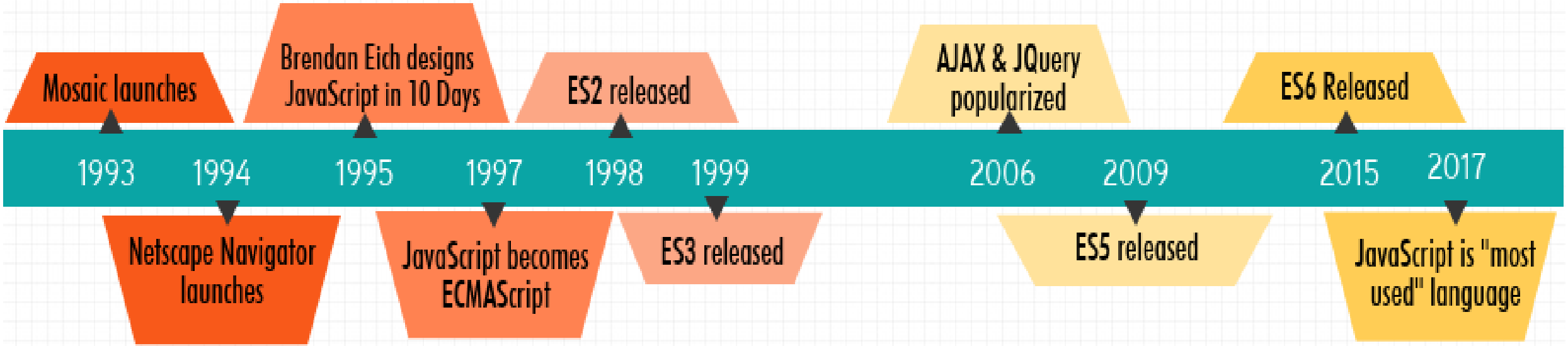
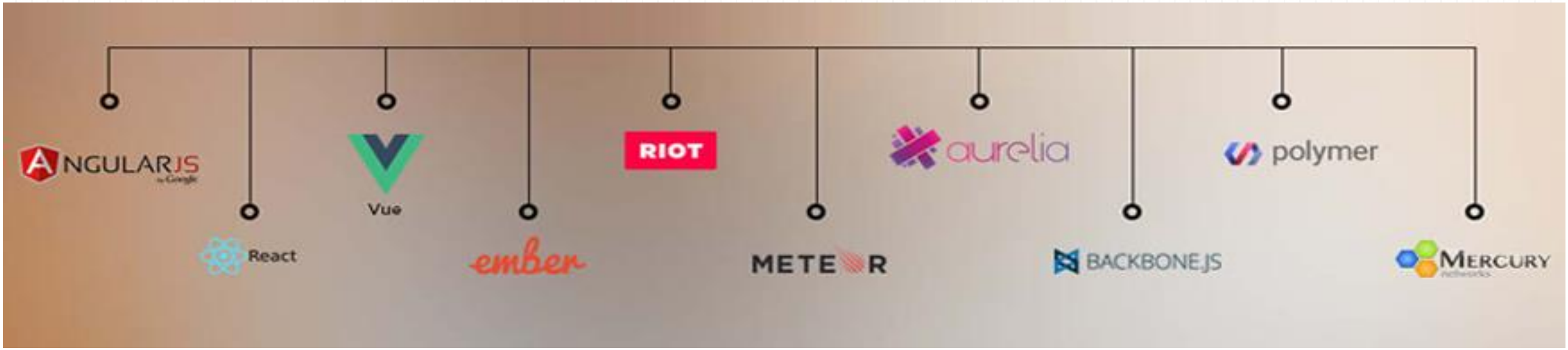
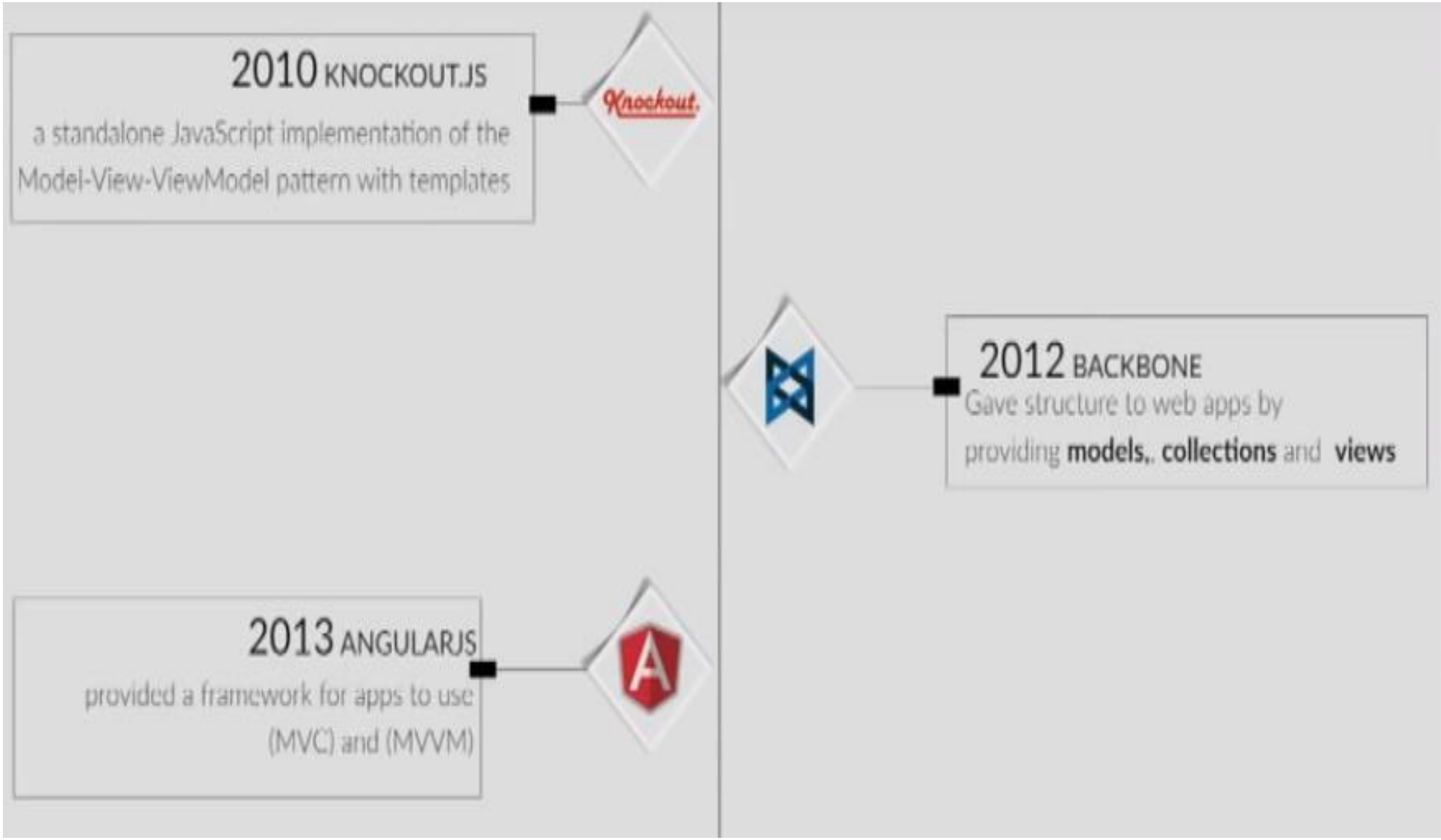
Karthik Raman

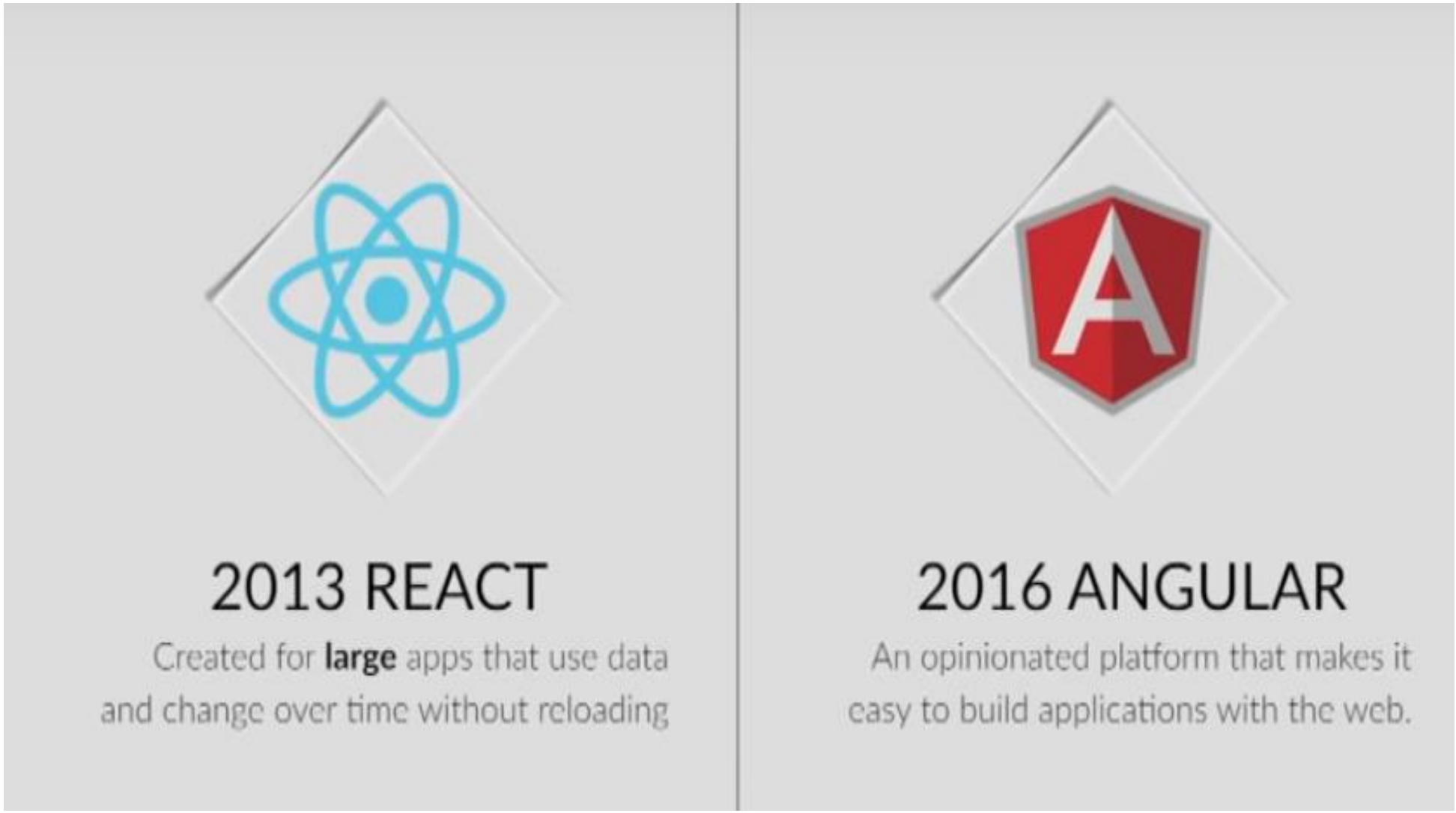
- High level Interpreted Scripted Programming Language
- Dynamic type checking (weakly typed)
- Multi-Paradigm (Structural , Object Oriented , Functional)
- Compliance to ECMAScript Specifications
- Runs on the Client Browser
- Runs on the Server (Node.js)

