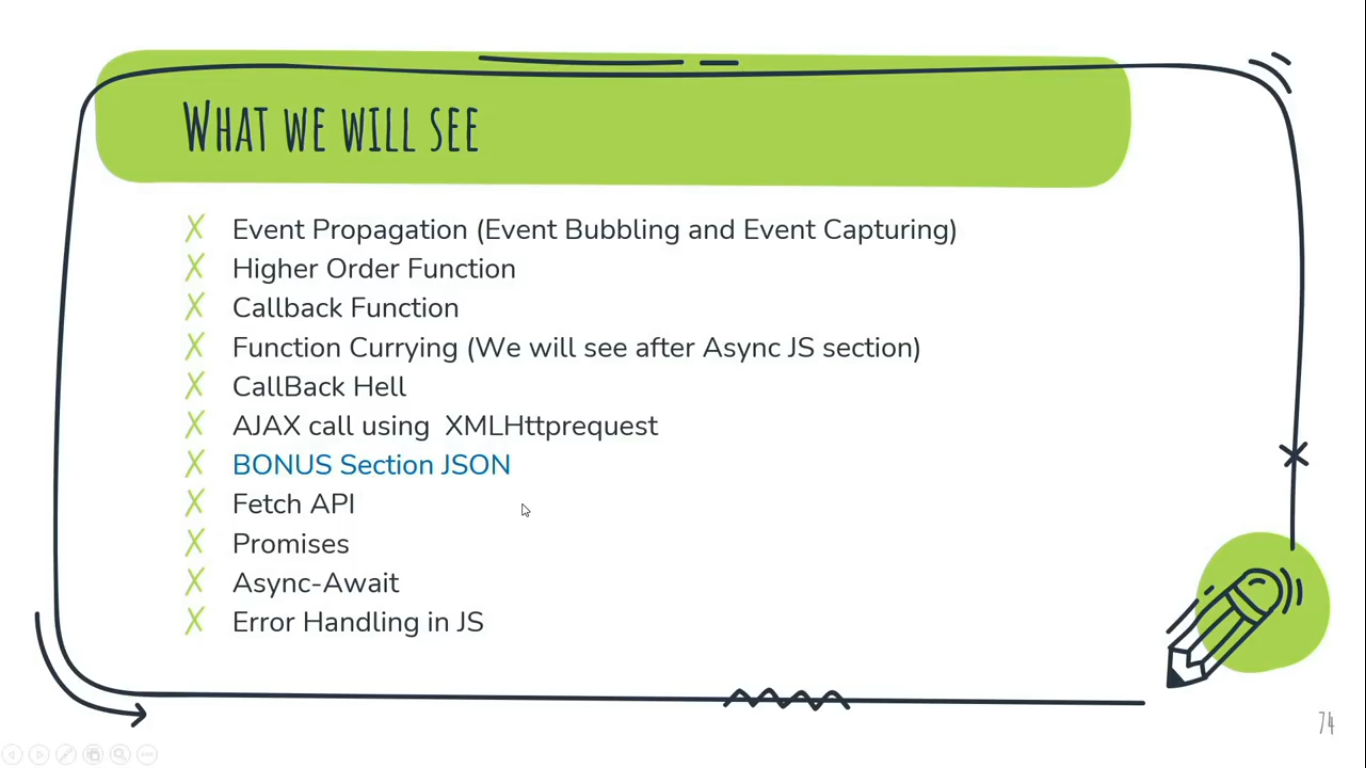
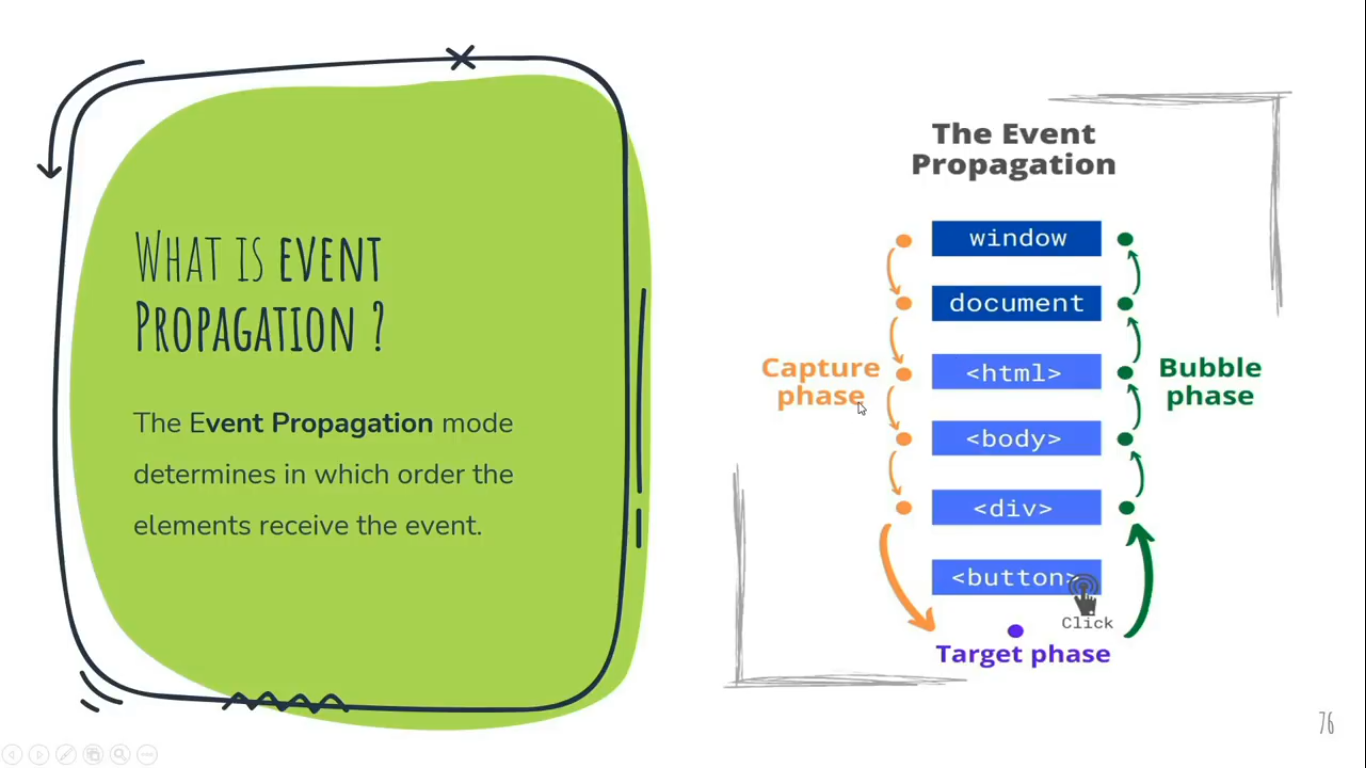
**Advance JavaScript**



