1. Functions

```
// function declaration
function greet(){
  console.log('hello there');
}

// function expression
const speak = function(){
  console.log('good day!');
};

greet();
speak();
```

Hoisting (The function is hoisted at top)

No Hoisting

Functions with arguements

```
const speak = function(name){
  console.log(`good day ${name}`);
};

speak('mario');
```

```
const speak = function(name = 'luigi', time = 'night'){
  console.log(`good ${time} ${name}`);
};

speak();
```

Functions with default arguements

Function with return type

```
const calcArea = function(radius){
  let area = 3.14 * radius**2;
  return area;
}

const a = calcArea(5);
console.log(a);
```

Arrow Functions

```
// regular function
const calcArea = function(radius){
  return 3.14 * radius**2;
}

// arrow function
const calcArea = (radius) => {
  return 3.14 * radius**2;
};
```

Simplifying arrow function

```
// arrow function
const calcArea = radius => 3.14 * radius**2;
```

If function contains a single return statement, then remove return keyword and perenthesis

If single parameter then remove perenthesis

Callback Function

```
// callbacks & foreach

const myFunc = (callbackFunc) => {
  // do something
  let value = 50;
  callbackFunc(value);
};

myFunc(function(value){
  // do something
  console.log(value);
});
```

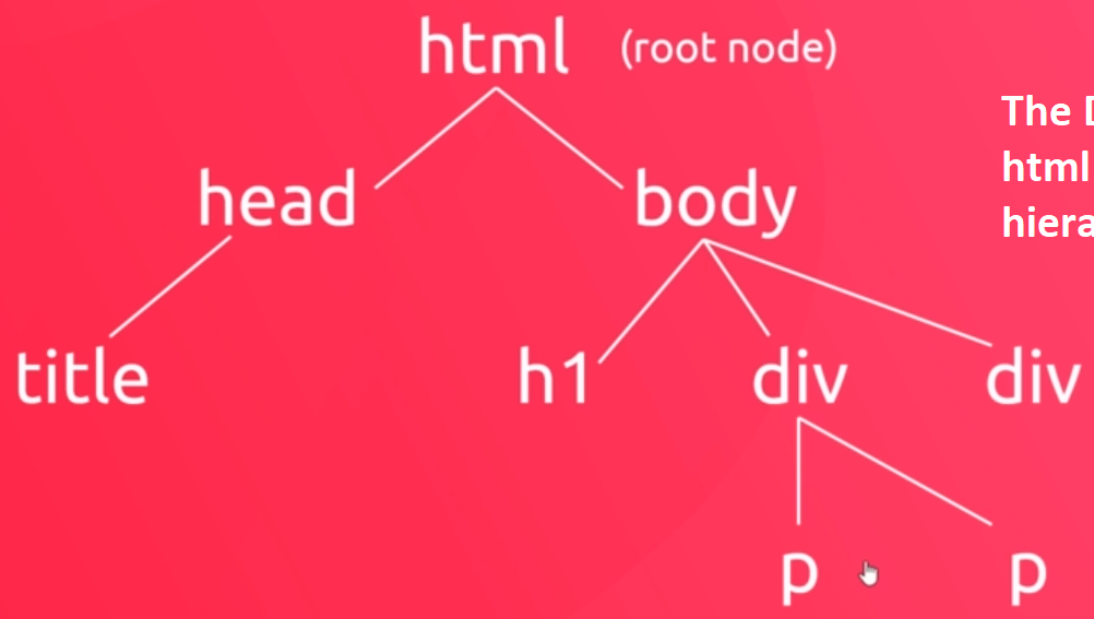
A function to which a function is passed as a parameter is called a callback function.