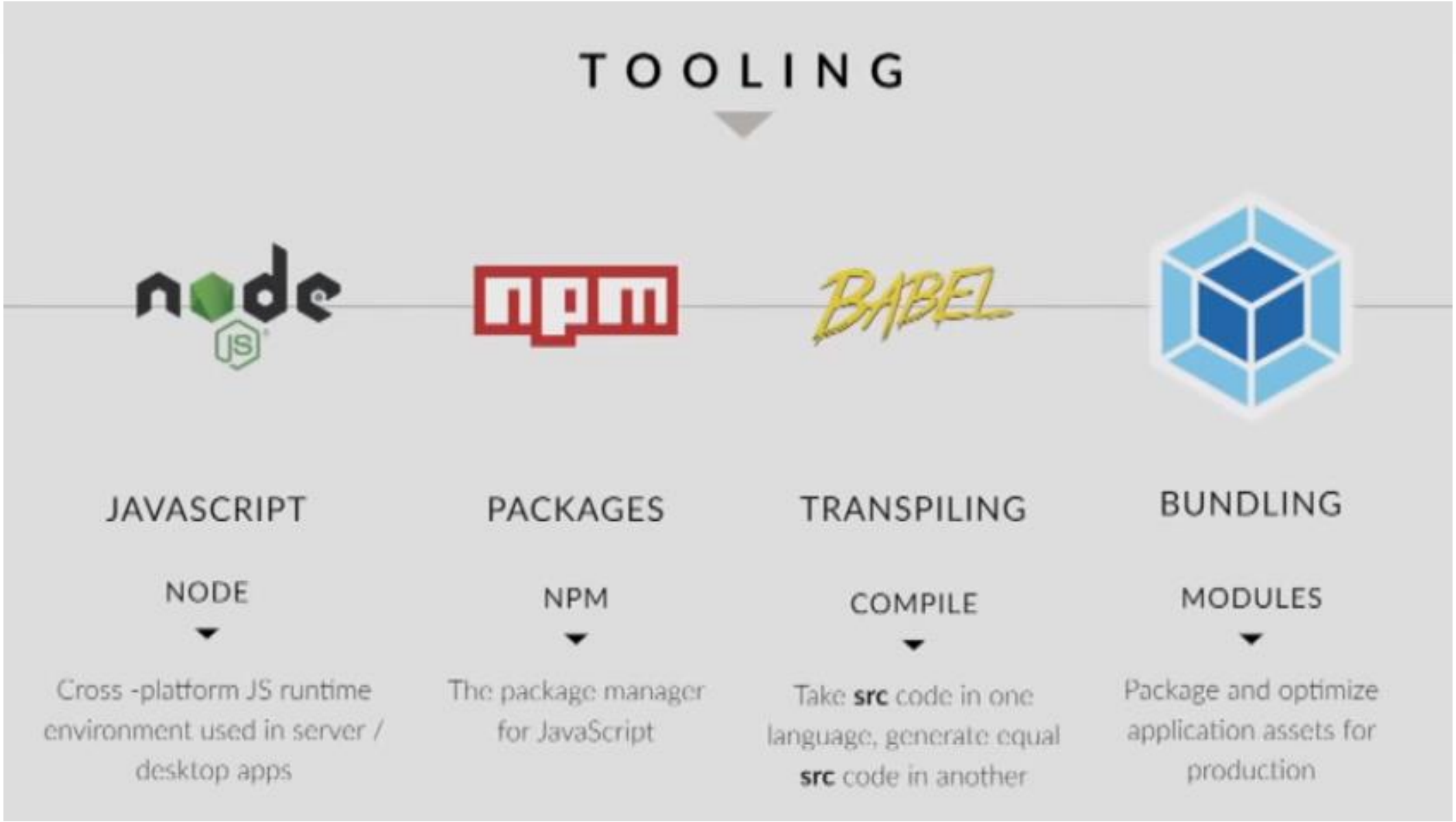
JavaScript Evolution





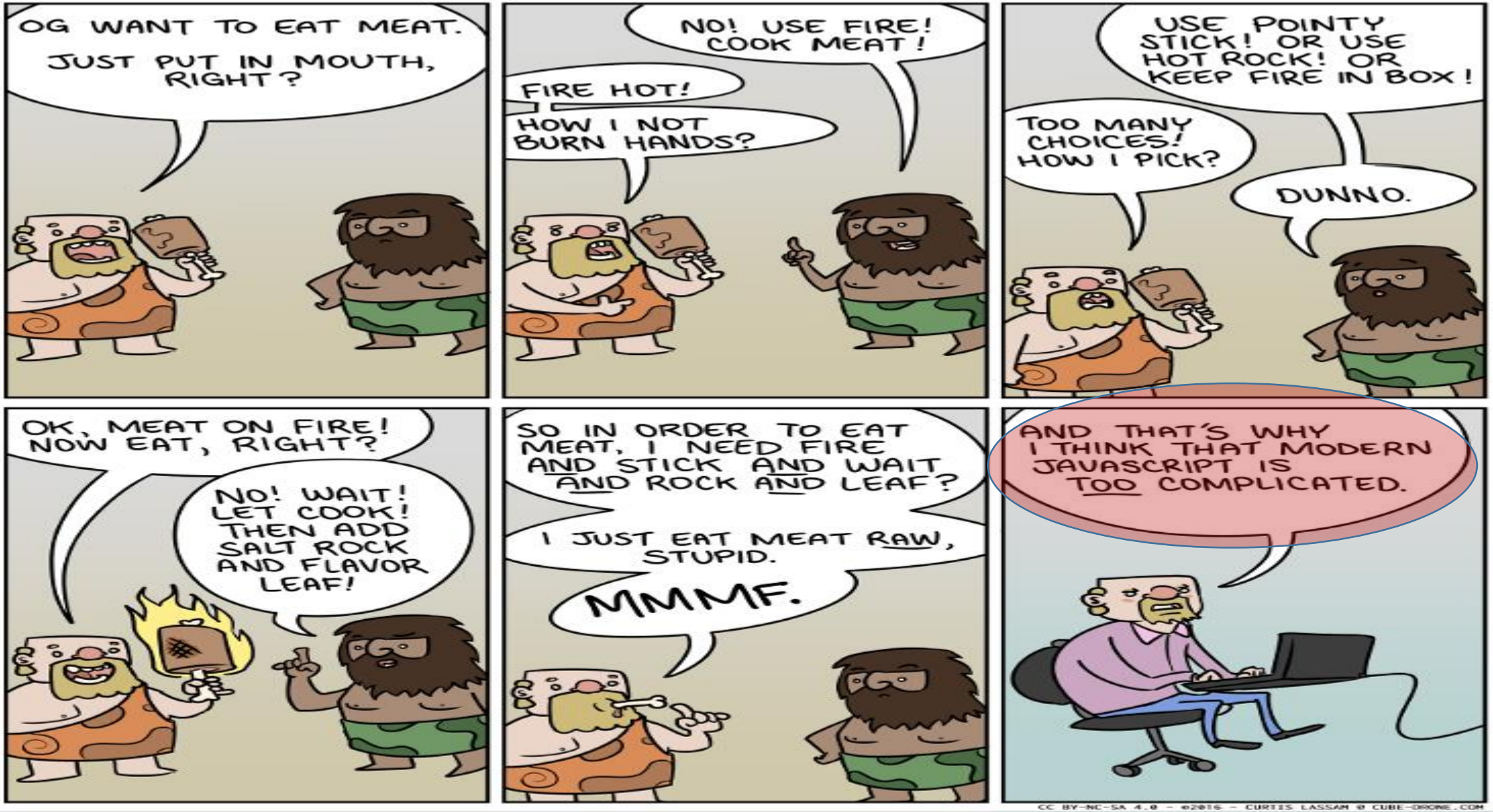






JavaScript Complexity

Clayfin



- Inline, between a pair of <script> and </script> tags

```
<script>
        JavaScript statements.....
</script>
```

- In an HTML event handler attribute, such as onclick or onmouseover

```
<button onclick="myFunction()">Click me</button>
```

- From an external file specified by the src attribute of a <script> tag

```
<script src = "common.js"> </script>
```

- In a URL that uses the special javascript: protocol

```
<a href="javascript:void(0);"></a>
```

```
console.log (“Hello World”);
document.write(“Hello World”);
```

- console.log
- console.error
- console.warn
- console.trace
- console.info
- console.dir

- Visual Studio Code
- WebStorm
- Atom
- Sublime Text
- Brackets

- How to define variables in Javascript
  3 ways : var , let , const
- let and const in ES6 or ECMA2015

- ***var*** *keyword declares a variable which is scoped to its current execution context, optionally initializing to a value*
- ***let*** *keyword declares a block scoped variable, optionally initializing it to a value*
- ***const*** *keyword declares constants which are block scoped much like variables defined using* ***let*** *but the value of a constant cannot change. The* ***const*** *declaration creates a read-only reference to a value*

- Variables declared with ***var*** keyword can be redeclared at any point in the code even within the same execution context. This is not the case for variables defined with ***let*** and ***const*** keywords as they can only be declared once within their lexical scope

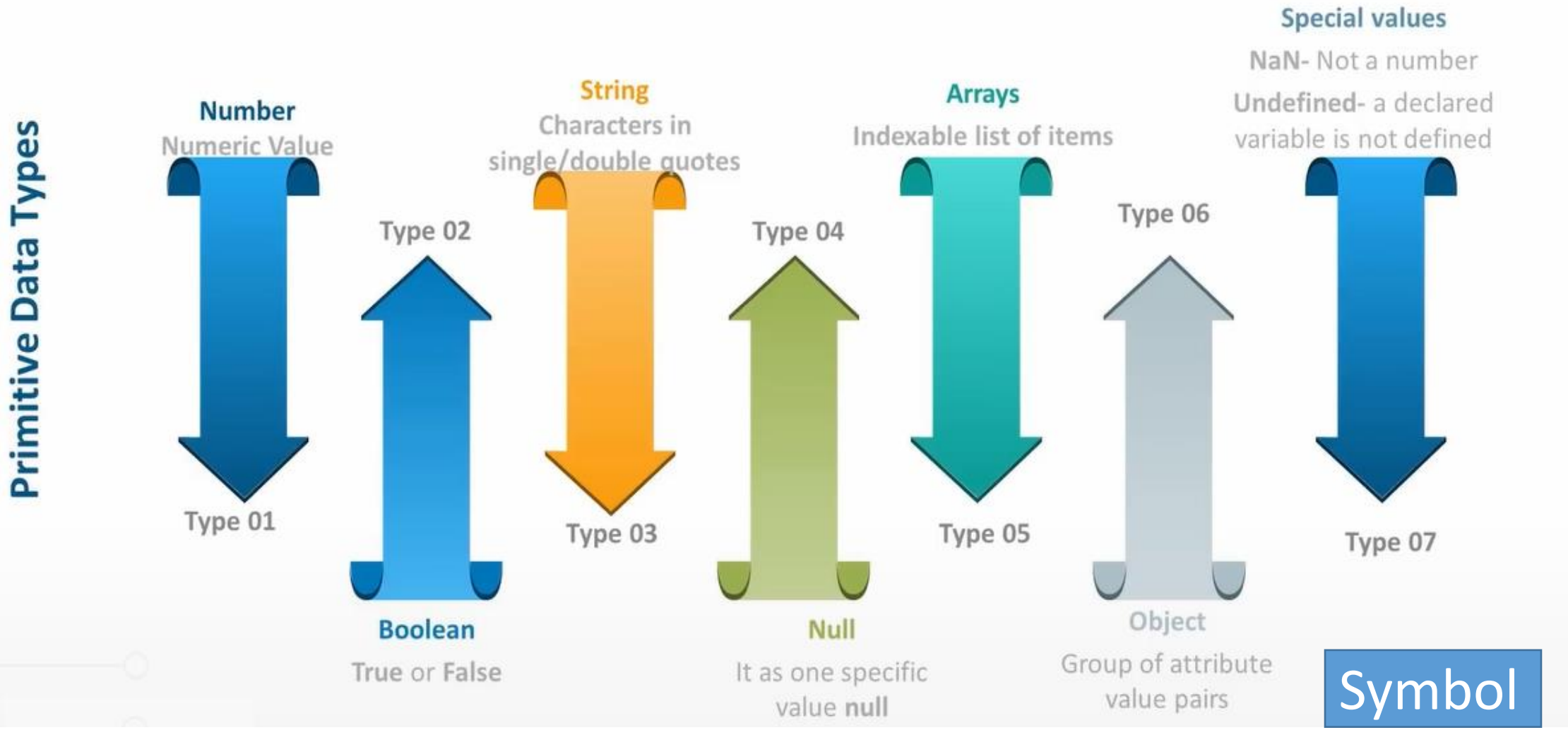
- Variables declared with ***const*** even though its value cannot be reassigned to, it is still mutable

Scope.html

- The primary importance of hoisting is that it allows you to use functions before you declare them in your code
- variable and function declarations are put into memory during the compilation phase but stay exactly where you typed them in your code
- all variables defined with the ***var*** keyword have an initial value of ***undefined***.
- variables defined with ***let*** or ***const*** keywords when hoisted are in a state called the **Temporal Dead Zone** and are not initialized until their definitions are evaluated

Hoisting.html

JavaScript – Data Types



- let name = ‘Karthik’;
- let age = 30;

Concatenation

- console.log(“My name is “ + name + “ and aged “ + age);

- **Template Strings**

- **console.log(`My name is ${name}  and aged ${age}`);**

- **Length , toLowerCase , substring , split**

JavaScript – Type Conversions

- Type Conversions/ Type Casting: A process where an entity of one data type is converted to another

Implicit Conversion - Integers converted to Strings and back automatically

```
<script>
        var num1=5;
        var num2=num1+5;          /* num2 is assigned value 10 as, num1 is type casted to integer*/
        var num3=num1 +"5";     /* num3 is assigned value 55 as, num1 is type casted to string*/


</script>
```

num1 Implicitly converted to type Integer

num1 Implicitly converted to type String

TypeConversion.html

JavaScript – Type Conversions

Explicit Conversion - Use JavaScript functions like **parseInt()**, **parseFloat()** etc.

```
<script>
num1 = prompt("Enter 1st Real/Floating-point Number: ");
num2 = prompt("Enter 2nd Real/Floating-point Number: ");
alert("The sum of real numbers is: " + (parseFloat(num1) + parseFloat(num2))); /*string to float conversion*/


</script>
```