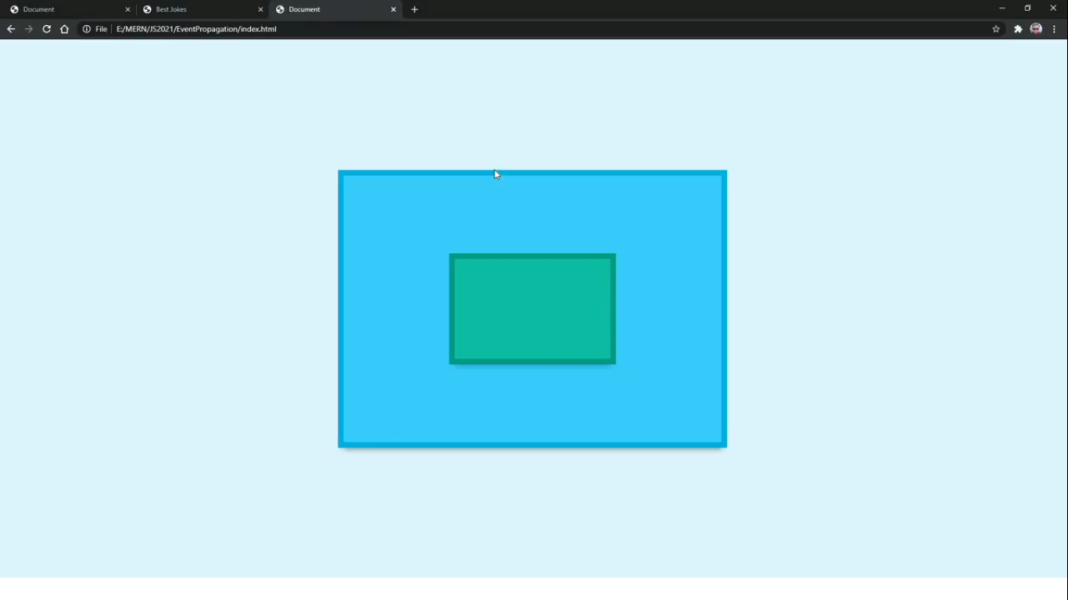
**Event Propagation (Event Bubbling & Event Capturing):-**

The Event Propagation mode determines in which order the elements receive the event.



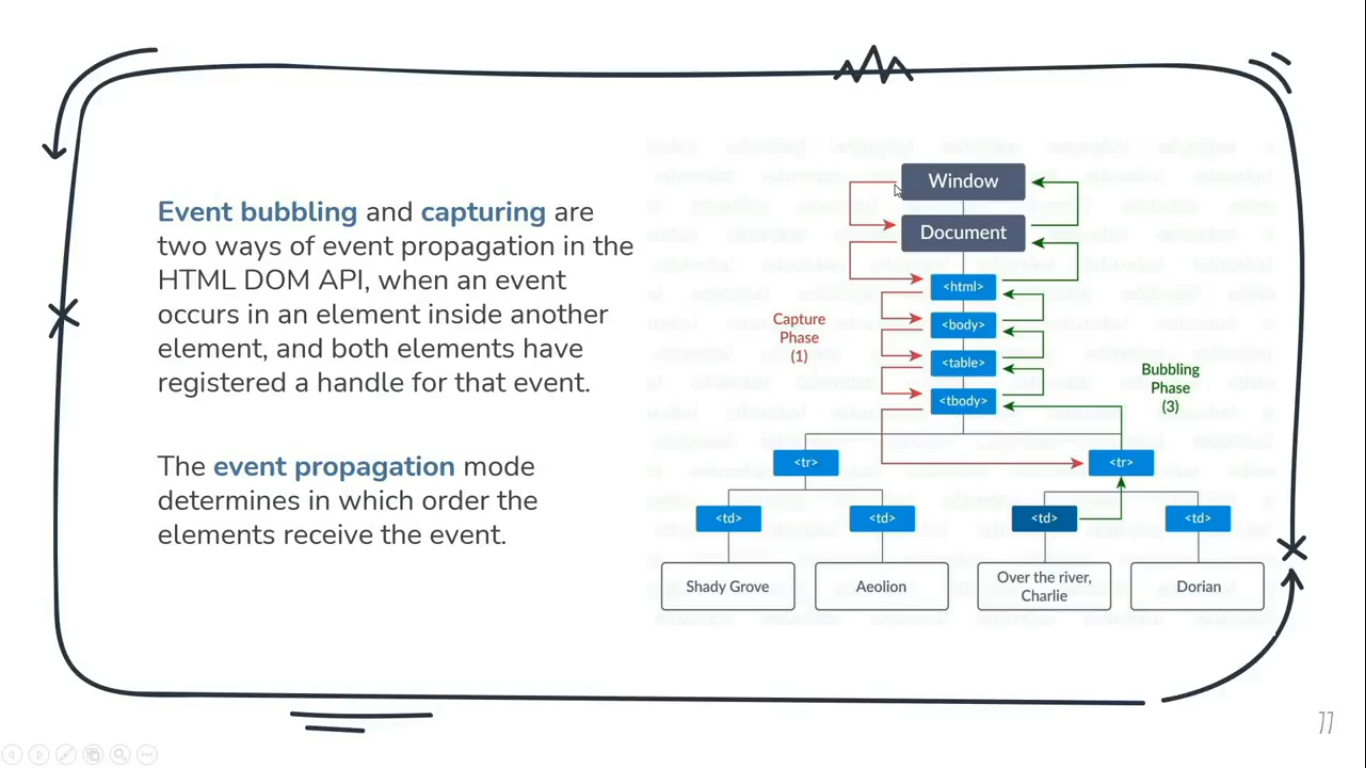
Let say, we have ‘2 nested div’ and both are acting as a ‘button’, when click on each firing an event of ‘alert box’ instantly one after another for their respective div.