Document Object Model

- Created by the browser
- 'document' object

Udemy

Document Object Model



The DOM sees the html page as a hierarchial tree

Query Selector

```
const para = document.querySelector('p');
```

Searches the hrml page from and top to bottom and gives reference to first **p** tag

```
const para = document.querySelector('.error');
```

Searches the hrml page from and top to bottom and gives reference to first tag with class '.error'

```
const para = document.querySelector('div.error');
```

Searches the hrml page from and top to bottom and gives reference to first **div** tag with class '.error'

```
const paras = document.querySelectorAll('p')
```

Searches the hrml page from and top to bottom and gives reference to all **p** tag s

querySelectorAll returns a node list containing all the tags of desired type

HTMLCollection methods

.item()
.namedItem()

Node List methods

.item()
.entries()
.forEach()
.keys()
.values()

CHANGE PAGE CONTENT

```
para.innerText = 'ninjas are awesome!';
```

Overwrites the already existing text inside

```
para.innerText += 'ninjas are awesome!';
```

Appends to already existing text inside

```
const content = document.querySelector('.content');

content.innerHTML = '<h2>THIS IS A NEW H2</h2>';

content.innerHTML += '<h2>THIS IS A NEW H2</h2>';
```

Used to overwrite the innner Html of a tag (div in this case)

Used to append tothe innner Html of a tag (div in this case)

Get and Set Attributes

```
const link = document.querySelector('a');

link.getAttribute('href')
link.setAttribute('href', 'https://www.thenetninja.co.uk');
```

Overwrites the attribute 'href' with 'https://www. .....'
If the attribute is not defined initially, it sets a new attribute

Styling using CSS

```
const title = document.querySelector('h1');

//title.setAttribute('style', 'margin: 50px;');

title.style.margin = '50px';
```

⇨ Overwrites already existing inline style

⇨ To avoid this, we use an object 'style' which has all the styling elements as its data elements

Adding and Removoing Classes

```
const content = document.querySelector('p');

console.log(content.classList);
content.classList.add('error');
content.classList.remove('error');
```

⇨ Gives list of classes of p tag

```
p.innerText
```

⇨ Only those which are visible to the browser are selected i.e if a tag is styled as display:none, it wont be selected

```
<p>lorem <span style="display: none;">error</span> ipsum</p>
```

```
p.textContent
```

⇨ This solves the problem -> gives reference to everything inside the text

```
title.classList.toggle('test')
```

⇨ Used to toggle the class, ie if the tag already contains 'title' class, it removes it else it adds the class

Parent Children and Sibling

```
article.children);
```

HTML collection of alll the children

```
Array.from(article.children)
```

Creating array from HTML collection. This is hwlpful when we want to use 'for each' method on it

```
(title.parentElement);
(title.parentElement.parentElement);
(title.nextElementSibling);
(title.previousElementSibling);
```

Gives next sibling

Gives prev sibling

Basic Events

```
button.addEventListener('click', () => {
  console.log('you clicked me');
});
```

Actively listens to an event
1st parameter -> event
2nd parameter -> callback func

```
item.addEventListener('click', e => {
  //console.log('item clicked');
  //console.log(e);
  console.log(e.target);
});
```

e -> gives reference to event
e.target -> html element on which event took place

```
e.target.remove();
```

Used to remove e.target from the webpage

```
const ul = document.querySelector('ul');
const button = document.querySelector('button');
button.addEventListener('click', () => {
  const li = document.createElement('li');
  li.textContent = 'something new to do';
  //ul.append(li);
  ul.prepend(li);
});
```

Creates a new element
But its not inserted into the web page yet

Inserts element at last

Inserts the element at the beginning

Event Deligation and Event Bubbling

Event Bubbling : When an object (like a button) is clicked, an event is directed to the object. If an event handler is set for the object, the event handler(callback func) is triggered. Then the event bubbles up (like a bubble in water) to the objects parent.

When an event takes place, callback function for that event is fired, but thats not the end
That event bubbles up the DOM to its parent and checks is there is any event listner attached to it and fires the callback function attached to it and the process continues

```
e.stopPropagation()
```

This stops event bubbling right from the tag in which it is placed.
It is placed in event listner of a tag
EG : Hierarchy : body > ul > li
If there are event listners attached to ul and li and an event occurs wrt to li, event bubbles up to li and then ul and finally body .
If e.stopPropogation() is placed in
1. li --> callback function of li event bubbling stops right at li
2. ul -> callback function of ul is fired and event bubbling stops right there

Event Deligation : Event Delegation is basically a pattern to handle events efficiently. Instead of adding an event listener to each and every similar element, we can add an event listener to a parent element and call an event on a particular target using the . target property of the event object

Form Handling

Submit Event

```
e.preventDefault();
```

⇨ Prevents default action of an event

```
<form class="signup-form">
  <input type="text" id="username" placeholder="username">
  <input type="submit" value="submit">
</form>
```