Inputs num1 and num2 of type String

num1 and num2 of type String type casted to Float

JavaScript – Arrays

Declaration

Example-

```
var space=[“ moon ”, “ star”, ” sun”];
```

OR

```
var space= new Array(“moon ”, ” star”, ” sun”);
```

Accessing elements

Example-

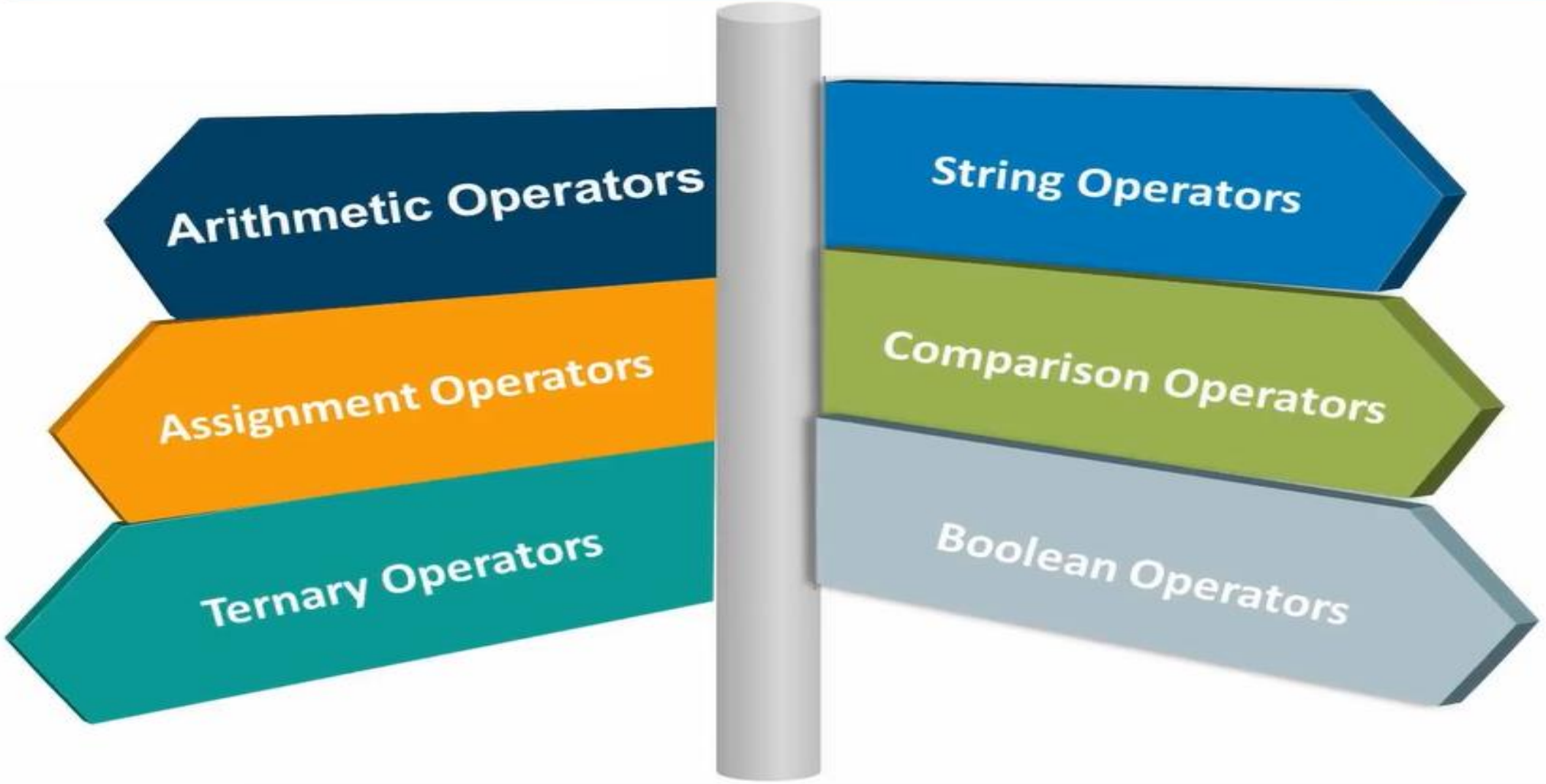
```
var bodies= space[0] +space[1]+space[3];
/*bodies will have the value “moon star sun” */
```

OR

```
space[0]=“planet”;   /*the first element of the space array will have the value “planet” */
```

Arrays.html

JavaScript – Operators



JavaScript – Operators

Arithmetic Operators

+: addition
- : subtraction
* : multiplication
/ : division
% : modulus
++ : increment
- - : decrement
- : unary minus

String Operator

+ : concatenation

Assignment Operators

= : assignment
+= : add, assign
-= : subtract, assign
*= : multiply, assign
/= : division, assign
%= : mod, assign

Comparison Operators

== : equal
!= : not equal
> : greater
< : lesser
>= : greater/equal
<= : lesser/equal
=== : equal value and same type
!== : not equal value or not same type

JavaScript – Operators

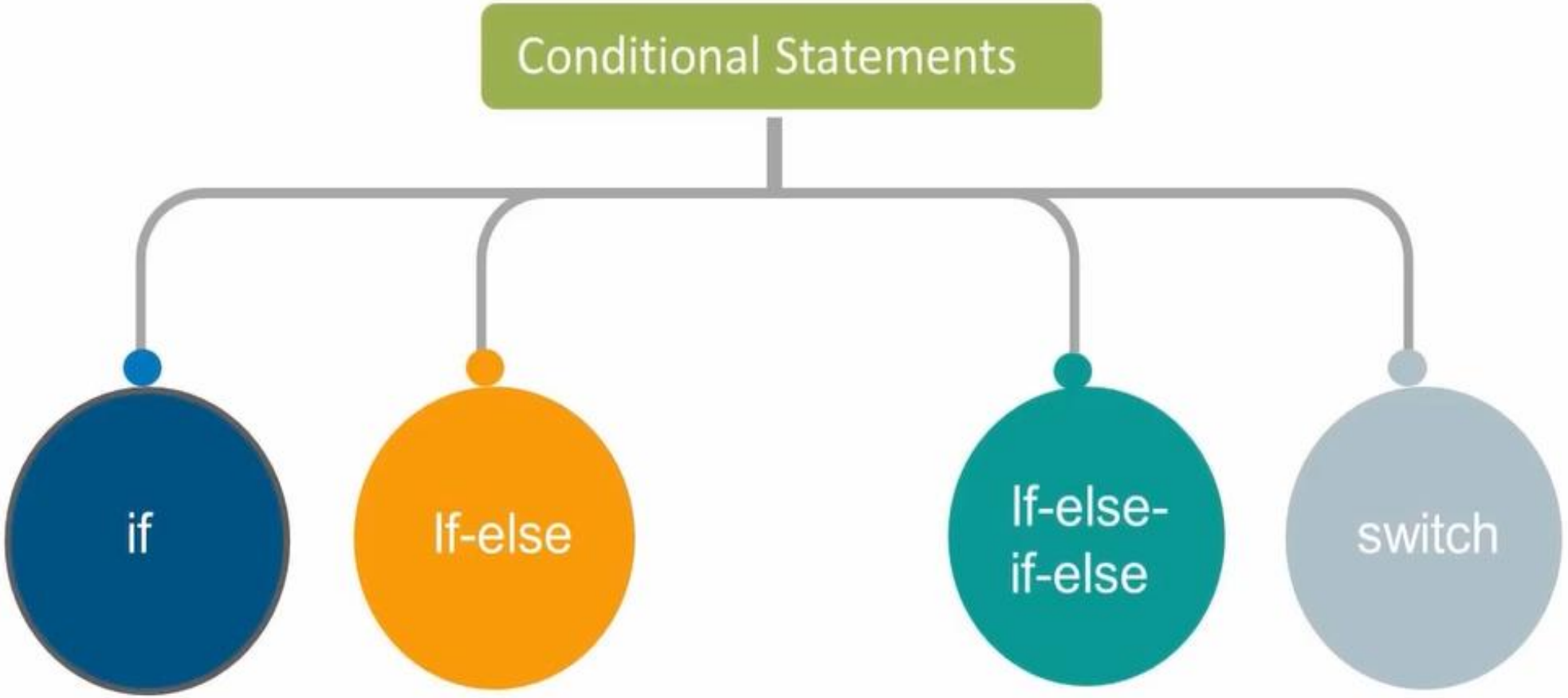
Boolean Operators

&& : AND
|| : OR
! : NOT

Ternary Operators

```
variable_name=
(condition)?
value1:value2;
/* if the condition is
true variable_name will
be assigned value1,or
else value2  */
```

JavaScript – Conditional Statements



if-else-if-else – Conditional Statement

```
<script>
age = prompt("Enter Your age:");
age = parseInt(age);
if (age > 60) {
    document.write("As you are more than 60 years old, you have to control your salt and sugar intake! ");
}
else if (age > 30) {
    document.write("As you are more than 30 years old, you have to take good care of your health!");}
else {
    document.write("As you are young, you can enjoy deep fried pakodas!");  }

</script>
```

Two if conditions are checked, and returns a Boolean value

If both the if conditions return false, then this else block statement is executed

Switch – Conditional Statement

```
10
11  <script>
12  weight = parseFloat(prompt("What is your weight"));
13  switch (weight)
14  {
15      case 10.5:
16          document.write("Your weight is 10.5 Kg<br>");
17          break;
18      case 20.5:
19          document.write("Your weight is 20.5 Kg<br>");
20          break;
21      default:
22          document.write("Your weight : " + weight + " does not match");
23  }
24
25
26  </script>
```

Value of the weight variable is passed in the switch statement

If weight matches the value of the case condition, the following block is executed

If none of the cases match with the weight variable, default case block is executed

Conditions.html

for Loop

- **for (initialization; condition; updation)** – Statement
- The Loop will keep on executing until the condition check returns false

```
<script>
subjects = new Array("Maths", "Physics", "Chemistry");
marks = new Array();
for (var i = 0; i < subjects.length; i++ ) {
    num  = prompt("Enter your marks in: " + subjects[i] + " subject" );
    marks[i] = parseInt(num);
}
msg = "";
for (var i = 0; i < subjects.length; i++ )
    msg += subjects[i]  + " Marks:== " + marks[i] + "\n";
alert(msg);


</script>
```

for-in Loop

- **for (variable in object)** – Statement
- The loop will keep on executing until the all variables in the object are passed in the for statement

```
10
11  <script>
12  subjects = new Array("Maths", "Physics", "Chemistry");
13  marks = new Array();
14  for ( i in subjects ) {
15      num  = prompt("Enter your marks in: " + subjects[i] + " subject" );
16      marks[i] = parseInt(num);
17  }
18  msg = "";
19  for ( i in subjects )
20      msg += subjects[i]  + " Marks:== " + marks[i] + "\n";
21  alert(msg);
22
23  </script>
```

while Loop

do – while statement

- **while (condition)** – Statement
- The while block will be executed unless the condition statement returns a Boolean false

```
<script>
subjects = new Array("Maths", "Physics", "Chemistry");
marks = new Array(); i = 0;
while ( i < subjects.length ) {
    num  = prompt("Enter your marks in: " + subjects[i] + " subject" );
    marks[i] = parseInt(num);
    i++;  }
msg = "";   i = 0;
while ( i < subjects.length ) {
    msg += subjects[i]  + " Marks:== " + marks[i] + "\n";
    i++;   }
alert(msg);
```

ControlStatements.html

Set - collection of items which are unique

- Can hold both *objects and primitive* values

```
var set1 = new Set([10, 20, 30, 30, 40, 40]);  // Set {10,20,30,40}
set1.add( "Karthik" );
var sz = set1.size;
//Methods add , delete ,  clear , union , intersect, difference

//Iterable  - Symbol.iterator
var set3 = new Set(["Karthik","Kannan","Rajesh"]);
var getit = set3[Symbol.iterator]();
console.log(getit.next());          // prints {value: "Karthik", done: false}
console.log(getit.next());          // prints {value: "Kannan", done: false}
console.log(getit.next());          // prints {value: "Rajesh", done: false}
console.log(getit.next());          // prints {value: undefined, done: true}
```

WeakSet - similar to Set with the following differences

- behave differently when a object referenced by their keys gets deleted
- keys cannot be primitive types
- non-Iterable

```
var set = new Set();
var weakset = new WeakSet();

(function(){
   var a = {x: 12};
   var b = {y: 12};
   set.add(a);
   weakset.add(b);
})()
```

Map - collection of elements stored as a *Key, Value* pair

```
var map1 = new Map([["fname" ,"Karthik"],
      ["lname", "Raman"], ["email", "karthik@clayfin.com"]]);
map1.set( "company","Clayfin" );
var sz = map1.size;
//Methods set, get,  clear , entries , keys, values

//Iterable  - Symbol.iterator

var getit = map1[Symbol.iterator]();
for(var elem of getit)
   console.log(elem);
```

WeakMap is for a Map similar to WeatSet is for a Set

MapAndWeakMap.html

JavaScript – Functions

Functions are building blocks of JavaScript, which can be reused many times with different arguments to produce different results

Functions

- A set of statements (function body) that performs a task or calculates a value
- Defined somewhere in the scope
- Invoked by a function call

