a façade
  - 

- In the below example, we create a public facing API with the class *ComplaintRegistry*. It exposes only one method to be used by the client, i.e. *registerComplaint*. It internally handles instantiating required objects of either *ProductComplaint* or *ServiceComplaint* based on the type argument. It also handles all the other complex functionalities like generating a unique ID, storing the complaint in memory, etc. But, all these complexities are hidden away

using the façade pattern.

```
let currentId = 0;

class ComplaintRegistry {
  registerComplaint(customer, type, details) {
    const id = ComplaintRegistry._uniqueIdGenerator();
    let registry;
    if (type === 'service') {
      registry = new ServiceComplaints();
    } else {
      registry = new ProductComplaints();
    }
    return registry.addComplaint({ id, customer, details });
  }

  static _uniqueIdGenerator() {
    return ++currentId;
  }
}

class Complaints {
  constructor() {
    this.complaints = [];
  }

  addComplaint(complaint) {
    this.complaints.push(complaint);
    return this.replyMessage(complaint);
  }

  getComplaint(id) {
    return this.complaints.find(complaint => complaint.id === id);
  }

  replyMessage(complaint) {}
}
```

```
class ProductComplaints extends Complaints {
  constructor() {
    super();
    if (ProductComplaints.exists) {
      return ProductComplaints.instance;
    }
    ProductComplaints.instance = this;
    ProductComplaints.exists = true;
    return this;
  }

  replyMessage({ id, customer, details }) {
    return `Complaint No. ${id} reported by ${customer} regarding ${details} have been filed wit
  }
}

class ServiceComplaints extends Complaints {
  constructor() {
    super();
    if (ServiceComplaints.exists) {
      return ServiceComplaints.instance;
    }
    ServiceComplaints.instance = this;
    ServiceComplaints.exists = true;
    return this;
  }

  replyMessage({ id, customer, details }) {
    return `Complaint No. ${id} reported by ${customer} regarding ${details} have been filed wit
  }
}

```

```
// usage
const registry = new ComplaintRegistry();

const reportService = registry.registerComplaint('Martha', 'service', 'availability');
// 'Complaint No. 1 reported by Martha regarding availability have been filed with the Service C

const reportProduct = registry.registerComplaint('Jane', 'product', 'faded color');
// 'Complaint No. 2 reported by Jane regarding faded color have been filed with the Products Com
```

Flyweight Pattern

- This is a **structural design pattern** focused on efficient data sharing through fine-grained objects. It is used for efficiency and memory conservation purposes.
- This pattern can be used for any kind of caching purposes. In fact, modern browsers use a variant of a flyweight pattern to prevent loading the same images twice.
- Uses similar pattern to Singleton.
- In this example, we create a fine-grained flyweight class *Icecream* for sharing data regarding ice-cream flavors and a factory class *IcecreamFactory* to create those flyweight objects. For memory conservation, the objects are recycled if the same object is instantiated twice. This is a simple example of flyweight implementation.

```
// flyweight class
class Icecream {
  constructor(flavour, price) {
    this.flavour = flavour;
    this.price = price;
  }
}

// factory for flyweight objects
class IcecreamFactory {
  constructor() {
    this._icecreams = [];
  }

  createIcecream(flavour, price) {
    let icecream = this.getIcecream(flavour);
    if (icecream) {
      return icecream;
    } else {
      const newIcecream = new Icecream(flavour, price);
      this._icecreams.push(newIcecream);
      return newIcecream;
    }
  }

  getIcecream(flavour) {
    return this._icecreams.find(icecream => icecream.flavour === flavour);
  }
}

```

```
// usage
const factory = new IcecreamFactory();

const chocoVanilla = factory.createIcecream('chocolate and vanilla', 15);
const vanillaChoco = factory.createIcecream('chocolate and vanilla', 15);

// reference to the same object
console.log(chocoVanilla === vanillaChoco); // true
```