Consider, a child div nested under a parent div at the center of the document.



In the first case, When click at the center of the parent div, It fire an event of alert box calling itself “I’m a Parent Div” instantly after firing an alert box calling itself “I’m a Child Div”. This case what called ‘**Bubble Phase**’.

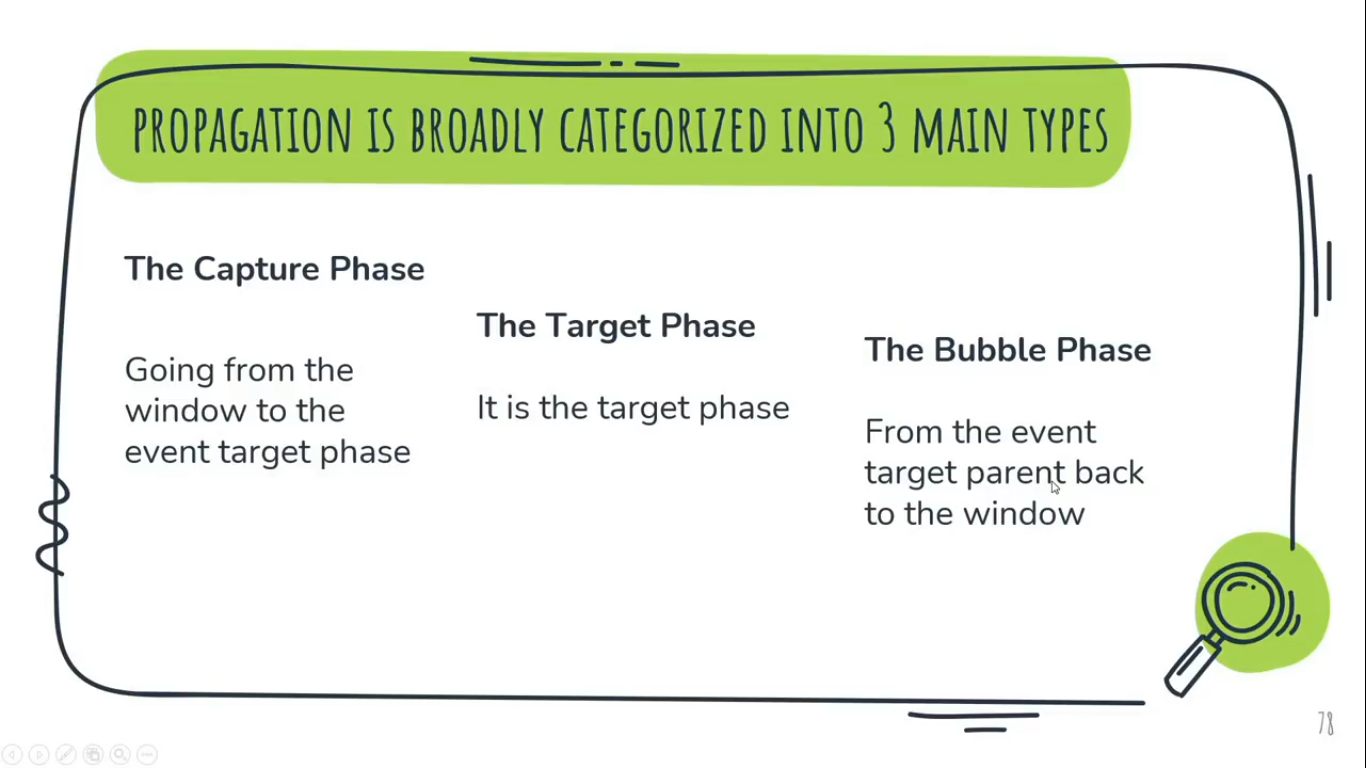
In the second case, When click at the center of the parent div, It fire an event of alert box calling itself “I’m a Child Div” instantly after firing an alert box calling itself “I’m a Parent Div”. This case what called ‘**Capture Phase**’.



**Event Bubbling:**With Event Bubbling, the event is first captured and handlred by the innermost element and then propagated to outer elements.

**Event Capturing:**With Event Capturing, the event is first captured by the outermost element and propagated to the inner elemetns.  
Capturing is also called “trickling”, whick helps remember the propagation order.