Function Parameters

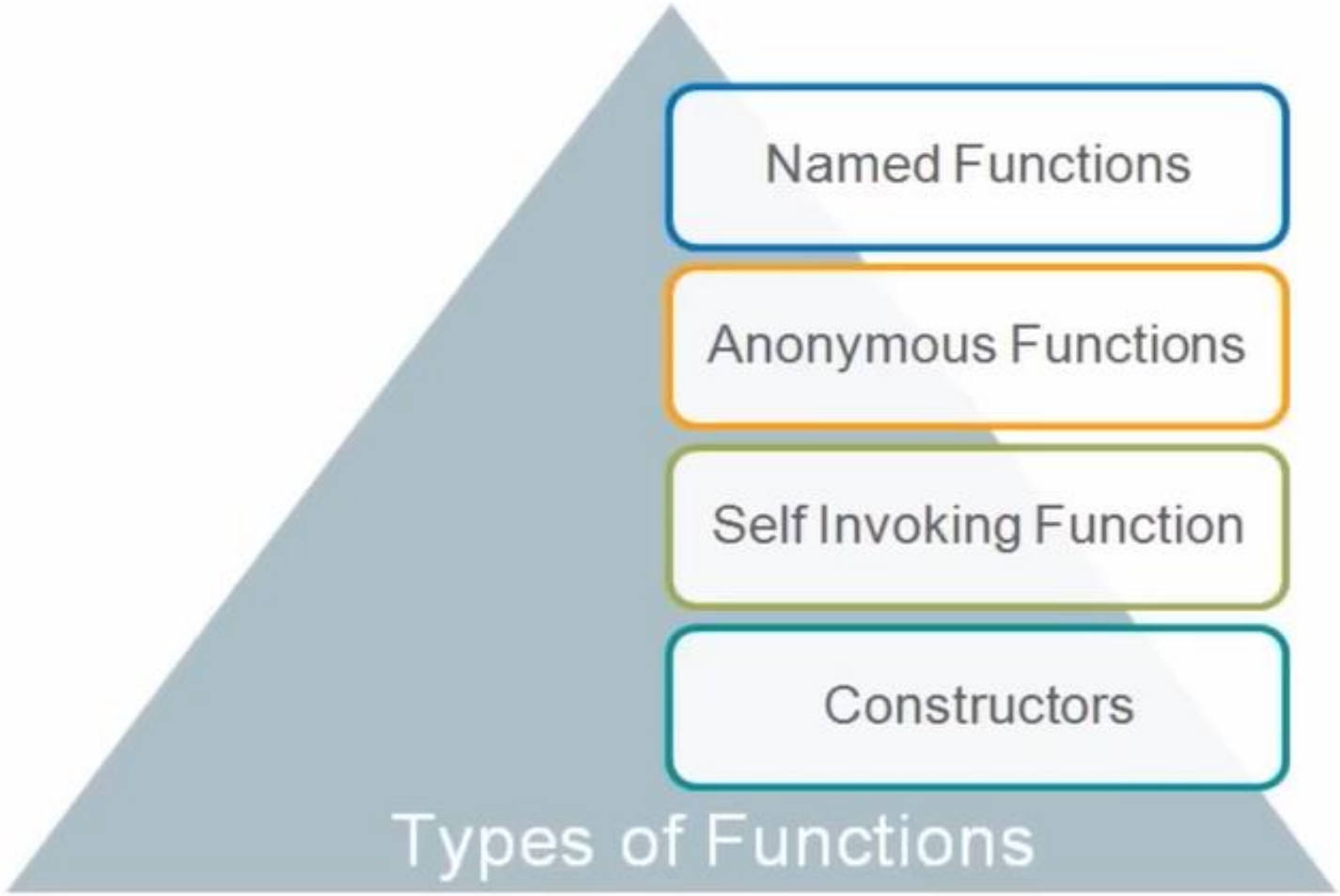
- A function parameter is a value, which is accepted by the function
- Parameters in a function call are arguments
- Arguments are parameters, which are passed to the functions by value

Return Statement

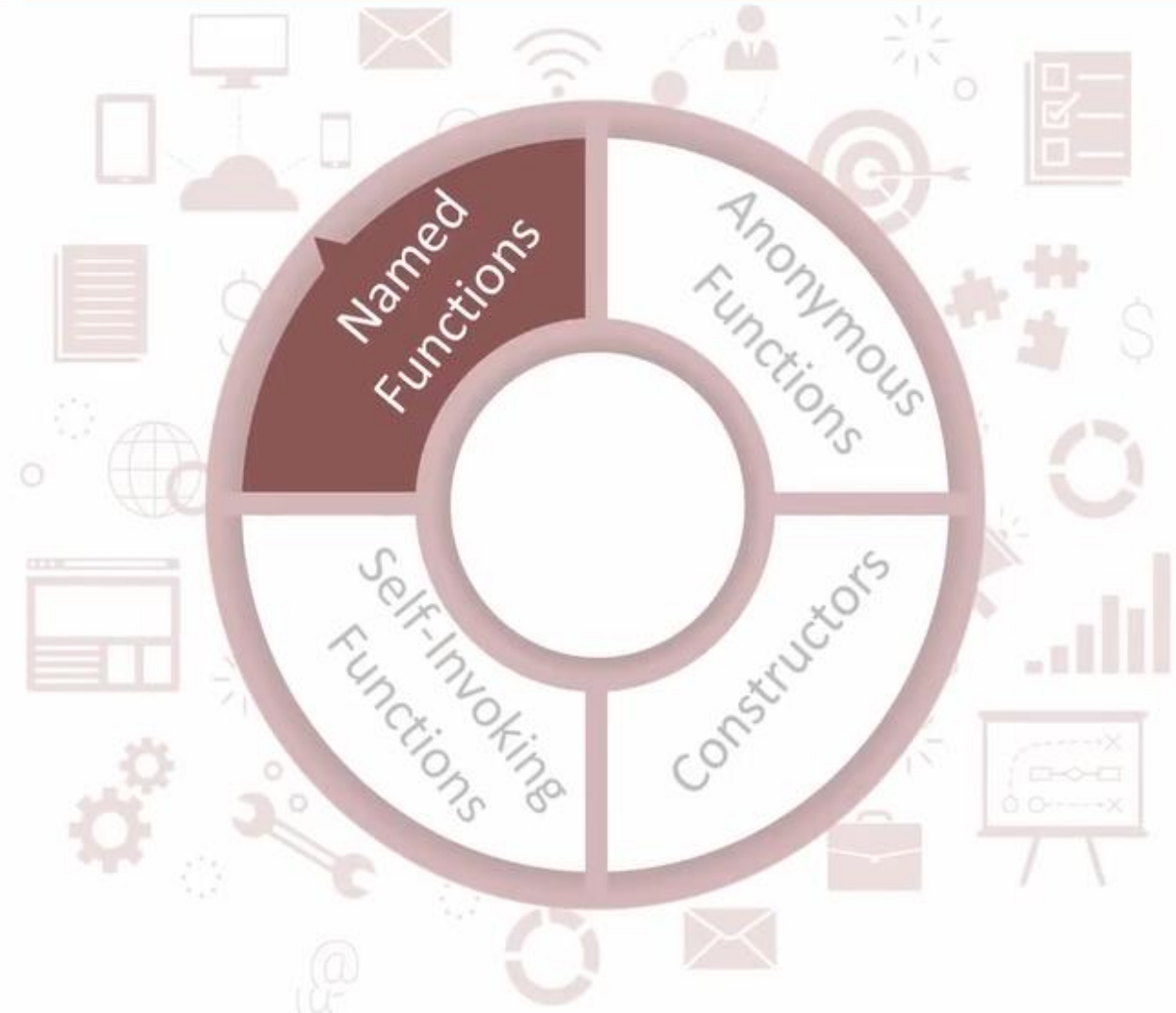
- Every Function should have a return statement in its body, which returns a specific value
- If a function does not have a return statement, a default value is returned
- The default value that is returned is the value **undefined**

Types Functions

On the basis of how it is defined and called, functions are categorized in 4 types :



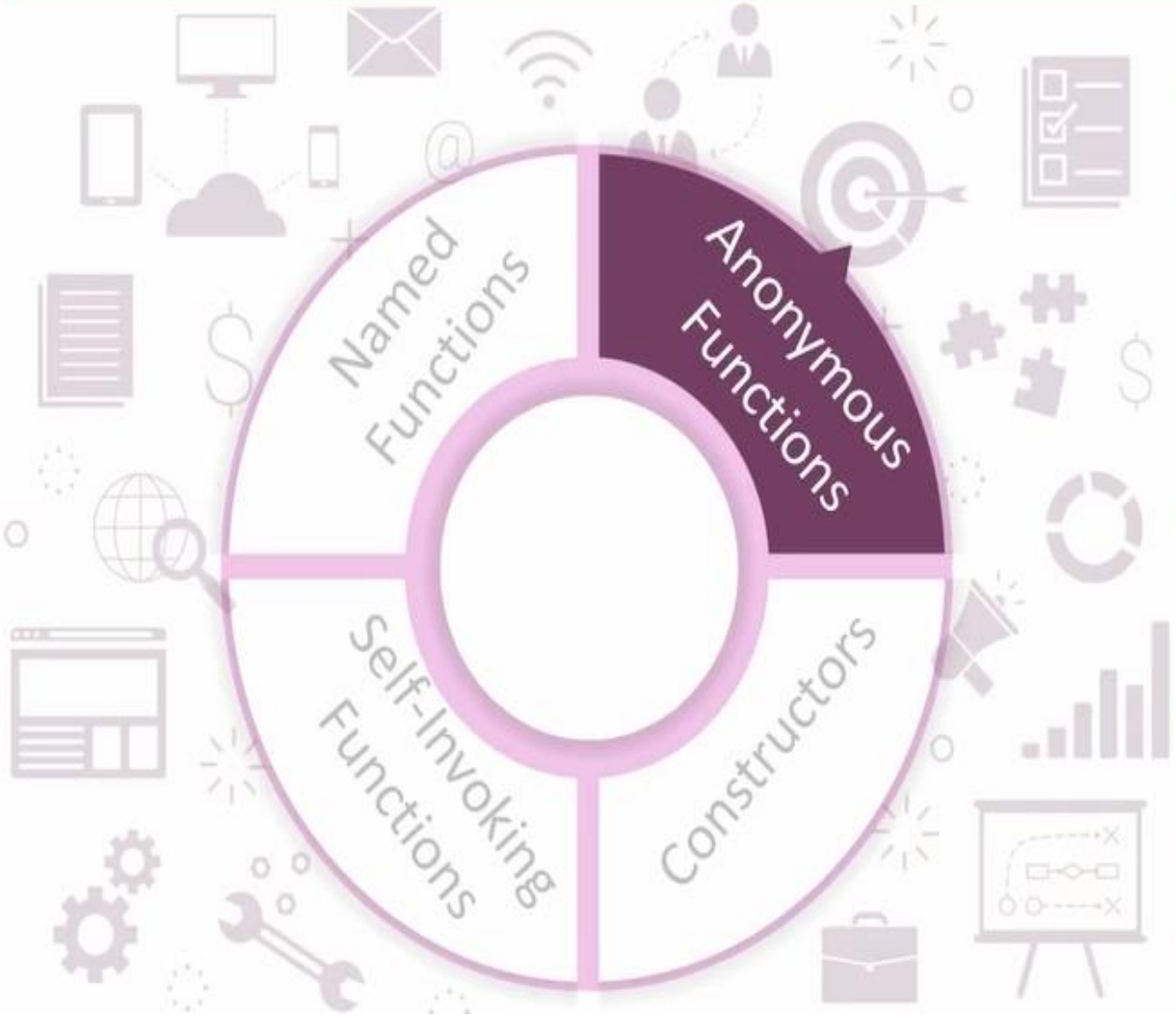
Named Functions



- Has a unique name
- Can be called/used in multiple places
- Example:

```
function addNum(a, b)
{
/*function definition with
parameters a and b*/
return a+b;
/* return statements */
}
var d= addNum(4,3);
/* function call, returns
value 7 */
var c= addNum(5,5);
/* function call, returns
value 10*/
```

Anonymous Functions



- Does not have a name
- Can be used at one place only (when it is called immediately after it is defined, or actual argument to function)
- The function defined is used as an expression
- Can be stored in a variable
- Passed as an actual argument to a function
- Can be returned as a value by a function