Decorator Pattern

- This is also a **structural design pattern** that focuses on the ability to add behavior or functionalities to existing classes dynamically. It is another viable alternative to sub-classing.
- The decorator type behavior is very easy to implement in JavaScript because JavaScript allows us to add methods and properties to object dynamically. The simplest approach would be to just add a property to an object, but it will not be efficiently reusable.

```
class Book {
  constructor(title, author, price) {
    this._title = title;
    this._author = author;
    this.price = price;
  }

  getDetails() {
    return `${this._title} by ${this._author}`;
  }
}

// decorator 1
function giftWrap(book) {
  book.isGiftWrapped = true;
  book.unwrap = function() {
    return `Unwrapped ${book.getDetails()}`;
  };

  return book;
}

// decorator 2
function hardbindBook(book) {
  book.isHardbound = true;
  book.price += 5;
  return book;
}
```

-

```
// usage
const alchemist = giftWrap(new Book('The Alchemist', 'Paulo Coelho', 10));

console.log(alchemist.isGiftWrapped); // true
console.log(alchemist.unwrap()); // 'Unwrapped The Alchemist by Paulo Coelho'

const inferno = hardbindBook(new Book('Inferno', 'Dan Brown', 15));

console.log(inferno.isHardbound); // true
console.log(inferno.price); // 20
```

-

- Decorators in TypeScript

```
function f() {
    console.log("f(): evaluated");
    return function (target, propertyKey: string, descriptor: PropertyDescriptor) {
        console.log("f(): called");
    }
}

function g() {
    console.log("g(): evaluated");
    return function (target, propertyKey: string, descriptor: PropertyDescriptor) {
        console.log("g(): called");
    }
}

class C {
    @f()
    @g()
    method() {}
}
```