Propagation is broadly categorized into 3 main types:



Note:

* Here, **Event Bubbling** and **Event Capturing** will pass as ***false*** and ***true***respectively through the 3rd parameter of addEventListener() method.
* By Default, Event Propagation in JavaScript or say the 3rd parameter of addEventListiner is **Event Bubbling** i.e.( Bottom to Top) and its value is **false**.  
  Sample: addEventListiner(‘event’, function, false).
* If we don’t want to propagate any Event propagation i.e. (instant propagation of Event one after another), We’ll use **event.stopPropagation();** method just after one event has successfully fired.  
  **Example:**<div class="parent-div"> **//Don’t forget to styling**  
   <div class="child-div">  
   </div>  
  </div>

const Parent = document.querySelector('.parent-div');  
const Child = document.querySelector('.child-div');

Parent.addEventListener('click', ()=>{

alert('Hi!, This is Parent Div');

});

Child.addEventListener('click', ()=>{

alert('Hi!, This is Child Div');

**event.stopPropagation();**

});

**Example-1: //By default Event Propagation i.e. Event Bubbling (false)**<div class="parent-div"> **//Don’t forget to styling**  
 <div class="child-div">  
 </div>  
</div>

const Parent = document.querySelector('.parent-div');  
const Child = document.querySelector('.child-div');

Parent.addEventListener('click', ()=>{

alert('Hi!, This is Parent Div');

}, **false**); **//By Default is Event Bubbling which is fasle, So we can avoid it freely.**

Child.addEventListener('click', ()=>{

alert('Hi!, This is Child Div');

}, **false**); **//By Default is Event Bubbling which is fasle, So we can avoid it freely.**

**Example-2: //By Event Capturing i.e. (true)**

<div class="parent-div"> **//Don’t forget to styling**  
 <div class="child-div">  
 </div>  
</div>const Parent = document.querySelector('.parent-div');  
const Child = document.querySelector('.child-div');

Parent.addEventListener('click', ()=>{

alert('Hi!, This is Parent Div');

}, **true**); **//true indicating Event Capturing**

Child.addEventListener('click', ()=>{

alert('Hi!, This is Child Div');

}, **true**); **// true indicating Event Capturing**

**Higher Order Function & CallBack Funciton:-**

**Higher Order Function:**Function which Takes or Accept or Hire another function as an Arguments is called Higher Order Function.

**CallBack Function:**Function which get Passed an an Argument to another function is called CallBack Function.  
 OR  
A callback function is a function that is passed as an argument to another function, to be ‘’called back” at a later time.

**Example: // Build a Calculator**const add = (a, b)=>{   
**// add is a constant variable assigning an anonumous fn by defining it to make it as a function.**  
 return a + b ;  
}  
const substract = (a, b)=>{  
 return a - b ;  
}  
const multiply = (a, b)=>{  
 return a \* b ;  
}

const calc = (num1, num2, operator)=>{ **//Higher Order Function**

return operator(num1, num2); **//working like this example: add(5,4);**

}

console.log(calc(5,4,add)); **// CallBack Fn ‘add’ going to workas an Operator**  
console.log(calc(5,4,substract)); **// CallBack Fn ‘sub..’ going to workas an Operator**  
console.log(calc(5,4,multiply)); **// CallBack Fn ‘mul..’ going to workas an Operator**

**Output:**9  
1  
20

**Here,** *add ,substract & multiply* are the Callback function function because these are passing as an arrgument to the *calc* function.   
And *calc* function is the Higher Order Function because it is accepting functions like (*add, subsctract & multiply*) as an Argument.

**Function Currying:-**

Function Currying is a technique of evaluating a funciton with *multiple arguments,* into *sequence of function* with *single argument*.

In other words, when a funciton, instead of taking all arguments at one time, takes the first one and return a new function that takes the second one and returns a new function which takes the third one, and so forth, untill all arguments have been fulfilled.

That is, when we turn a function call add(1,2,3) into add(1)(2)(3) . By using this technique, the little piece can be configured and reused with ease.

**Example: (without Function Currying )**function sum(num1, num2, num3){  
 console.log(num1+num2+num3);  
}

sum(1,2,3); **//Addition of 1, 2 and 3**

**Output:**6 **//Result of 1 + 2 + 3**

**Example: (with Function Currying )**function sum(num1){ **//1st function with 1st parameter**

return (num2)=>{ **//accepting 1st parameter num1 and returning 2nd function**

return(num3)=>{ **//accepting 2nd parameter num2 and returning 3rd function**  
 console.log(num1+num2+num3);  
 }

}

}

sum(1)(2)(3); **//Addition of 1, 2 and 3**

**Output:**6 **//Result of 1 + 2 + 3**

**Note:  
*We can also write the above Example of Funciton Currying in one Single Line.***

sum = (num1)=>(num2)=>(num3)=> console.log(num1+num2+num3);

sum(1)(2)(3);

**Output:**6

**CallBack Hell:-**

As we already know that, a function called a CallBack function when it passes as an argument through another function or Call another function i.e. (CallBack) inside the same funtion after some interval of time.

So, Calling nested function after some interval of time called callback Hell.

**Example:**setTimeout(()=>{  
 console.log('1st Work has Done');  
 setTimeout(()=>{  
 console.log('2nd Work has Done');  
 setTimeout(()=>{  
 console.log('3rd Work has Done');  
 setTimeout(()=>{  
 console.log('4th Work has Done');  
 setTimeout(()=>{  
 console.log('5th Work has Done');  
 },1000);  
 },1000);  
 },1000);  
 },1000);  
},1000);

**Output:**1st Work has Done  
2nd Work has Done  
3rd Work has Done  
4th Work has Done  
5th Work has Done

**AJAX Call using *XMLHttpRequest(*XHR*)*:-**

**AJAX,** which stands for Asynchronous JavaScript and XML, is a programming practice of building complex, dynamic webpages using a technology known as XMLHttpRequest (XHR).

Ajax allows you to update parts of the DOM of an HTML page instead without the need for a full page refresh. Ajax also lets you work asynchronously, meaning your code continues to run while the targeted part of your web page is trying to reload (compared to synchronously, which blocks your code from running untill that part of your page is done reloading).

**XMLHttpsRrequest** (XHR) objects are used to interact with servers. You can retrieve data from a URL without having to do a full page refresh. This enables a Web Page to update just part ofa page without disrupting what the user is doing. XMLHttpRequest is used heavily in AJAX programming.

|  |  |  |
| --- | --- | --- |
| Event Target 🡨 | XMLHttpRequestEventTarget | 🡨XMLHttpRequest |

Despite its name, XMLHttpRequest can be used to retrieve any type of data, not just XML.