Anonymous Function: Example

- Stored in a variable

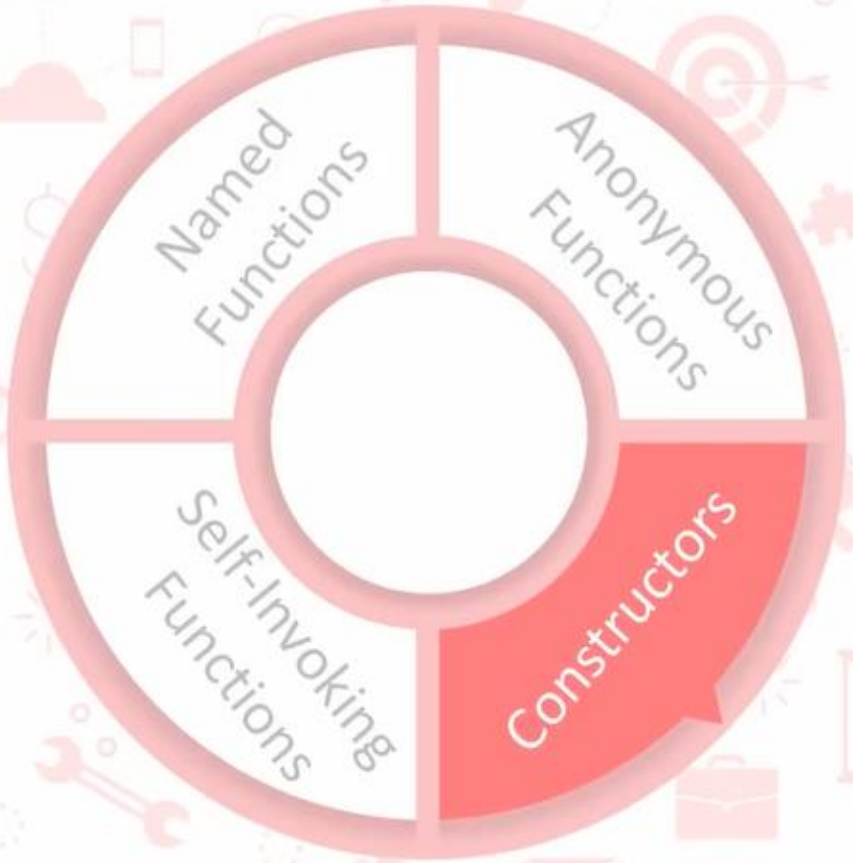
```
var add = function (a, b) { return a+b; };
alert(add(2, 3));
```

- Passed as an argument to another function

```
setTimeout( function()
 {
    alert("this message is displayed after 5 seconds");
}, 5000);
```

No name for the function

Constructors



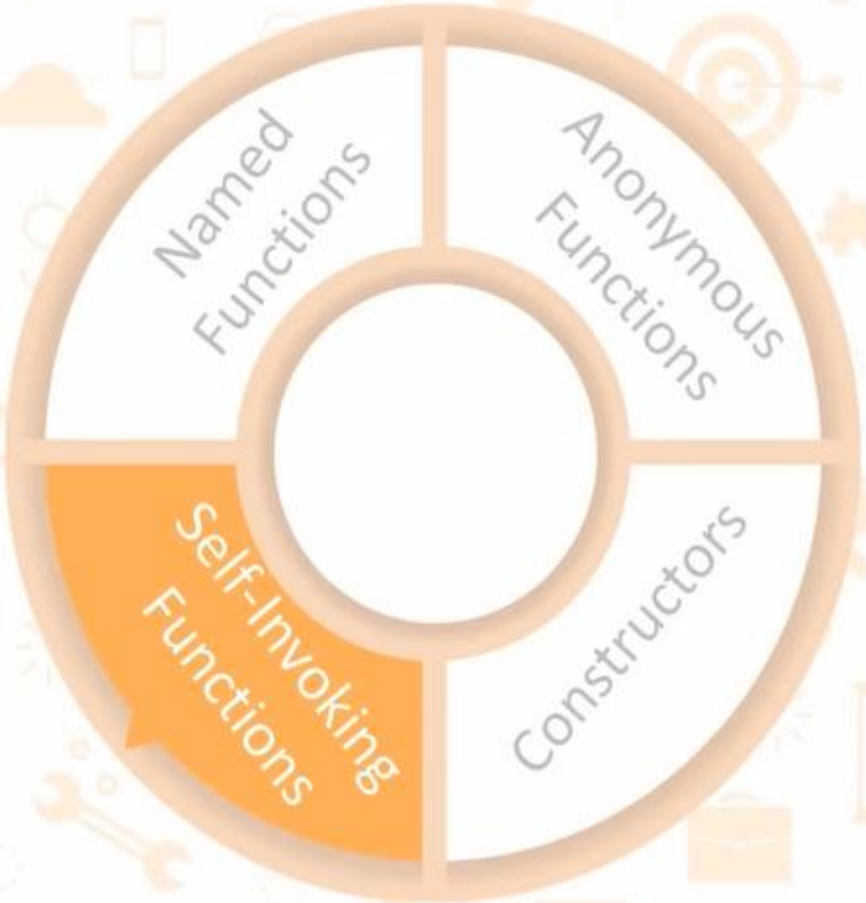
- A constructor is called when an object is created using the new keyword
- Example –

```
var addFunc = new Function("a", "b",
"return a + b;");
var c = addFunc(2,3);
```

is the same as

```
var addFunc = function(a, b) {
return a + b;
};
var c = addFunc(2,3);
```

Self-Invoking Functions



- Self-invoking functions are anonymous functions, which are invoked right after the function has been defined
- You can execute the code once, without declaring any global variables
- No reference is maintained to this function, not even to its return value
- Syntax –

```
(function(){
    //body
}() );
```

```
(function(){
 //body
 }) ();
```

Self-Invoking Functions - Example

```
(function()
 alert("this is a self invoking function");
}) ();
```

OR

```
(function()
alert("this is a self invoking function");
}());
```

```
//named function
function nCalSquare(a){
    return a*a;
}

console.log("calling named function:"+nCalSquare(9));

//anonymous function
var aCal=function (b){
    return b*b;
}
console.log("calling anonymous function:"+aCal(6));

// new constructor
var cCal=new Function("a","return a*a;");
console.log("calling constructor function:"+ cCal(7));

//self invoking function
(function(a){
     console.log("calling self invoking function:");
     return a*a;
    })(9);
```

Default Parameters for Functions

The **JavaScript default parameter** is very useful while writing a function. When calling the function, if a**parameter** is missing, the **default parameter** feature allows you to assign a **default** value to the function**parameter**, rather than leaving it undefined

Functions.html

DefaultParameters.html

- Never use alert for debugging

- console.log
- console.dir
- console.table
- console.time
- console.timeEnd

- debugger

Debugging.html

Object Creation

Using new keyword

```
var person = new Object();
person.fname = “Karthik”;
person.lname = “Raman”;
person.email = “karthik@clayfin.com”;
```

Using { } with attributes as dot(.)

```
var person = { };
person.fname = “Karthik”;
person.lname = “Raman”;
person.email = “karthik@clayfin.com”;
```

Object Creation

Using { } with attributes as [ ]

```
var person = new Object();
person[fname] = “Karthik”;
person[lname] = “Raman”;
person[email] = “karthik@clayfin.com”;
```

Using { } with attributes as :

```
var person = {
        person.fname  :  “Karthik”,
        person.lname  :  “Raman”,
        person.email   :   “karthik@clayfin.com”;
};
```

Object Creation

Using constructor function

```
function  Person () {
      this.fname = “Karthik”;
      this.lname = “Raman”;
      this.email = “karthik@clayfin.com”;
      }
var person = new Person( );
```

Using Object.create()

```
var Person = {
      this.fname = “Karthik”;
      this.lname = “Raman”;
      this.email = “karthik@clayfin.com”;
      }
var person = Object.create(Person);
```

Object property deletion

```
function  Person () {
      this.fname = “Karthik”;
      this.lname = “Raman”;
      this.email = “karthik@clayfin.com”;
      }
var person = new Person( );

delete person.email;
```

Objects are destroyed and enabled for GC if all the references to the objects are removed.

Objects.html

Factory Pattern

ObjectFactory.html

```
var PersonFactory = function  Person (fname,lname,email) {
      var  temp = {};
      temp.fname = fname;
      temp.lname = lname;
      temp.email = email;

      temp.displayName( ) {
           console.log(`${this.fname} ${this.lname}`);
      };
      return temp;
};
var person1 = new PersonFactory (“Karthik”,”Raman”,”karthik@clayfin.com”);
person1.displayName();
```

Constructor Pattern

ObjectConstructor.html

```
var PersonConstructor = function (fname,lname,email) {
this.fname = fname;
this.lname = lname;
this.email = email;

this.displayName = function( ) {
        console.log(`${this.fname} ${this.lname}`);
};

};
var person1 = new PersonConstructor ('Karthik','Raman','karthik@clayfin.com');
person1.displayName();
```

Prototype Pattern

ObjectProto.html

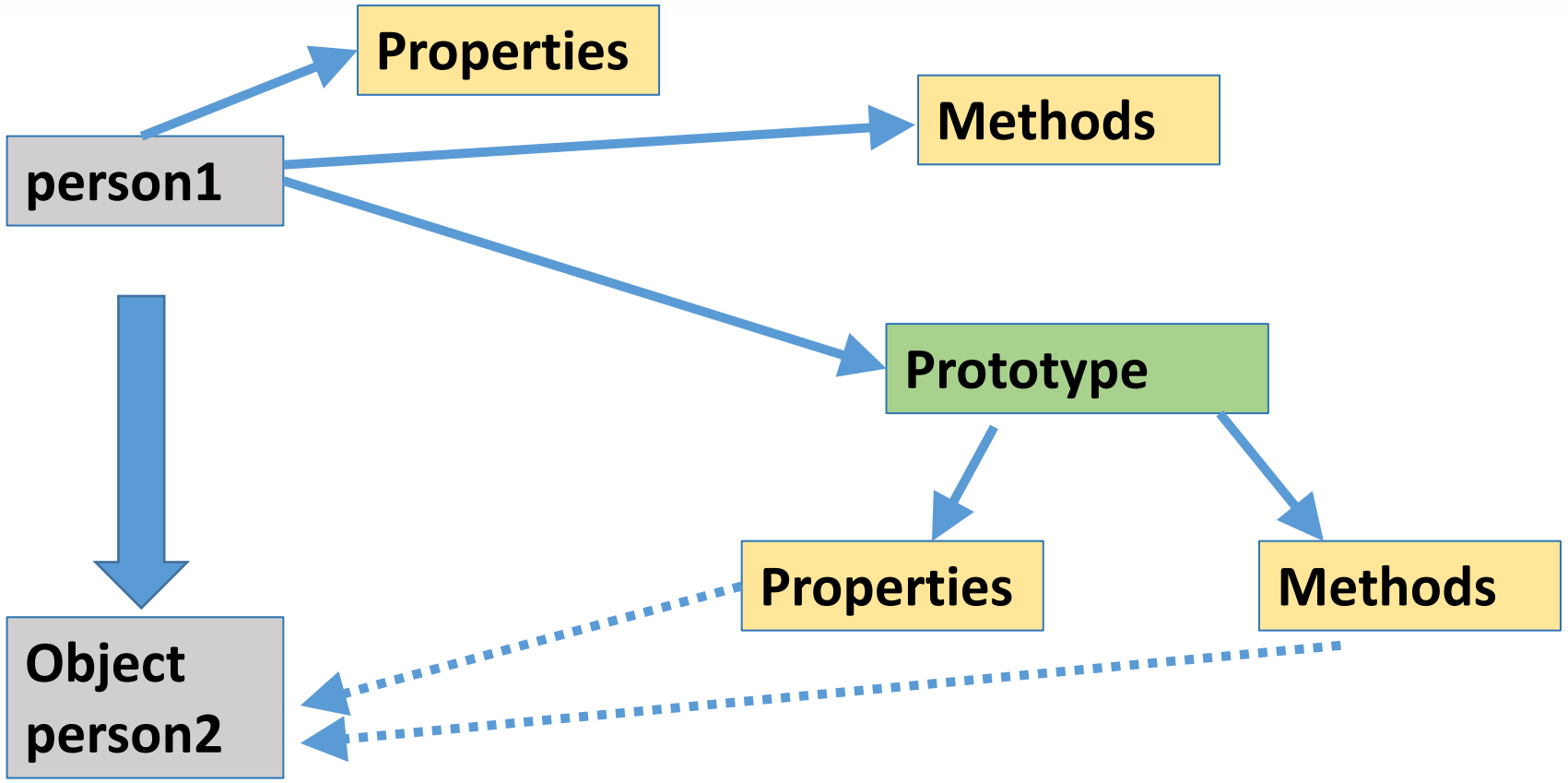
```
var PersonProto = function () {
};

PersonProto.prototype.fname = "";
PersonProto.prototype.lname = "";
PersonProto.prototype.email = "";

PersonProto.prototype.displayName = function ( ) {
        console.log(`${this.fname} ${this.lname}`);
};
var person1 = new PersonProto ();
person1.fname = "Karthik";
person1.lname = "Raman";
person1.displayName();
```