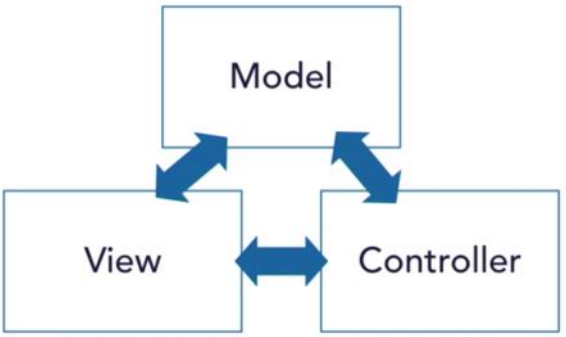
-

- Decorators in Angular

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-heroes',
  template: `
    <h2>Heroes</h2>
    <app-hero-list></app-hero-list>
  `
})
export class HeroesComponent { }
```

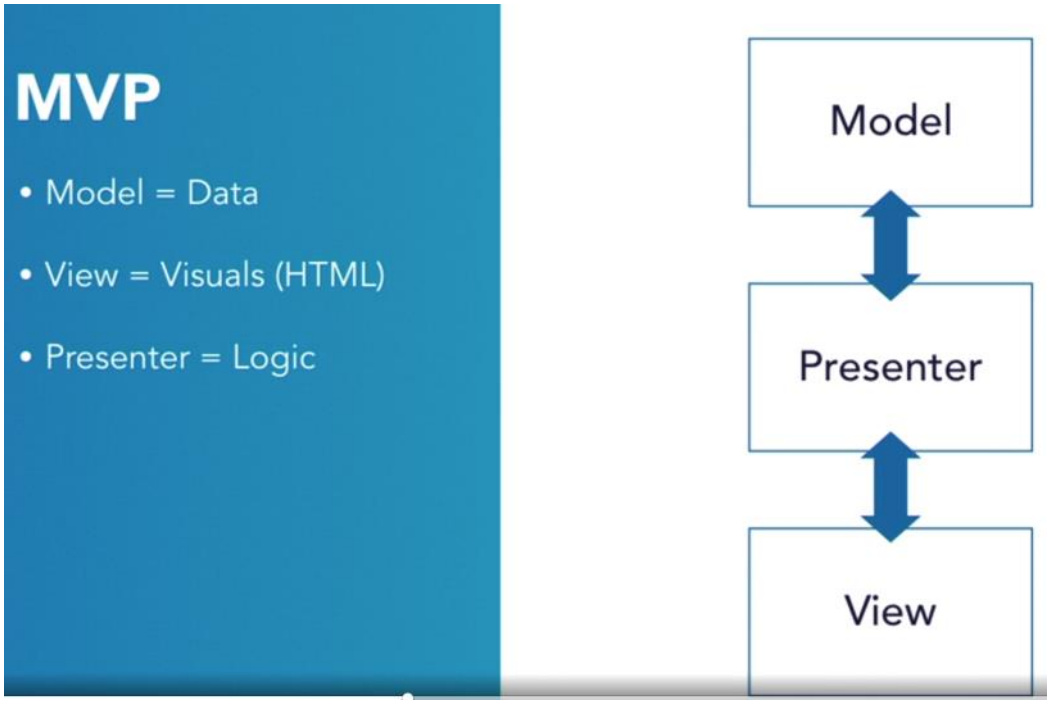
MVC (Model-View-Controller) Pattern

- This pattern basically defines how an application should be split and often reflects how your modules are organized within three simple categories, models, views and controllers.
- The model is where your data resides where you define your schemas and the models for your data.
- The views is where you have your views and in most case, the pure HTML of your application, where the visuals are
- The controllers are where you have your logic of your application, the functions that makes your application run.



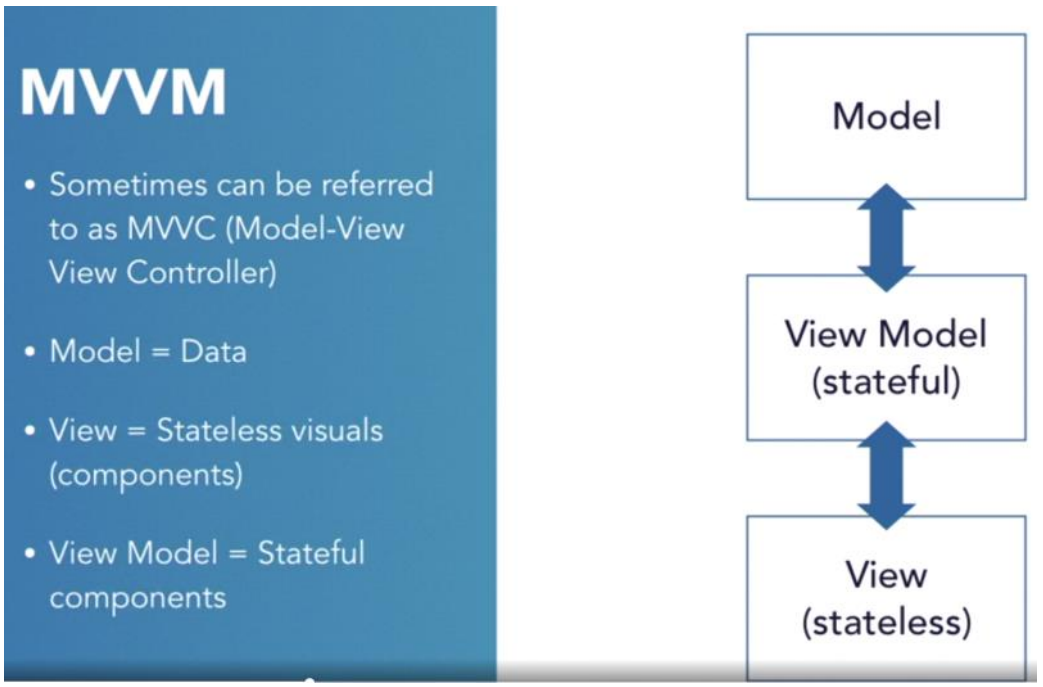
MVP (Model-View-Presenter) Pattern

- The views have access to both models and controllers in an **MVC** model. Where **MVP** differs is the view doesn't have access to the model. It has to get it from the presenter. And the presenter serves as the logic and supplier of data.
- In this pattern, the view passes through the presenter to get the data through functions. And the presenter pulls from the model. It is the major difference.
- The MVP pattern is seen in several frameworks, such as Backbone, but is quite popular in Android development. So if you plan on developing with Android, understanding this pattern will help you tremendously.



MVVM (Model-View View Controller) Pattern

- The first view is your view which doesn't have any data or logic. It is simply a dumb component, or component without any logic or data, which is the view.
- Then you have the second view model, or view controller in MVVC, which holds the logic and the state of the data. And this view model connects to a model.
- MVVM approach can be seen in action with applications built using react and angular.
- For example, in react, your application is architectural in stateless components, which are views. Stateful components which hold data and logic, therefore the view model, and then, finally, the model, is where react typically connects to a back end to process data, where your models are defined.



Behavioral Patterns

Observer Design Pattern

- The observer pattern is one where we maintain a list of objects, based on events, and is typically done with updating data based on events. It is implemented for example with the subscribe and publish methods in MeteorJS.
- It is a crucial **behavioral design pattern** that defines one-to-many dependencies between objects so that when one object (publisher) changes its state, all the other dependent objects (subscribers) are notified and updated automatically. This is also called PubSub (publisher/subscribers) or event dispatcher/listeners pattern.
- The publisher is sometimes called the subject, and the subscribers are sometimes called observers.
- Chances are, you're already somewhat familiar with this pattern if you have used *addEventListener* or jQuery's .on to write even-handling code. It has its influences in Reactive Programming (think RxJS) as well.
- In the example, we create a simple *Subject* class that has methods to add and remove objects of *Observer* class from subscriber collection. Also, a *fire* method to propagate any changes in the *Subject* class object to the subscribed Observers. The *Observer* class, on the other hand, has its internal state and a method to update its internal state based on the change propagated from the *Subject* it has subscribed to.