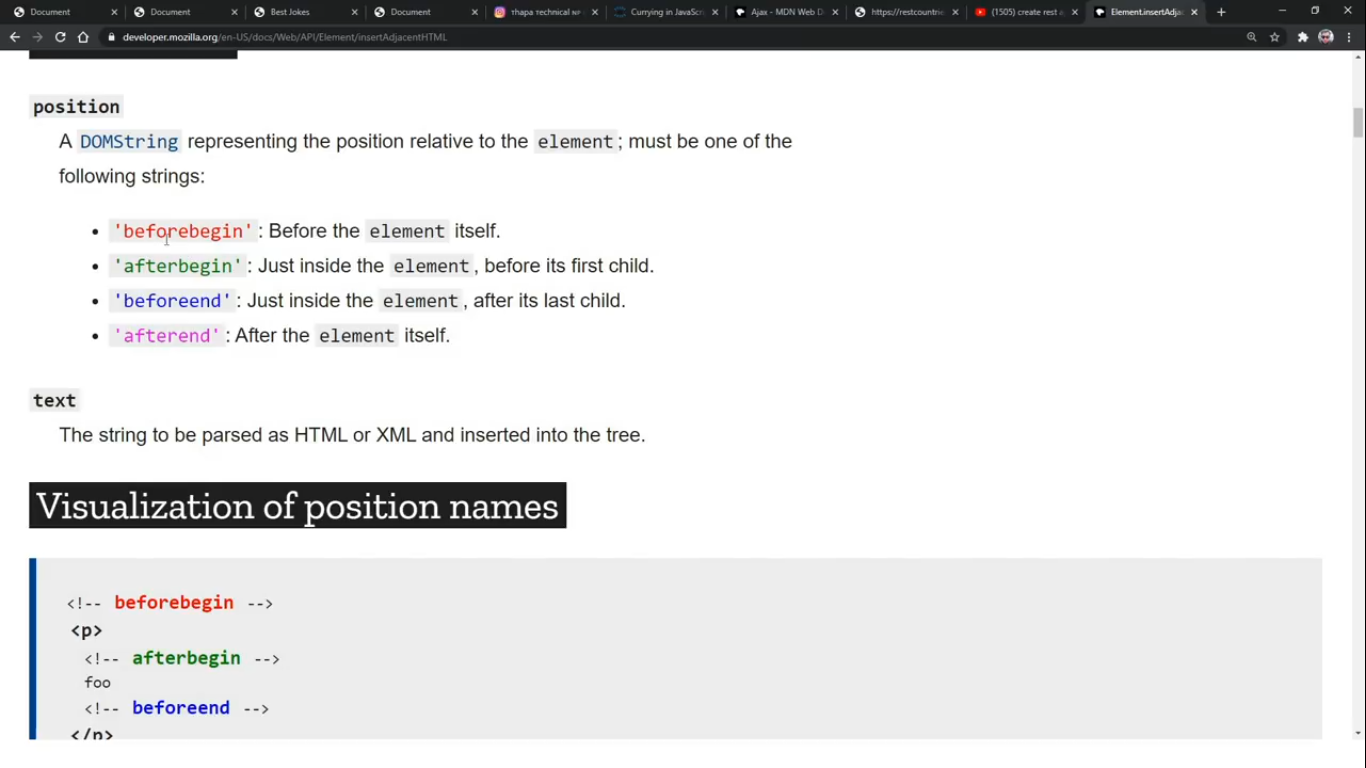
**Example: (Weather Detector Web Application)**

[*file:///E:/xampp/htdocs/websites/Learning/JavaScript/Tutorial/Practical/Project/API%20Based/Weather%20Detector/index.html*](file:///E:/xampp/htdocs/websites/Learning/JavaScript/Tutorial/Practical/Project/API%20Based/Weather%20Detector/index.html)(Open in Chrome Browser)

**Note:-**

* To call an XMLHttpRequest we have to create of its instance first.
* XMLHttpRequest is an Object.
* Open method of XMLHTTPReques Object passes two Arguments, the first argument use to get data which can be ‘POST’, ‘PATCH, ‘GET’, ‘PUT’ etc. But to read data we’ve to pass ‘GET’ argument.
* Send method of (XHR) object used to send request for requesting data which API going to deliver.
* resonseText property used to get the Data Response from the API which we’ve requested. This data would be in pure JSON format on which we cannot work as well as it cannot read by JavaScript.  
  JSON is an Object having two methods, parse and stringify.
* JSON.parse() method use to covert JSON data into JavaScript Object.

**insertAdjacentHTML() method**

**

It is used to render *html data* using JavaScript at the exact position before and after the html container by taking its reference.

This method accept 2 arguments,  
First Argument decide the *exact position* of the HTML data after or before the HTML container as ‘**beforebegin**’ or ‘**afterbegin**’ or ‘**beforeend**’, ‘**afterend**’.  
Second argumet takes the *reference* of the HTML container as **variable**(stored its access using DOM object method via class or id).

**Example:**  
container.insertAdjacentHTML('beforeend', htmlData);

**Note:**The beforebegin and afterend positions work sonly if the node is in the DOM tree and has a parent element.

**JSON (JavaScript Object Notation)**

JSON is similar to the Object which contains data in the form for Key or say Property & Value pair under curly Braces {}.  
But the major difference is that it Object consider String only to the Values(containing Characters) but not Keys or say Property whereas JASON consider String to the whole data irrespective of Key & Value. JSON consider both Key & value pair as string.

Datatype of Object is ‘Object’ where as JSON is ‘string’.

We use JSON for communicating or data transfering from One server to another in JSON Standard Format. Previously it was used in XML Stardard Format.

JSON is an Object having 2 methods,

* stringify() method
* parse() method

**stringify() method:**JSON.stringify() method turns a JavaScript object into JSON text and stores that JSON text in a String.