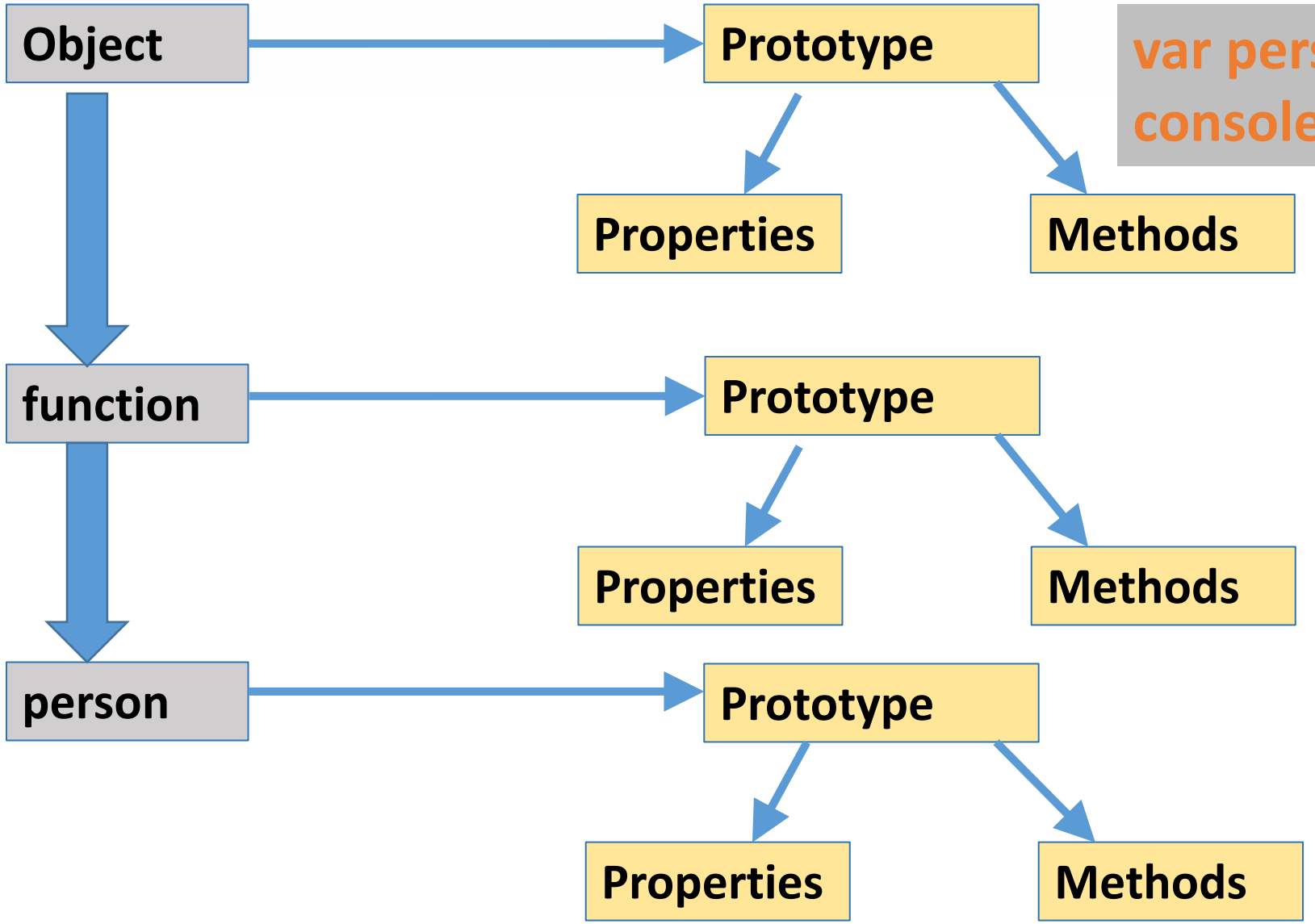
Dynamic Prototype Pattern

ObjectDynProto.html

```
var PersonDynProto = function (fname,lname,email) {
this.fname = fname;
this.lname = lname;
this.email = email;
if (typeof(this.displayName !== 'function')) {
        PersonDynProto.prototype.displayName = function (){
                console.log(`${this.fname} ${this.lname}`);
                };
        };
};
var person1 = new PersonDynProto ('Karthik','Raman','karthik@clayfin.com' );
person1.fname = "Karthik";
person1.lname = "Raman";
person1.displayName();
```

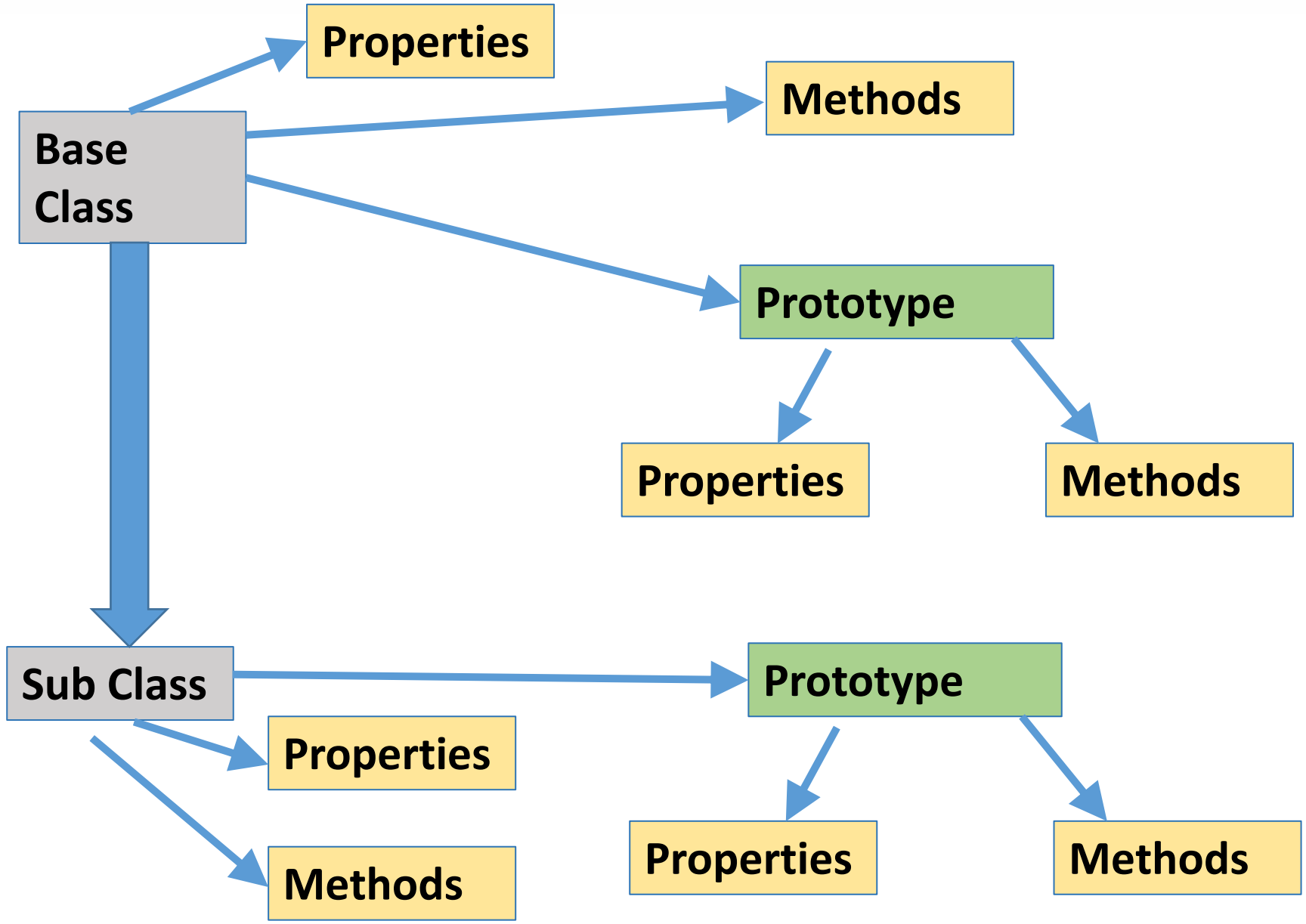
Prototype based Inheritance






```
var person = function  () { };
console.dir(person);
```

SubClass.html



Clayfin

Class.html

Javascript did not have classes till ES6.
It used Prototype based inheritance for OOPs

In ES6, Class keyword was introduced which was more of Syntactical Sugaring as the internal concept and implementation remained the same as ES5

Subclass can be created using extend keyword

Chaining Methods, also known as Cascading, refers to repeatedly calling one **method** after another on an object, in one continuous line of code

```
Object . Method1 ( ) . Method2 ( ) . Method3 ( );
```

Chaining Methods can be done only if all the methods are Chainable by returning the current object

MethodChaning.html

Encapsulation can be done in javascript by the use of var instead of this for the properties and methods in the object

Encapsulation.html

```
var  sum  = function ( a  ,  b ) {
        return a + b ;
};
```

ES5

```
let  sum  = ( a  ,  b ) =>  return (a + b) ;
```

ES6

Does not have it own “this” and uses Parent “this”
Cleaner alternative for that = this pattern

Arrow.html

Functions can be passed as Arguments for another Functions

Functions can return Functions as a return Value

FunctionsAsArgandReturn.html

setTimeout() method calls a function or evaluates an expression after a specified number of milliseconds.
The function is only executed once.
If you need to repeat execution, use the **setInterval**() method.
Use **clearTimeout**() method to prevent the function from running

TimerId = setTimeout(*function, milliseconds, param1, param2, ...*)

TimerId = setInterval(function*, milliseconds, param1, param2, ...*)

clearTimeout(TimerId)

SetTimeout.html

Spread operator {...} allows an iterable to expand in places where 0+ arguments are expected. It is mostly used in variable array where there is more than 1 values are expected.It allows us the privilege to obtain a list of parameters from an array

```
var myfunc = function ( ...n ) {
        console.log(n);
};
myfunc(1,2,3);
myfunc(1,2,3,4,5);
```

Spread.html

call, **apply**, and **bind** methods allows to borrow methods from other objects (can enable multiple inheritance)

call, **apply**, and **bind** methods to set the **this** keyword independent of how the function is called. This is especially useful for the callbacks

call and **apply** call the function immediately
Time of binding immediate and Time of Execution immediate

bind returns a function that, when later executed
Time of binding immediate and Time of Execution Future

```
var obj = {num:2};

var functionName = function(arg1, arg2, arg3){

};


functionName.call(obj, arg1, arg2, arg3 );

functionName.apply(obj [arg1, arg2, arg3]);

var bound = functionName.bind(obj);
bound(arg1, arg2, arg3);
```

Call-Apply-Bind.html

- A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return.
- The yield statement suspends function's execution and sends a value back to caller, but retains enough state to enable function to resume where it is left off.
- When resumed, the function continues execution immediately after the last yield run

```
function * fun()   // Generate Function
{
   yield 10;
   yield 20;
   yield 30;
}

// Calling the Generate Function
var gen = fun();
console.log(gen.next().value);
console.log(gen.next().value);
console.log(gen.next().value);
```

```
function * nextNatural() // Generate Function
{
   var naturalNumber = 1;
   while (true) {   // Infinite Generation
      yield naturalNumber++;
   }
}

var gen = nextNatural(); // Calling the Generate Function

// Loop to print the first 10 Generated number
for (var i = 0; i < 10; i++) {

   console.log(gen.next().value); // Generating Next Number

}
```

Closures

- It is an implicit permanent link between the function and its scope chain
- A function definition's (Lamda) hidden [[scope]] ("[[]]" denotes internal property) reference:
  - Holds the Scope Chain (preventing garbage Collection)
  - It is used and copied as the "outer environment reference" anytime the function is run

```
var add = (function () {
    var counter = 0;
    return function () {return counter += 1;}
})();
```

Closures – functions with preserved data

Closure.html

Async in Javascript is about two activities, where one activity triggers another activity, which will be completed in future
Ex: setTimeout or registering for a click event and waiting for the user to click

```
function showSessionExpire(){
        console.log("Your session expired");
}

setTimeout(showSessionExpire, 3000);

console.log("showSessionExpire will execute after 3000 ms");
```

Currying is a fundamental tool in functional programming, a programming pattern that tries to minimize the number of changes to a program's state (known as *side effects*) by using *immutable* data and *pure* (no side effects) functions

Currying is the process of taking a function with *multiple* arguments and returning a series of functions that take *one* argument and eventually resolve to a value