```
class Subject {
  constructor() {
    this._observers = [];
  }

  subscribe(observer) {
    this._observers.push(observer);
  }

  unsubscribe(observer) {
    this._observers = this._observers.filter(obs => observer !== obs);
  }

  fire(change) {
    this._observers.forEach(observer => {
      observer.update(change);
    });
  }
}
```

```
class Observer {
  constructor(state) {
    this.state = state;
    this.initialState = state;
  }

  update(change) {
    let state = this.state;
    switch (change) {
      case 'INC':
        this.state = ++state;
        break;
      case 'DEC':
        this.state = --state;
        break;
      default:
        this.state = this.initialState;
    }
  }
}
```

```
// usage
const sub = new Subject();

const obs1 = new Observer(1);
const obs2 = new Observer(19);

sub.subscribe(obs1);
sub.subscribe(obs2);

sub.fire('INC');

console.log(obs1.state); // 2
console.log(obs2.state); // 20
```

State Design Pattern

- It is a **behavioral design pattern** that allows an object to alter its behavior based on changes to its internal state. The

object returned by a state pattern class seems to change its class. It provides state-specific logic to a limited set of objects in which each object type represents a particular state.

- Basically the state pattern is one were we hold the state of the application with all the data and properties needed, sometimes called props, in React. And when it changes, it updates the rendering of the application. And again, needless to say, React, Angular, and every state management library are a great example of its use.
- We will take a simple example of a traffic light to understand this pattern. The *TrafficLight* class changes the object it returns based on its internal state, which is an object of *Red*, *Yellow*, or *Green* class.

```
class TrafficLight {
  constructor() {
    this.states = [new GreenLight(), new RedLight(), new YellowLight()];
    this.current = this.states[0];
  }

  change() {
    const totalStates = this.states.length;
    let currentIndex = this.states.findIndex(light => light === this.current);
    if (currentIndex + 1 < totalStates) this.current = this.states[currentIndex + 1];
    else this.current = this.states[0];
  }

  sign() {
    return this.current.sign();
  }
}

class Light {
  constructor(light) {
    this.light = light;
  }
}

class RedLight extends Light {
  constructor() {
    super('red');
  }

  sign() {
    return 'STOP';
  }
}
```

```
class YellowLight extends Light {
  constructor() {
    super('yellow');
  }

  sign() {
    return 'STEADY';
  }
}

class GreenLight extends Light {
        constructor() {
                super('green');
        }

        sign() {
                return 'GO';
        }
}
```

```
// usage
const trafficLight = new TrafficLight();

console.log(trafficLight.sign()); // 'GO'
trafficLight.change();

console.log(trafficLight.sign()); // 'STOP'
trafficLight.change();

console.log(trafficLight.sign()); // 'STEADY'
trafficLight.change();

console.log(trafficLight.sign()); // 'GO'
trafficLight.change();

console.log(trafficLight.sign()); // 'STOP'
```

Chain of Responsibility Design Pattern

- The chain of responsibility is a pattern to help solve common practical issues of having a request from a client and needing this request to pass through multiple functions or logic to get the result. This is where chain of responsibility comes into play.
- This is a **behavioral design pattern** that provides a chain of loosely coupled objects. Each of these objects can choose to act on or handle the request of the client
- A good example of the chain of responsibility pattern is the event bubbling in DOM in which an event propagates through a series of nested DOM elements, one of which may have an “event listener” attached to listen to and act on the event.
- In this example, we create a class *CumulativeSum*, which can be instantiated with an optional *initialValue*. It has a method *add* that adds the passed value to the *sum* attribute of the object and returns the *object* itself to allow chaining of *add* method calls.
- This is a common pattern that can be seen in jQuery as well, where almost any method call on a jQuery object returns a jQuery object so that method calls can be chained together.