**Example:**const person = {

name: "Chandan",  
 age: 20,  
 present: true,

};

console.log(person); **//gives Object**  
console.log(typeof(person)); **//gives DataTypeof Object**  
console.log(JSON.stringify(person)); **// Convert Object into JSON**  
console.log(typeof(JSON.stringify(person))); **//given DataTypeof JSON**

**Output:**{name: 'Chandan', age: 20, present: true} **// Object**object **// DataTypeof Object**{"name":"Chandan","age":20,"present":true} **// Converted into JSON**  
string **// DataTypeof JSON**

**parse() method:**JSON.parse() method turns a string of JSON text into a JavaScript object.

**Example:**const person = {

name: "Chandan",  
 age: 20,  
 present: true,

};

console.log(JSON.stringify(person)); **// Convert Object into JSON**  
console.log(JSON.parse((JSON.stringify(person)))); **// Convert JSON into Object**

**Output:**{"name":"Chandan","age":20,"present":true} **// Converted into JSON**  
{name: 'Chandan', age: 20, present: true} **// Converted Again back into Object**

**Promises**

A Promise is an object that keep track about whether a certain event thas happened alreay or not.

Promises are used to handle asynchronous operations in JavaScript. They are easy to manage when dealing with multiple asynchronous operations where callbacks can cerate *callback hell* leading to unmanageable code.  
So, we can say that, promises are used to eliminate *callback hell*.

It determines what happens after the events has happened.

Promises is similar to our real life promises that we commit to complete something, and as a result outcomes is either we’re able to complete the promise or not then it consider as ‘sucess’ / ‘fulfiled’ or failed / ‘reject’.

A Promise is in one of these 3 states:

* *pending*: initial state, neither fulfilled nor rejected.  
  A pending promise can either be fulfilled with a value or rejected with a reason (error).
* *fulfilled*: meaning that the operation was completed successfully.
* *rejected*: meaning that the operation failed.

Example: Let say, I promised someone to call tomorrow exact at 10:00 a.m.. Now,

1. Before 10:00 am, promise would be in ‘**Pending**’ stage.
2. Exactly at 10:00 am i.e. (Event time), either Call (Event) will occur or say (Success or Fullfiled) or may not (Fail or Reject).  
   If event would be success then it would be consider as ‘**Resolve**’.
3. If Event doesn’t occur, it would be consider as ‘**Reject**’.

We can use Promise in two ways, either as Promise’s Constructor or Promise as an object.

**Note:**

* *Constructor Syntax*var promiseObj = new Promise(executor);   
  **// Whenever, we’re going to create new object as (promiseObj) of Promise Object using *‘new*’ keyword and its constructor i.e. *Promise()*, we always have to execute a new function called executor as callback function i.e.( Argument of Promise constructor) and then this executor() itself passes two parameter i.e. (resolve function & reject function).**resolveFun(value) **//call on fulfilled**  
  resolveFun(reason) **//call on rejected**
* A function i.e.(executor) to be exucuted by the consturctor, during the process of constructing the promiseObj.  
  The Executor is a custom code that ties an outcome to a promise.

**Promise’s Methods:**

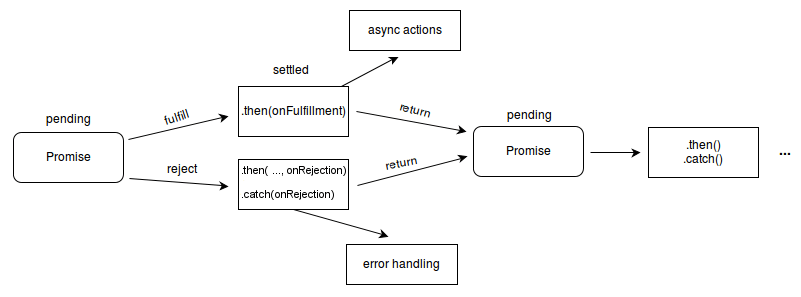
1. promise.then();
2. promise.catch();

The methods promise.then(), promise.catch(), and promise.finally() are used to associate further action with a promise that becomes settled.

As the Promise.then() and Promise.catch() methods return promises, they can be chained.

The .then() method takes up to *two arguments*; the first argument is a callback function for the *resolved case* of the promise, and the second argument is a callback function for the *rejected case*. Each .then() returns a newly generated promise object, which can optionally be used for chaining.

Processing continues to the next link of the chain even when a .then() lacks a callback function that returns a Promise object. Therefore, a chain can safely omit every rejection callback function until the final .catch().



The termination condition of a promise determines the "**settled**" state of the next promise in the chain. A "**resolved**" state indicates a successful completion of the promise, while a "**rejected**" state indicates a lack of success. The return value of each resolved promise in the chain is passed along to the next .then(), while the reason for rejection is passed along to the next rejection-handler function in the chain.

**Promise’s States**

Promises have three possible mutually exclusive states: fulfilled, rejected, and pending.

* A promise is *fulfilled* if promise.then(f) will call f "as soon as possible."
* A promise is *rejected* if promise.then(undefined, r) will call r "as soon as possible."
* A promise is *pending* if it is neither fulfilled nor rejected.

We say that a promise is *settled* if it is not pending, i.e. if it is either fulfilled or rejected.

**Promise’s Fates**

Promises have two possible mutually exclusive fates: resolved, and unresolved.

* A promise is *resolved* if trying to resolve or reject it has no effect, i.e. the promise has been "locked in" to either follow another promise, or has been fulfilled or rejected.
* A promise is *unresolved* if it is not resolved, i.e. if trying to resolve or reject it will have an impact on the promise.

A promise can be "resolved to" either a promise or thenable, in which case it will store the promise or thenable for later unwrapping; or it can be resolved to a non-promise value, in which case it is fulfilled with that value.