In submit event, we attach the event listener to the form instead of the submit button
Default action of a submit event is to refresh the page when event occurs -> we prevent the default action using e.preventDefaut()

To have a reference to the username input field of the form we havve two ways

1. To attach a query selector to username
2. refer to it using the 'form' using the id or name
   form.username
   'username' is id of the input field (id="username") or name (name="username")

The second approach is a better one

Regular Expression

When we want to put some restrictions on the input we use Regular Expression

^ -> start of input field
$ -> end of input field
[] -> character set
[a-z] : match charaters from a-z
[a-zA-z] : match all capital and small case letters
But this matches only one character, what if we want to match words
{a,b} -> no of letters
[A-Za-z]{6-10} : word of 6-10 letters with only small case and capital letters
.{6-10} : word of any character of length 6-10

[a-z]{6-10} : this means the input should contain a-z and no of letters should be 6-10 and there
%**adldlfk**123@ -> Even this is accepted coz it contains alphabets from a-z and 6-10 letters

^ [a-z]{6-10} $: It means a-z alphabett of 6-10 letters must be present at start as well as at end

Regular expression always starts and ends with '/'

```
const username = 'shaun';
const pattern = /[a-z]{6,}/;

let result = pattern.test(username);
```

Returns true if username passes the test ie pattern

```
let result = username.search(pattern);
```

Returns the index at which pattern is present in username, -1 if not present

Live Feedback

Keyup event -> a key is clicked on keyboard

Everytime the user clicks someting on the keyboard, we send the feedback -> to achieve this we need to test the input field for pattern each time user clicks something on the keyboard

```
// live feedback
form.username.addEventListener('keyup', e => {
  //console.log(e.target.value, form.username.value);
  if(usernamePattern.test(e.target.value)){
    form.username.setAttribute('class', 'success');
  } else {
    form.username.setAttribute('class', 'error');
  }
});
```

```
// setTimeout(() => {
//    // do something
// }, 3000);

let i = 0;
const timer = setInterval(() => {
  console.log('hello');
  i++;
  if(i === 5){
    clearInterval(timer);
  }
}, 1000);
```

This is a method stores in the window object that does something mentioned in the callback function after given interval of time

setTimeout(callback func, time) -> Fires the callback function after 'time' ms

This is a method stored int the window object that does something repeatedly as mentioned in the callback function over fixed interval of time as mentioned in the function call

setInterval(callback func, time) -> fires the callback func every 'time' ms

aborts the function when condition is true

BY SHIVKARAMAN

Array Methods

1. for each

```
arrayname.foreach(callback func)
arranName.foreach( (data, index) => {
        console.log(data, index);
});
```

2. filter -> Creates a new array with items for which the callback func returns true. The original array is unchanged

```
const users = [
  {name: 'shaun', premium: true},
  {name: 'yoshi', premium: false},
  {name: 'mario', premium: false},
  {name: 'chun-li', premium: true}
];

const premiumUsers = users.filter(user => {
  return user.premium;
});

console.log(premiumUsers);
```

Creates copy of object

```
▼(2) [{…}, {…}]
  ▶0: {name: "shaun", premium: true}
  ▶1: {name: "chun-li", premium: true}
   length: 2
  ▶__proto__: Array(0)
```

3. map -> creates a new array whose elements depend on the original array This mmethod can alter the original array too

```
const products = [
  {name: 'gold star', price: 20},
  {name: 'mushroom', price: 40},
  {name: 'green shells', price: 30},
  {name: 'banana skin', price: 10},
  {name: 'red shells', price: 50}
];

const saleProducts = products.map((product) => {
  if(product.price > 30){
    return {name: product.name, price: product.price / 2};
  } else {
    return product;
  }
});
```

```
product.price = product.price / 2;
```

This would alter the price in original array

This is object itself and not a copy

```
▼(5) [{…}, {…}, {…}, {…}, {…}]
  ▶0: {name: "gold star", price: 20}
  ▶1: {name: "mushroom", price: 20}
  ▶2: {name: "green shells", price: 30}
  ▶3: {name: "banana skin", price: 10}
  ▶4: {name: "red shells", price: 25}
   length: 5
  ▶__proto__: Array(0)
```