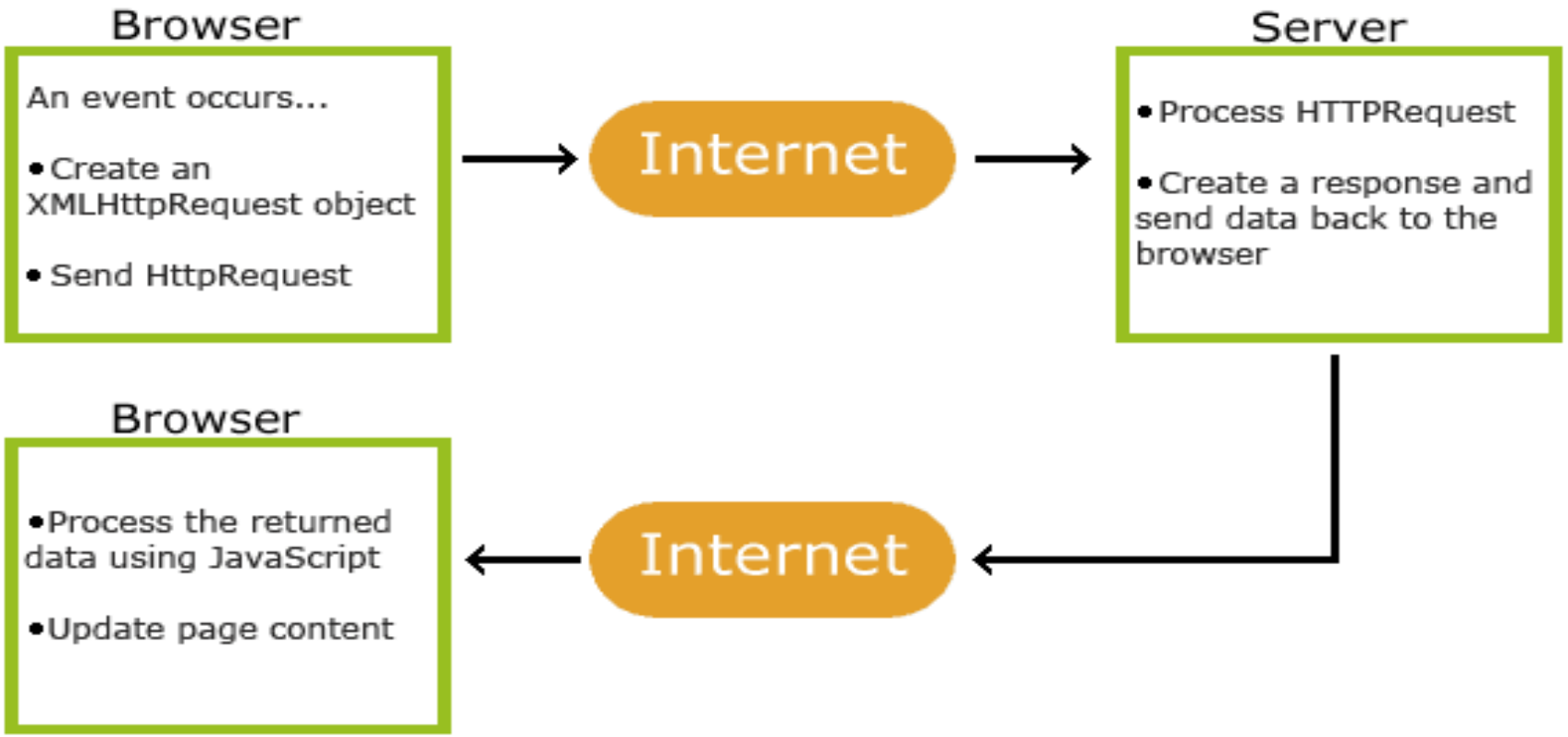
Currying.html

Asynchronous JavaScript and XML

Read data from server

Update a web page without reloading the page

Send data to server - in the background (asynchronous)



```
var xhttp = new XMLHttpRequest();
 xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) { return  this.responseText; }
 };
 xhttp.open("GET", "ajax_info.txt", true);
 xhttp.send();
```

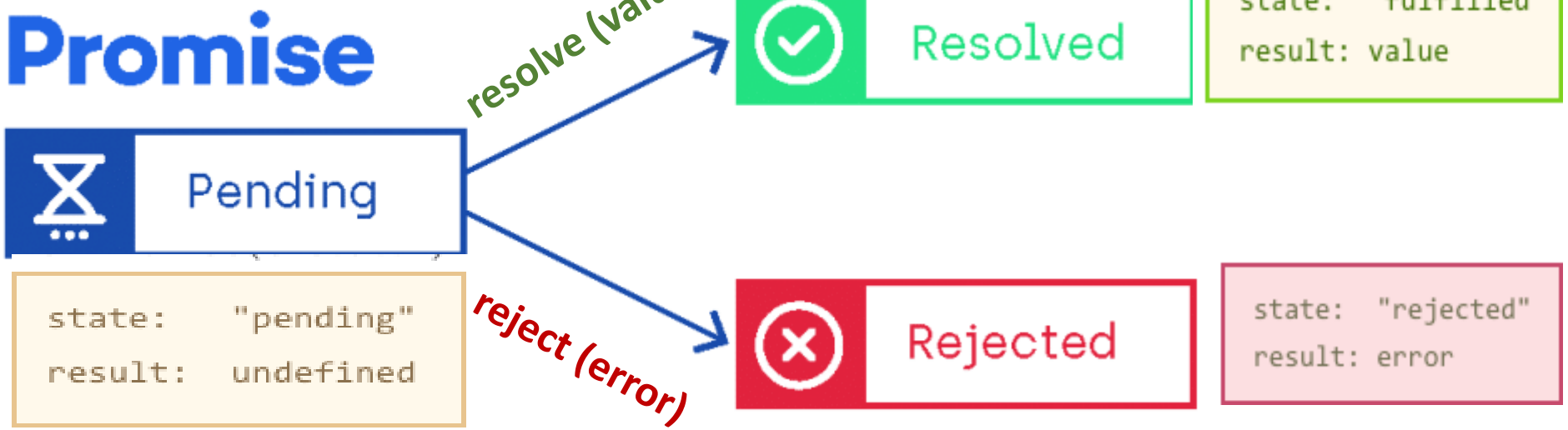
onreadystatechange	Defines a function to be called when the readyState property changes
readyState	Holds the status of the XMLHttpRequest.
	0: request not initialized
	1: server connection established
	2: request received
	3: processing request
	4: request finished and response is ready
status	200: "OK"
	403: "Forbidden"
	404: "Page not found"

```
function AJAXCall(methodType, url, callback){
 var xhr = new XMLHttpRequest();
 xhr.open(methodType, url, true);
 xhr.onreadystatechange = function(){
      if (xhr.readyState === 4 && xhr.state === 200){
            callback(xhr.response);
      }
  }
  xhr.send();
 console.log("request sent to the server");
}

var url = "https://api.clayfin.com/users";

function ProcessUserData(data){
  //Process User Details Data
}

AJAXCall("GET", url, ProcessUserData);
```

A Promise object represents a value that may not be available yet, but will be resolved or rejected at some point in the future

Promises are used to handle asynchronous operations in JavaScript. They are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code

New Promise(executor)



Pending: not yet fulfilled or rejected
Fulfilled: Promise resolved successfully
Rejected: Promise rejected

Promise.html

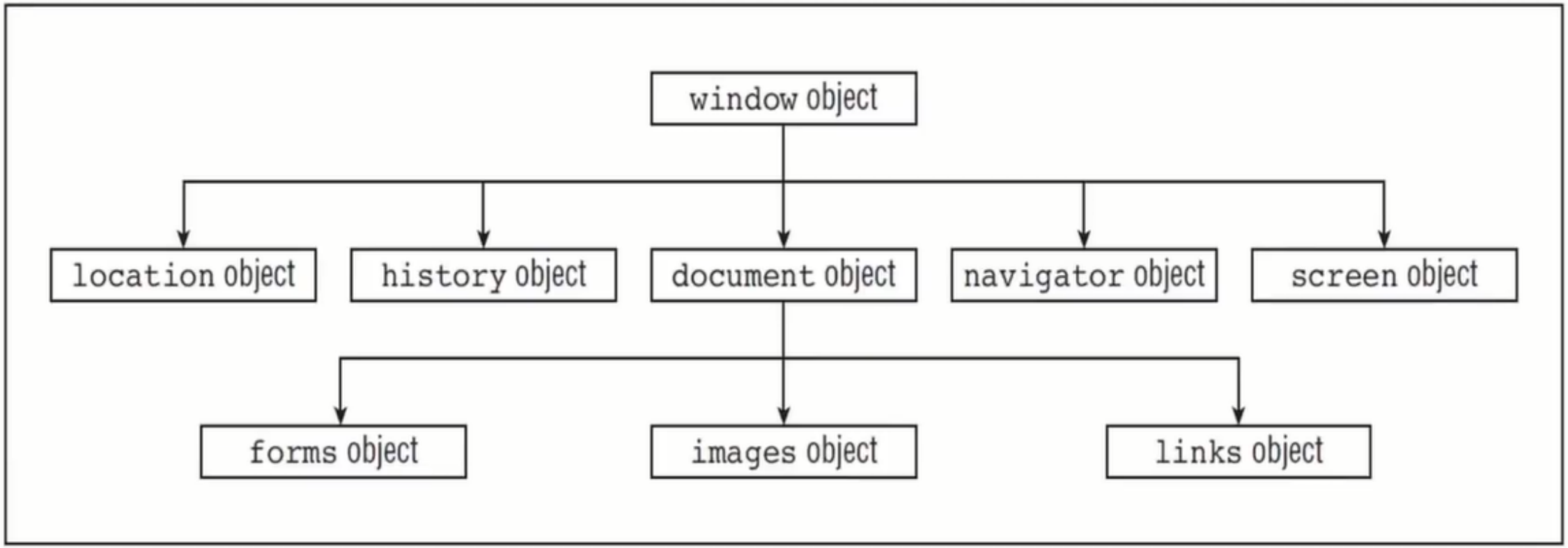
```
const promise = new Promise((resolve, reject) => {
  const request = new XMLHttpRequest();

  request.open('GET', 'https://api.clayfin.com/users');
  request.onload = () => {
    if (request.status === 200) {
      resolve(request.response); // we got data here, so resolve the Promise
    } else {
      reject(Error(request.statusText)); // status is not 200 OK, so reject
    }
  };
  request.onerror = () => {
    reject(Error('Error fetching data.')); // error occurred, reject the  Promise
  };
  request.send(); // send the request
});

console.log('Asynchronous request made.');

promise.then((data) => {
  console.log('Got data! Promise fulfilled.');
  console.log(JSON.parse(data));
}, (error) => {
  console.log('Promise rejected.');
  console.log(error.message);
});
```

```
function request(url) {
    return new Promise( function(resolve,reject){
        makeAjaxCall( url, function(err,text){
            if (err) reject( err );
            else resolve( text );
        } );
    } );
};
```

```
function *generator(){
  yield request('url1');
  yield request('url2');
};
```

Browser Object Model (BOM)

- Browser Objects are global objects
- Available throughout the JavaScript code
- BOM allows JavaScript code to interact with the browser properties
- The image below shows different BOM objects :



BOM Window Object

- Represents the browser's window
- Global variables are properties of the window object.
- Global functions are methods of the window object.
- Even the document object (of the HTML DOM) is a property of the window object
  - window.open() - open a new window
  - window.close() - close the current window
  - window.moveTo() -move the current window
  - window.resizeTo() -resize the current window

BOM – Screen Object

- The Screen object contains information of the users screen
- The window.screen object can be written without the Window prefix
- Some basic properties of the screen are:

screen.width	•Returns the width of the users screen
screen.height	• Returns the height of the users
screen.availWidth	• Returns the width of the users screen, excluding features like Windows Taskbar
screen.availHeight	• Returns the width of the users screen, excluding features like Windows Taskbar
screen.colorDepth	• Returns the number of bits used to display one color
screen.pixelDepth	• Returns the pixel depth of the screen.

BOM – Navigator Object

- The Navigator object contains information of the visitors browser
- The window.navigator object can be written without the window prefix

navigator.appName

- Returns the application name on the browser

navigator.appCodeName

- Returns the application codename on the browser

navigator.platform

- Returns the platform(Operating System) name

navigator.cookieEnabled

- Returns true if cookies are enabled

navigator.product

- Returns the product name of the browser engine

BOM Location Object

The window.location object can be used to:

- Get the current page address (URL)
- Redirect the browser to a new page.