```
class CumulativeSum {
  constructor(intialValue = 0) {
    this.sum = intialValue;
  }

  add(value) {
    this.sum += value;
    return this;
  }
}

// usage
const sum1 = new CumulativeSum();
console.log(sum1.add(10).add(2).add(50).sum); // 62


const sum2 = new CumulativeSum(10);
console.log(sum2.add(10).add(20).add(5).sum); // 45
```

Iterator Design Pattern

- It is a **behavioral design pattern** that provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
  - The iterator pattern is a method of iterating through a list of items, whereas the chain of responsibility would use more of a handler type and go through a chain. The iterator is best used with a for loop and is perfect when you want to iterate through arrays of objects.
- Iterators have a special kind of behavior where we step through an ordered set of values one at a time by calling *next()* until we reach the end. The introduction of Iterator and Generators in ES6 made the implementation of the iterator pattern extremely straightforward.
- We have two examples below. First, one *IteratorClass* uses iterator spec, while the other one *iteratorUsingGenerator* uses generator functions.
- The *Symbol.iterator* (*Symbol*—a new kind of primitive data type) is used to specify the default iterator for an object. It must be defined for a collection to be able to use the *for...of* looping construct. In the first example, we define the constructor to store some collection of data and then define *Symbol.iterator*, which returns an object with *next* method for iteration.
- For the second case, we define a generator function passing it an array of data and returning its elements iteratively using *next* and *yield*. A generator function is a special type of function that works as a factory for iterators and can explicitly maintain its own internal state and yield values iteratively. It can pause and resume its own execution cycle.

```
// using Iterator
class IteratorClass {
  constructor(data) {
    this.index = 0;
    this.data = data;
  }

  [Symbol.iterator]() {
    return {
      next: () => {
        if (this.index < this.data.length) {
          return { value: this.data[this.index++], done: false };
        } else {
          this.index = 0; // to reset iteration status
          return { done: true };
        }
      },
    };
  }
}

// using Generator
function* iteratorUsingGenerator(collection) {
  var nextIndex = 0;

  while (nextIndex < collection.length) {
    yield collection[nextIndex++];
  }
}
```

```
// usage
const gen = iteratorUsingGenerator(['Hi', 'Hello', 'Bye']);

console.log(gen.next().value); // 'Hi'
console.log(gen.next().value); // 'Hello'
console.log(gen.next().value); // 'Bye'
```

- Another example, where we iterate through a list of objects

```
newsFeed = [
    {
        type: 'top-headlines',
        query: 'sources=bbc-news'
    },
    {
        type: 'everything',
        query: 'domains=techcrunch.com&language=en'
    },
    {
        type: 'everything',
        query: 'domains=comicbookmovie.com&language=en'
    }
]

for (let feed of newsFeeds) {
    console.log(feed.type);
}
```

Strategy Pattern

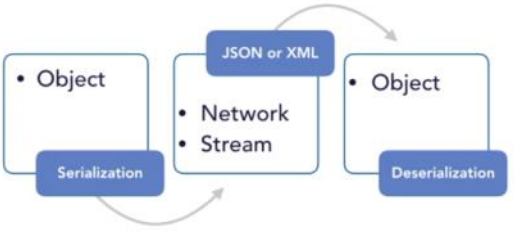
- It is a **behavioral design pattern** that allows encapsulation of alternative algorithms for a particular task. It defines a family of algorithms and encapsulates them in such a way that they are interchangeable at runtime without client interference or knowledge.
- The strategy pattern is basically a way to encapsulate different algorithms or functions. And then at round time, practically use the same code to run different scenarios.
- In the example below, we create a class *Commute* for encapsulating all the possible strategies for commuting to work. Then, we define three strategies namely *Bus*, *PersonalCar*, and *Taxi*. Using this pattern we can swap the implementation to use for the *travel* method of the *Commute* object at runtime.

```
// encapsulation
class Commute {
  travel(transport) {
    return transport.travelTime();
  }
}

class Vehicle {
  travelTime() {
    return this._timeTaken;
  }
}

// strategy 1
class Bus extends Vehicle {
  constructor() {
    super();
    this._timeTaken = 10;
  }
}

// strategy 2
class Taxi extends Vehicle {
  constructor() {
    super();
    this._timeTaken = 5;
  }
}
```

```
// strategy 3
class PersonalCar extends Vehicle {
  constructor() {
    super();
    this._timeTaken = 3;
  }
}