4. reduce -> Iterates over the array and returns a value depending upon the callback func
value-> can be of any data type

```
const scores = [10, 20, 60, 40, 70, 90, 30];

const result = scores.reduce((acc, curr) => {
  if(curr > 50){
    acc++;
  }
  return acc;
}, 0);
```

Accumulator

Current value
(element of array)

Initial value of **acc**

5. find -> returns the first array element that satisfies the condition

```
const scores = [10, 5, 0, 40, 60, 10, 20, 70];

const firstHighScore = scores.find((score) => {
  return score > 50;
});
```

6. Sort -> sorts the array

arrayName.sort() -> this sorts elements if they are of primitive type, if the array is array of objects we need to pass in a callback function which helps sort the array -> this func looks only at the first part of any element

```
const players = [
  {name: 'mario', score: 20},
  {name: 'luigi', score: 10},
  {name: 'chun-li', score: 50},
  {name: 'yoshi', score: 30},
  {name: 'shaun', score: 70}
];

players.sort((a,b) => {
  if(a.score > b.score){
    return -1;
  } else if (b.score > a.score){
    return 1;
  } else {
    return 0;
  }
});
```

return type
1. +ve -> swap
2. 0 -> no change
3. -ve number -> no change

```
[10, 50, 20, 5, 35, 70, 45];
```

scores.sort(()

```
[10, 20, 35, 45, 5, 50, 70]
```

solution

```
scores.sort((a,b) => b - a);
```

```
players.sort((a,b) => {
  return b.scor - a.score
})
```

SIMPLIFIED VERSION

BY SHIVKARAMAN

Chaining Array Methods

```
const products = [
  {name: 'gold star', price: 30},
  {name: 'green shell', price: 10},
  {name: 'red shell', price: 40},
  {name: 'banana skin', price: 5},
  {name: 'mushroom', price: 50}
];

const filtered = products.filter(product => product.price > 20);

const promos = filtered.map(product => {
  return `the ${product.name} is ${product.price / 2} pounds`;
});

console.log(promos);
```

Chaining

```
const promos = products
  .filter(product => product.price > 20)
  .map(product => `the ${product.name} is ${product.price / 2} pounds`);
```

Creating a search bar

```
addForm.add.value.trim();
```

⇨ Removes unwanted blank (space) in the input

```
addForm.reset();
```

⇨ When a submit event takes place, all the inputs given in the form remain there. This is used to clear all input fields in the form

Search for an element in list

```
const filterTodos = (term) => {

  Array.from(list.children)
    .filter((todo) => !todo.textContent.includes(term))
    .forEach((todo) => todo.classList.add('filtered'));

  Array.from(list.children)
    .filter((todo) => todo.textContent.includes(term))
    .forEach((todo) => todo.classList.remove('filtered'));
};

// keyup event
search.addEventListener('keyup', () => {
  const term = search.value.trim().toLowerCase();
  filterTodos(term);
});
```

filter out elements of list if the searched element is not present

iterate over each element and add class 'filtered' which makes the elements style -> display: none

This part fo code removes the 'filter' class when the element searched is removed from search bar

1. Attach an event listner with 'keyup' event to input field of search bar
2. Trin extra spaces at beginning and end pg the todo to be added
3. Call a function tpo filter out the list according to the searched element

Dates and Time

Method	Description
now()	Returns the numeric value corresponding to the current time (the number of milliseconds elapsed since January 1, 1970 00:00:00 UTC)
getFullYear()	Gets the year according to local time
getMonth()	Gets the month, from 0 to 11 according to local time
getDate()	Gets the day of the month (1–31) according to local time
getDay()	Gets the day of the week (0-6) according to local time
getHours()	Gets the hour from 0 to 23 according to local time
getMinutes	Gets the minute from 0 to 59 according to local time
getUTCDate()	Gets the day of the month (1–31) according to universal time
setFullYear()	Sets the full year according to local time
setMonth()	Sets the month according to local time
setDate()	Sets the day of the month according to local time
setUTCDate()	Sets the day of the month according to universal time

```
// dates & times
const now = new Date();

console.log(now);
console.log(typeof now);

// year, months, day, times
console.log('getFullYear:', now.getFullYear());
console.log('getMonth:', now.getMonth());
console.log('getDate:', now.getDate());
console.log('getDay:', now.getDay());
console.log('getHours:', now.getHours());
console.log('getMinutes:', now.getMinutes());
console.log('getSeconds:', now.getSeconds());

// timestamps
console.log('timestamp:', now.getTime());

// date strings
console.log(now.toDateString());
console.log(now.toTimeString());
console.log(now.toLocaleString());
```

```
Wed Mar 20 2019 10:53:16 GMT+0000 (Greenwich Mean Time
object

getFullYear: 2019
getMonth: 2
getDate: 20
getDay: 3
getHours: 10
getMinutes: 53
getSeconds: 16

timestamp: 1553079196801
```