window.location.href

- Returns the href (URL) of the current page

window.location.hostname

- Returns the domain name of the web host

window.location.pathname

- Returns the path and filename of the current page

window.location.protocol

- Returns the web protocol used (http: or https:)

window.location.assign

- Loads a new document

BOM History Object

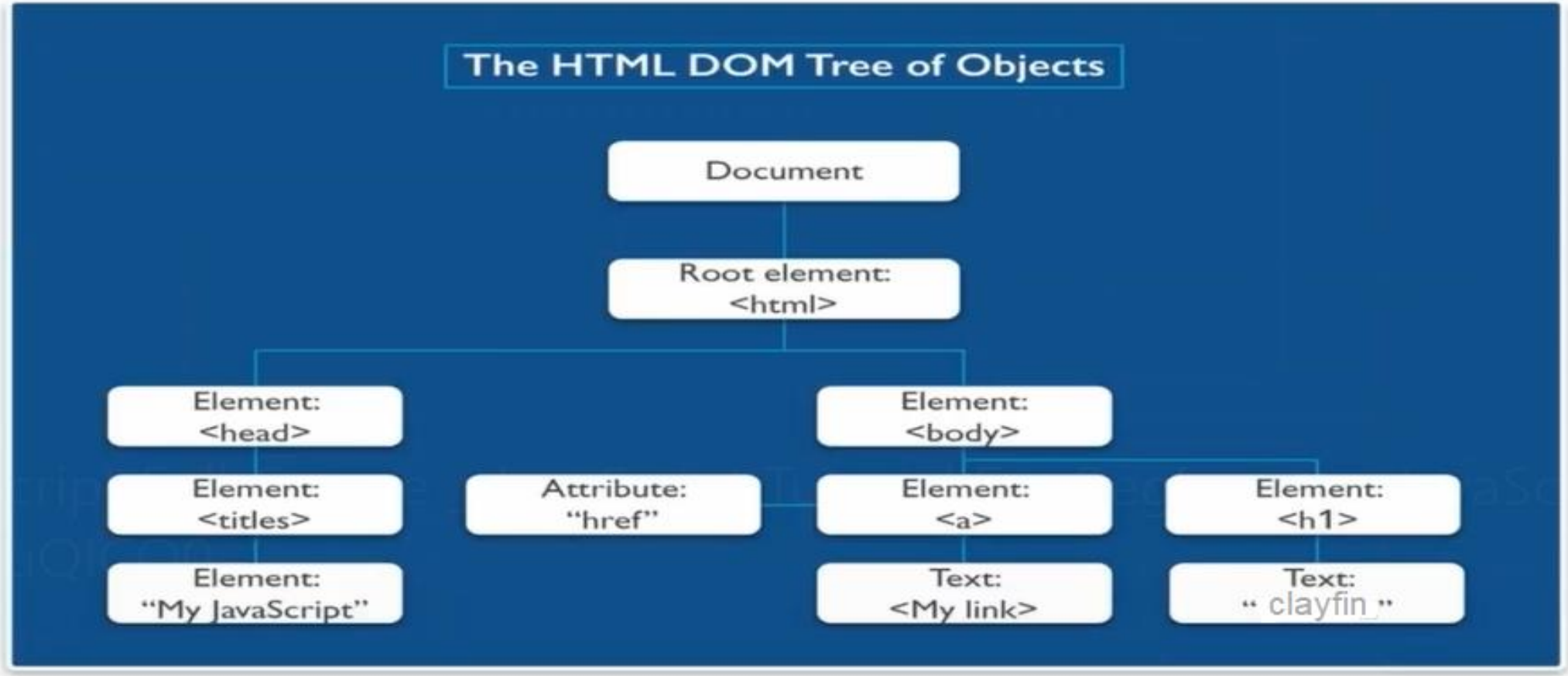
- The window.history object contains the browsers history
- Some methods of history object :
  - history.back() - same as clicking back in the browser
  - history.forward() - same as clicking forward in the brow

Document Object Model (DOM)

- The HTML DOM is a standard **object** model and **programming interface** for HTML.
- It defines:
  - The HTML elements as **objects**
  - The **properties** of all HTML elements
  - The **methods** to access all HTML elements
  - The **events** for all HTML elements
- **D**ocument **O**bject **M**odel of the page is created by the browser, when a webpage is loaded

Document Object Model

- The HTML DOM model is constructed as a tree of Objects:



What can JavaScript do with DOM?

- JavaScript + DOM = Dynamic HTML on Client-Side

Change all the HTML elements and attributes in the page

Change all the CSS styles in the page

Add and Remove existing HTML elements and attributes

JavaScript can react to all existing HTML events in the page

JavaScript can create new HTML events in the page

Document Object Model – Finding Elements

- The document object represents your web page
- To access any HTML element, you will start with accessing the document object and DOM methods to find them

getElementById() Method	getElementsByClassName() Method	getElementsByTagName() method
Gets the HTML Element with specified ID	Gets the HTML Elements with specified class name	Gets the HTML Elements with specified element tag

```
// Single element
document.getElementById("myID");
document.QuerySelector(".myclass");
document.QuerySelector("#myID");
document.QuerySelector("div");

//Multiple elements
document.QuerySelectorAll(".myclass");
document.getElementsByClassName("myclass");
document.getElementsByTagName("div");
const items = document.QuerySelectorAll(".myclass");
items.forEach((item) => console.log(item));

const ul = document.QuesrySelector(".myul");

ul.remove();
ul.lastElementChild.remove();
ul.firstElementChild.textContent = "Hi";
ul.children[1].innerText = “Karthik";
ul.lastElementChild.innerHTML="<h4>Hello</h4>";
```

Document Object Model – Events

Events are "things that happen"
Example of events are:

- When a user clicks the mouse
- When a web page has loaded
- When an image has been loaded
- When the mouse moves over an element
- When an input field is changed
- When an HTML form is submitted
- When a user strokes a key

- JavaScript has the ability to react on these events

Document Object Model – Events

Document Object Model – Events

- These Events can be accessed by adding them as Event Listeners to Elements

Mouse Events

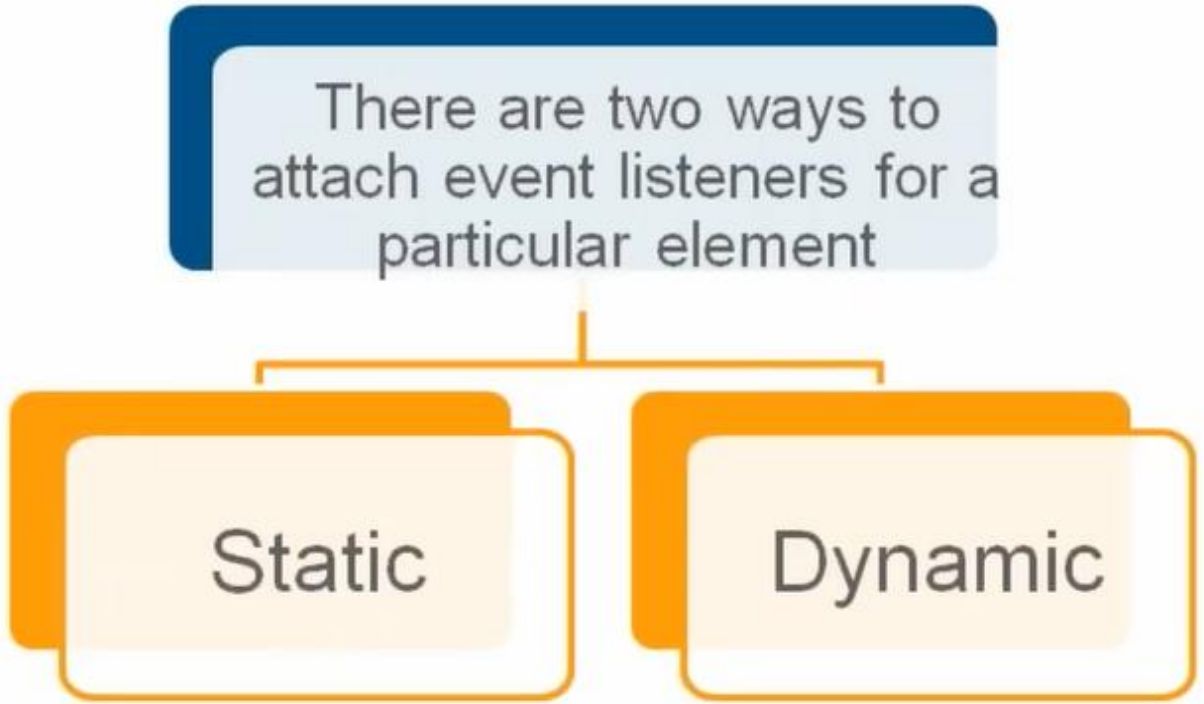
- mousedown
- mouseup
- click
- dblclick
- mousemove
- mouseover
- mousewheel
- mouseout
- contextmenu

Touch Events

- touchstart
- touchmove,
- touchend,
- touchcancel

Document Object Model – Event Listeners

- Event Listeners are: The type of event that is occurring
- We can attach any number of event listeners to an element



Document Object Model – Event Listeners

- Attach Event Handlers as HTML Element Attributes, Statically
  - Syntax : <element event_listener1=" function_name1(); " event_listener2=" function_name2(); " >
  - Example : Event "click" will occur, when we click on the element. It will call the function paraClicked(), which will change the CSS properties of the element

```
<p id="myPara" onclick="paraClicked();" style="color: green;">
<script>
 function paraClicked() {
document.getElementById("myPara").setAttribute("style", "color:
green"); }
</script>
```

Document Object Model – Event Listeners

- Attach Event Listeners with addEventListener method, dynamically
- Syntax :

  element.addEventListener(event, function);
- Example : Event "click" will occur and call the function paraClicked(), which will change the CSS properties of the element

```
<script>
document.getElementById("myPara").addEventListener("click",
paraClicked);
function paraClicked()
{
 document.getElementById("myPara").setAttribute("style", "color:
cyan");
}
</script>
```

```
const btn = document.QuerySelector(".btn");
btn.style.background = 'red';

btn.addEventListener("click",(e) => {
        e.preventDefault();
        console.log(e.target);
        console.log(e.target.className);
        console.log(e.target.id);

        });
```

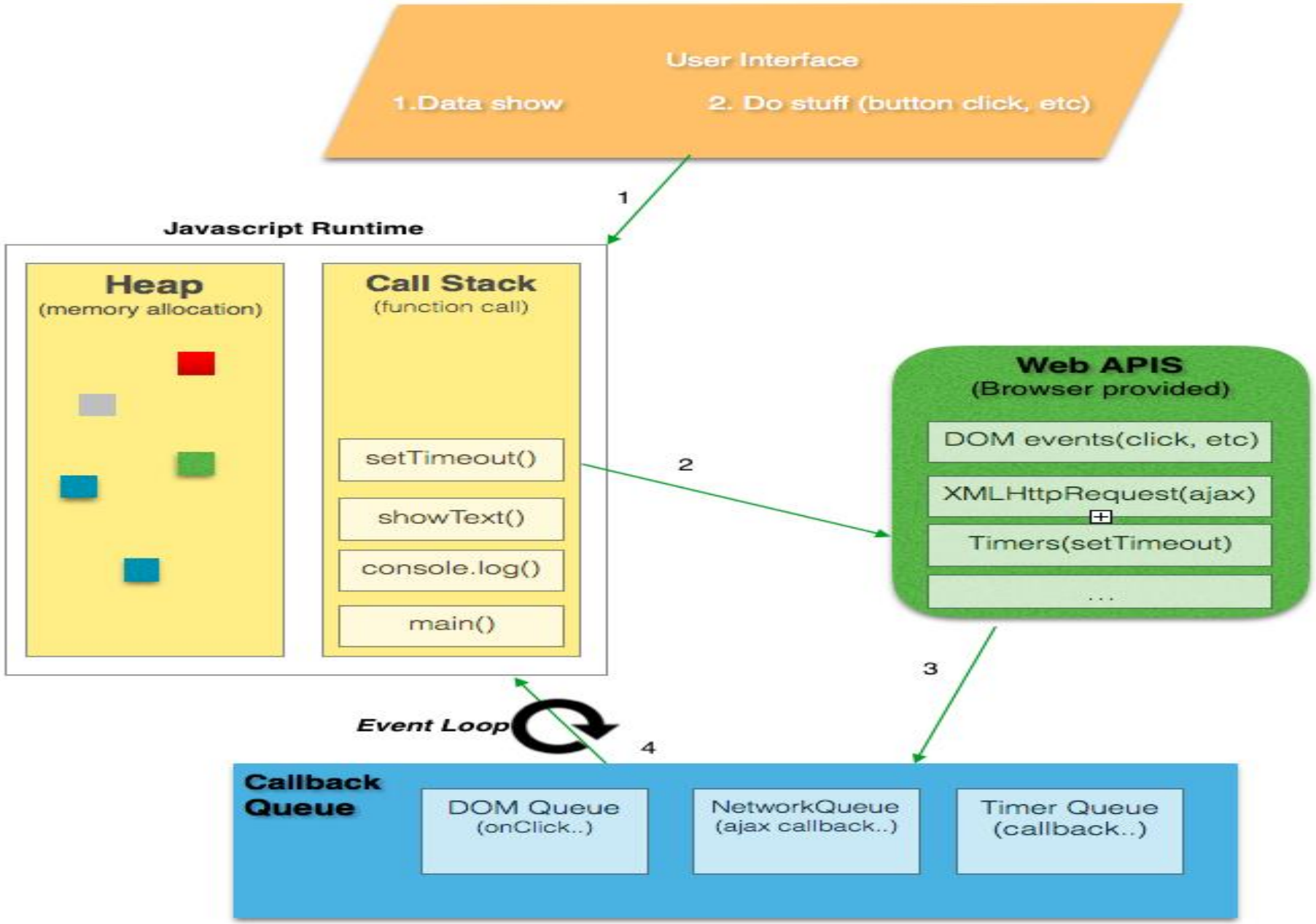
Form Validations

- HTML forms can be validated by JavaScript
- The main goal of Data Validation is to ensure correct user input
- Validation can be defined by many different methods, and deployed in many different ways :
  - **Server-side validation** is performed by a web server, after input has been sent to the server
  - **Client-side validation** is performed by a web browser, before input is sent to a web server
- Saves a lot of unnecessary calls to the server as all processing is handled by the web browser

Typical Validations

- Check if the input field is empty
- Check if the input field is a numeric value
- Check if the input field is a valid email id

Execution Model



Thank You

www.clayfin.com