**Example - 1: (Promises using Constructor)**

**// Create 'new Promise Object' using its constructor which passes an argument i.e. executer as callBack function takes 2 parameter i.e. resolve & reject function.**

**//This is called Promise Produce. (Basically Producing Promises).**const promiseObject1 = new Promise((resolve, reject)=>{

setTimeout(()=>{  
 let roll\_no = [1,2,3,4,5];

***// If able to Resolve the above instruction, use resolve function***  
 resolve(roll\_no); ***//call resolve function by passing the required argument to fetch it.***

***//If unable to resolve the above instruction, use reject function***  
 reject(‘error because resolve not called');

},2000);

});

**// To show the output/result of resolve function, we use .then() & .cache() method**

**//This is called Promise Consume (Consuming Promises)**promiseObject1.then((roll\_no)=>{ ***//Passing resolve’s argument to Get it.*** console.log(roll\_no);  
}).catch((error)=>{  
 console.log(`Your Error: ${error}`);  
});

**Output:**[1, 2, 3, 4, 5] ***//When resolved successfully called, went to the then scope***  
Your Error: error because resolve not called   
***//When reject called instead of resolved, went to the catch scope***

**Example - 2: (Promises using Object)**

**// Create 'new Promise Object' using its constructor which passes an argument i.e. executer as callBack function takes 2 parameter i.e. resolve & reject function.**

**//This is called Promise Produce. (Basically Producing Promises). Promise using constructor**const promiseObject1 = new Promise((resolve, reject)=>{

setTimeout(()=>{  
 let roll\_no = [1,2,3,4,5];

***// If able to Resolve the above instruction, use resolve function***  
 resolve(roll\_no); ***//call resolve function by passing the required argument to fetch it.***

***//If unable to resolve the above instruction, use reject function***  
 reject(‘error because resolve not called');

},2000);

});

**// Promise using Object**const getBioData = (acceptRollNo)=>{ ***//(2). Getting value of Argument of getBioData while calling and Passing in acceptRollNo as parameter***

**//When we have to create new task, will always consider creating that task as new Promise** return new Promise((resolve, reject)=>{ **// setTimeout(Handler, Time in ms, Argument);**

setTimeout((acceptRollNo)=>{ ***//(4). Passing acceptRollNo as Parameter***

let bioData = {  
 Name: 'Chandan Kumar',  
 Age: 21  
 }

resolve(`My name is ${bioData.Name}, My Roll No. is ${acceptRollNo} and My Age is ${bioData.Age}.`);

}, 2000, acceptRollNo); ***//(3). Accepting acceptRollNo as argument***

});

};

**// To show the output/result of resolve function, we use .then() & .cache() method**

**//This is called Promise Consume (Consuming Promises)**promiseObject1.then((roll\_no)=>{ ***//Passing resolve’s argument to Get it.*** console.log(roll\_no);  
 return getBioData(roll\_no[0]);  ***//(1). Getting value of roll\_no (array) of index number 0 i.e. 1***

}).then((printSecondResolve)=>{ **//using .then to print second vlaue of resolve function**

console.log(printSecondResolve);

}).catch((error)=>{  
 console.log(`Your Error: ${error}`);  
});

**Output:**[1, 2, 3, 4, 5] ***//Print after 2 second***  
My name is Chandan Kumar, My Roll No. is 1 and My Age is 21. ***//Print after 2 second***

**Fetch API**

As we already discussed about calling AJAX in JS with the help of XMLHttpRequest (XHR). Now we’re going to call AJAX using Fetch API.

The Fetch API provides a fetch() method defined on the window object (means fetch Is the method of Window Object), which you can use to perform requests.

This fetch() method returns a Promise that you can use to retrieve the response of the request.

For the sake of ease, we can remind “fetch()” 🡪 ”then()”🡪 “catch()”

**Example:** <p class="fetch-api"></p>

fetch(`https://api.covid19api.com/summary`)

**//Promise consume to Get Response**  
.then((apiData)=>{

**//return keyword let Promise to execute/return another Promise** return apiData.json(); ***//Convert JSON data (Whole API Data) into Js Object***

}).then((actualData)=>{

const myCountryData = actualData.Countries[76];

document.querySelector('.fetch-api').innerHTML= `Our country ${myCountryData.Country}, currently having new Confirmed Cases are <b> ${myCountryData.NewConfirmed}</b>, New Death Cases are ${myCountryData.NewDeaths} & ${myCountryData.NewRecovered} Recovered. ${myCountryData.Country} having Total Confirmed Cases are ${myCountryData.TotalConfirmed} and Total Death Cases are ${myCountryData.TotalDeaths}`;

}).catch((error)=>{  
console.log(`The error: ${error} `);  
});

**Output: // Output print in the document**Our country India, currently having new Confirmed Cases are **179723**, New Death Cases are 146 & 0 Recovered. India having Total Confirmed Cases are 35707727 and Total Death Cases are 483936

**(Fordeep knowledge)  
Project *in Fetch API*:- *//DadJokes using both Promise & Async-Await***[*file:///E:/xampp/htdocs/websites/Learning/JavaScript/Tutorial/Practical/Project/API%20Based/DadJokes/index.html*](file:///E:/xampp/htdocs/websites/Learning/JavaScript/Tutorial/Practical/Project/API%20Based/DadJokes/index.html)(Open in Chrome Browser)

**Async-Await**

There’s a special syntax to work with promises in a more comfortable fashion, called “async / await”. It’s surprisingly easy to understand and use.

“Async / await” comes for the sake of ease of Promise Consuming in ES8.

To word “async” before a function means one simple thing: a function always retruns a promise. So, the async keyword is added to functions to tell them to return a promise rather than directly returning the value.

We can use await when calling any function that returns a Promise, including with API functions. The keyword await makes JavaScript wait until that promise settled and returns its result.  
We can only use await keyword with only async function.