in ms

```
Wed Mar 20 2019
10:53:53 GMT+0000 (Greenwich Mean Time)
20/03/2019, 10:53:53
```

BY SHIVKARAMAN

Timestamps are used to compare times

```
const before = new Date('February 1 2019 7:30:59');
const now = new Date();
const diff = now.getTime() - before.getTime();
const mins = Math.round(diff / 1000 / 60);
```

Gives difference in time in minutes

60s = 1min

1s = 1000ms

Creating a Date object from

```
const timestamp = 1675938474990;
const Date = new Date(timestamp)
```

Creating A Digital Clock

```
const clock = document.querySelector('.clock');

const tick = () => {

  const now = new Date();

  const h = now.getHours();
  const m = now.getMinutes();
  const s = now.getSeconds();

  const html = `
    <span>${h}</span> :
    <span>${m}</span> :
    <span>${s}</span>
  `;

  clock.innerHTML = html;

};

setInterval(tick, 1000);
```

Date and Time Library

```
const now = new Date();
// formatting options
console.log(dateFns.format(now, 'YYYY'));
console.log(dateFns.format(now, 'MMM'));
console.log(dateFns.format(now, 'dddd'));
console.log(dateFns.format(now, 'Do'));
console.log(dateFns.format(now, 'dddd, Do, MMMM, YYYY'));
console.log(dateFns.format(now, 'dddd Do MMMM YYYY'));
```

```
2019
Mar
Wednesday
20th
Wednesday, 20th, March, 2019
Wednesday 20th March 2019
```

```
<script src="http://cdn.date-fns.org/v1.9.0/date_fns.min.js"></script>
```

Async JavaScript

- Governs how we perform tasks which take some time to complete (e.g. Getting data from a database)

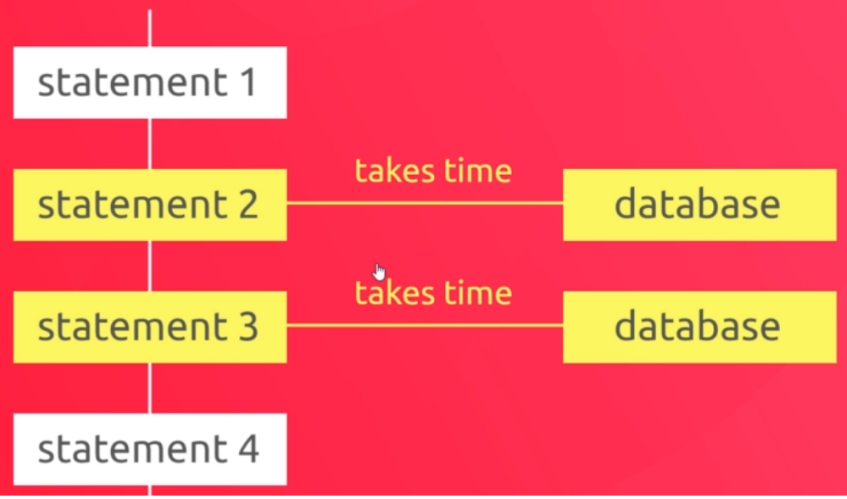
Start something now and finish it later

Synchronous JavaScript

- JavaScript can run ONE statement at a time

```
console.log('line one');
console.log('line two');
console.log('line three')
```

Single Threaded



JS is a single threaded languaage ie it can execute one statement at a time and in a sequence

executed before statement and so on. But what if statement 2 and 3 are requesting some dta from the data base and it takes around 5-6s seconds , now we are blocking the rest of the code coz state,ent 2 and 3 are waiting -> This is the disadvt of synchronous code