// usage
const commute = new Commute();

console.log(commute.travel(new Taxi())); // 5
console.log(commute.travel(new Bus())); // 10
```

Memento Design Pattern

- The Memento Pattern is basically providing a temporary state of an object and restoration of that object from a conversion into a different format or whatnot.
- It is often seen in serialization and deserialization of data. In this use case, an object is typically in a JavaScript object where we need to transmit the data to some type of backend APIs where we serialize this object into JSON for streaming it through HTTP protocols.
- Then, when this object hits the backend server or when we get a response back from the server through HTTP, we might need to deserialize the JSON object back into a JavaScript object for consumption into our application.
- Well, this is basically the Memento Pattern in action, where the data never loses its accuracy despite several conversions in between format.



Mediator Pattern

- It is a **behavioral design pattern** that encapsulates how a set of objects interact with each other. It provides the central authority over a group of objects by promoting loose coupling, keeping objects from referring to each other explicitly.
- The mediator pattern provides a set of objects which interact with each other, mostly by having a central authority dictating the terms in between objects.
- In this example, we have *TrafficTower* as Mediator that controls the way *Airplane* objects interact with each other. All the *Airplane* objects register themselves with a *TrafficTower* object, and it is the mediator class object that handles how an *Airplane* object receives coordinates data of all the other *Airplane* objects.

```
class TrafficTower {
  constructor() {
    this._airplanes = [];
  }

  register(airplane) {
    this._airplanes.push(airplane);
    airplane.register(this);
  }

  requestCoordinates(airplane) {
    return this._airplanes.filter(plane => airplane !== plane).map(plane => plane.coordinates);
  }
}

class Airplane {
  constructor(coordinates) {
    this.coordinates = coordinates;
    this.trafficTower = null;
  }

  register(trafficTower) {
    this.trafficTower = trafficTower;
  }

  requestCoordinates() {
    if (this.trafficTower) return this.trafficTower.requestCoordinates(this);
    return null;
  }
}

```

```
// usage
const tower = new TrafficTower();

const airplanes = [new Airplane(10), new Airplane(20), new Airplane(30)];
airplanes.forEach(airplane => {
  tower.register(airplane);
});

console.log(airplanes.map(airplane => airplane.requestCoordinates()))
// [[20, 30], [10, 30], [10, 20]]
```

Command Pattern

- This is a **behavioral design pattern** that aims to encapsulate actions or operations as objects. This pattern allows loose coupling of systems and classes by separating the objects that request an operation or invoke a method from the ones that execute or process the actual implementation.
- The command pattern is one that encapsulates actions or operations as objects. So, in other words, in this pattern, you abstract the actual function or execution of the action from the action itself.
- The clipboard interaction API somewhat resembles the command pattern. If you are a Redux user, you have already come across the command pattern. The actions that allow the awesome time-travel debugging feature are nothing but encapsulated operations that can be tracked to redo or undo operations. Hence, time-travelling made possible.
- In this example, we have a class called *SpecialMath* that has multiple methods and a *Command* class that encapsulates commands that are to be executed on its subject, i.e. an object of the *SpecialMath* class. The *Command* class also keeps track of all the commands executed, which can be used to extend its functionality to include undo and redo type operations.

```
class SpecialMath {
  constructor(num) {
    this._num = num;
  }

  square() {
    return this._num ** 2;
  }

  cube() {
    return this._num ** 3;
  }

  squareRoot() {
    return Math.sqrt(this._num);
  }
}

class Command {
  constructor(subject) {
    this._subject = subject;
    this.commandsExecuted = [];
  }
  execute(command) {
    this.commandsExecuted.push(command);
    return this._subject[command]();
  }
}
```

```
// usage
const x = new Command(new SpecialMath(5));
x.execute('square');
x.execute('cube');

console.log(x.commandsExecuted); // ['square', 'cube']
```

- Another example is Redux, where in the reducers we use the actions which are specified in another file
  - **Action Creators**

    actions/index.js

    ```
    let nextTodoId = 0
    export const addTodo = text => ({
      type: 'ADD_TODO',
      id: nextTodoId++,
      text
    })

    export const setVisibilityFilter = filter => ({
      type: 'SET_VISIBILITY_FILTER',
      filter
    })

    export const toggleTodo = id => ({
      type: 'TOGGLE_TODO',
      id
    })
    ```

  - **Reducers**

    reducers/todos.js

    ```
    const todos = (state = [], action) => {
      switch (action.type) {
        case 'ADD_TODO':
          return [
            ...state,
            {
              id: action.id,
              text: action.text,
              completed: false
            }
          ]
        case 'TOGGLE_TODO':
          return state.map(todo =>
            (todo.id === action.id)
              ? {...todo, completed: !todo.completed}
              : todo
          )
        default:
    ```

ES6

Monday, November 11, 2019 2:54 PM

ECMA Script 6

ECMA - European Computer Manufacturer Association

Transpiling - The process of converting code to a format that can be read by a browser