**Example:**

**// Create 'new Promise Object' using its constructor which passes an argument i.e. executer as callBack function takes 2 parameter i.e. resolve & reject function.**

**//This is called Promise Produce. (Basically Producing Promises). Promise using constructor**const promiseObject1 = new Promise((resolve, reject)=>{

setTimeout(()=>{  
 let roll\_no = [1,2,3,4,5];

***// If able to Resolve the above instruction, use resolve function***  
 resolve(roll\_no); ***//call resolve function by passing the required argument to fetch it.***

***//If unable to resolve the above instruction, use reject function***  
 reject(‘error because resolve not called');

},2000);

});

**// Promise using Object**const getBioData = (acceptRollNo)=>{ ***//(2). Getting value of Argument of getBioData function while calling it and Passing in acceptRollNo as parameter***

**//When we have to create new task, will always consider creating that task as new Promise** return new Promise((resolve, reject)=>{ **// setTimeout(Handler, Time in ms, Argument);**

setTimeout((acceptRollNo)=>{ ***//(4). Passing acceptRollNo as Parameter***

let bioData = {  
 Name: 'Chandan Kumar',  
 Age: 21  
 }

resolve(`My name is ${bioData.Name}, My Roll No. is ${acceptRollNo} and My Age is ${bioData.Age}.`);

}, 2000, acceptRollNo); ***//(3). Accepting acceptRollNo as argument***

});

};

**// Promise Consumption using Async Await. This is where exactly Async-Await comes into play.**

async function getData(){ ***//Because of async function, This function is going to directly return a promise which passes either resolve or reject function***

**//To fetch the data of 1st promise. We use await keyword, because the data in 1st promise returning after 2 sec. So, await keyword let function to wait till 2 sec then execute it.**

const rollNoData = await promiseObject1; ***//1st promise’s data*** console.log(rollNoData);

const bioDatas = await getBioData(rollNoData[0]); ***//2nd promise’s data*** console.log(bioDatas);

return bioDatas; ***//This instruction will go under pending state. Becuase of return keyword & calling its function as function expression it'll directly trying to return the data (bioDatas) of 2nd Promise that has already been set to execute their instruction after 2 sec while producing Second Promise.***

}

const data = getData().then((repeatData)=>{ ***//Function Expression*** console.log(repeatData);

}).catch((error)=>{  
 console.log(`Your Error: ${error}`);  
});

console.log(data); //***Because of this, return bioDatas(); wil execute***

**Output:**Promise {<pending>} ***//return BioDatas(); execute before 2 sec***[1, 2, 3, 4, 5] ***//Print after 2 second***  
My name is Chandan Kumar, My Roll No. is 1 and My Age is 21. ***//Print after 2 second***My name is Chandan Kumar, My Roll No. is 1 and My Age is 21. ***//Print after 2 second*** **OR**  
Your Error: ReferenceError: repeatDat is not defined ***//Under Error Condition or say, reject function executed in the promise when, resolve function somehow failed to execute.***

**Error Handling in JS**

Error Handling in JavaScript or we can say that Error Handling in Async-Await.

There are two method in Error Handling in Async-Await.

* try {} 🡪 Contains all resolve insturctions.
* catch() 🡪 Contains reject insturctions.

**Example:**

**// Create 'new Promise Object' using its constructor which passes an argument i.e. executer as callBack function takes 2 parameter i.e. resolve & reject function.**

**//This is called Promise Produce. (Basically Producing Promises). Promise using constructor**const promiseObject1 = new Promise((resolve, reject)=>{

setTimeout(()=>{  
 let roll\_no = [1,2,3,4,5];

***// If able to Resolve the above instruction, use resolve function***  
 resolve(roll\_no); ***//call resolve function by passing the required argument to fetch it.***

***//If unable to resolve the above instruction, use reject function***  
 reject(‘error because resolve not called');

},2000);

});

**// Promise using Object**const getBioData = (acceptRollNo)=>{ ***//(2). Getting value of Argument of getBioData function while calling it and Passing in acceptRollNo as parameter***

**//When we have to create new task, will always consider creating that task as new Promise** return new Promise((resolve, reject)=>{ **// setTimeout(Handler, Time in ms, Argument);**

setTimeout((acceptRollNo)=>{ ***//(4). Passing acceptRollNo as Parameter***

let bioData = {  
 Name: 'Chandan Kumar',  
 Age: 21  
 }

resolve(`My name is ${bioData.Name}, My Roll No. is ${acceptRollNo} and My Age is ${bioData.Age}.`);

}, 2000, acceptRollNo); ***//(3). Accepting acceptRollNo as argument***

});

};

**// Promise Consumption using Async Await. This is where exactly Async-Await comes into play.**

async function getData(){ ***//Because of async function, This function is going to directly return a promise which passes either resolve or reject function***

try{

**//To fetch the data of 1st promise. We use await keyword, because the data in 1st promise returning after 2 sec. So, await keyword let function to wait till 2 sec then execute it.**

const rollNoData = await promiseObject1; ***//1st promise’s data*** console.log(rollNoData);

const bioDatas = await getBioData(rollNoData[0]); ***//2nd promise’s data*** console.log(bioData); ***//Error occurred in bioData***

return bioDatas; ***//This instruction will go under pending state. Becuase of return keyword & calling its function as function expression it'll directly trying to return the data (bioDatas) of 2nd Promise that has already been set to execute their instruction after 2 sec while producing Second Promise.***

}catch(error){  
 console.log(`Your Error: ${error}`);  
}

}

const data = getData().then((repeatData)=>{ ***//Function Expression*** console.log(repeatData);

});

console.log(data); //***Because of this, return bioDatas(); wil execute***

**Output:**Promise {<pending>} ***//return BioDatas(); execute before 2 sec***[1, 2, 3, 4, 5] ***//Print after 2 second***  
Your Error: ReferenceError: bioData is not defined ***//Under Error Condition or say, function went to the catch scope instead of going to the try scope.***