BY SHIVKARAMAN

HTTP Requests

- Make HTTP requests to get data from another server

- We make these requests to API endpoints

API Endpoints

- Example API endpoint:
  http://www.musicapi.com/artist/moby

Sending http Request

```
const request = new XMLHttpRequest();

request.addEventListener('readystatechange', () => {
  console.log(request, request.readyState);
});

request.open('GET', 'https://jsonplaceholder.typicode.com/todos/');
request.send();
```

Creating a requestt object. This is used to send the request for data

Fires every time when there is a state change

State of current request

Type of request we want to make

End point from which we want to get data from

This is used to set up the request and the request is not actually sent

Sends the request

The **XMLHttpRequest.readyState** property returns the state an XMLHttpRequest client is in.
An XHR client exists in one of the following states:

Value	State	Description
0	UNSENT	Client has been created. `open()` not called yet.
1	OPENED	`open()` has been called.
2	HEADERS_RECEIVED	`send()` has been called, and headers and status are available.
3	LOADING	Downloading; `responseText` holds partial data.
4	DONE	The operation is complete.

Request is complete

```
const request = new XMLHttpRequest();

request.addEventListener('readystatechange', () => {
  //console.log(request, request.readyState);
  if(request.readyState === 4){
    console.log(request.responseText);
  }
});

request.open('GET', 'https://jsonplaceholder.typicode.com/todos/');
request.send();
```

This condition alone is not enough coz there can be some error in the request

To overcome this, we need to check for status along with ready state

status = 200 means data has been sent successfully

```
if(request.readyState === 4 && request.status === 200){
  console.log(request, request.responseText);
} else if(request.readyState === 4){
  console.log('could not fetch the data');
}
```

Data which we had requested

BY SHIVKARAMAN

Callback Function

To improve the reusability of the code, we wrap the entire code into a function.
But now when we call the function, response text is being logged to the console.

Instead it would be better if we pass in a callback function to which tells what to do with the data

```
const getTodos = (callback) => {
  const request = new XMLHttpRequest();

  request.addEventListener('readystatechange', () => {
    if(request.readyState === 4 && request.status === 200){
      callback(undefined, request.responseText);
    } else if(request.readyState === 4){
      callback('coud not fetch data', undefined);
    }
  });

  request.open('GET', 'https://jsonplaceholder.typicode.com/todos/');
  request.send();
};

getTodos((err, data) => {
  console.log('callback fired');
  console.log(err, data);
});
```

Demonstration of Asynchronous Code

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

getTodos((err, data) => {
  console.log('callback fired');
  if(err){
    console.log(err);
  } else {
    console.log(data);
  }
});

console.log(3);
console.log(4);
```

Asynchronous Function

```
1
2
3
4
callback fired
[
  {
    "userId": 1,
    "id": 1,
    "title": "delectus aut autem",
    "completed": false
  },
  {
    "userId": 1,
    "id": 2,
    "title": "quis ut nam facilis et officia qui",
    "completed": false
  },
  {
    "userId": 1,
    "id": 3,
    "title": "fugiat veniam minus",
    "completed": false
  },
  {
    "userId": 1,
    "id": 4,
    "title": "et porro tempora",
```

JSON Data

JSON is a form of tdata that is used to transfer data between server and client

JSON datais just a string formatted to look like array of objects

TO use the data efficiently we need to create an actual object using the string

```
const data = JSON.parse(request.responseText);
```

Converts the JSON string and converts it into actual JS object

Creating Our Own JSON file

Unlike normal JS object, everyting in JSON has to be quoted inside double quotes wxcept values like numbers, boolean

```
[
  { "text": "play mariokart", "author": "shaun" },
  { "text": "by some bread", "author": "Mario" },
  { "text": "do the plumming", "author": "Luigi" }
]
```

todos.json

```
request.open('GET', 'todos.json');
```

```javascript
const getTodos = (callback) => {
  const request = new XMLHttpRequest();

  request.addEventListener('readystatechange', () => {
    if(request.readyState === 4 && request.status === 200){
      const data = JSON.parse(request.responseText);
      callback(undefined, data);
    } else if(request.readyState === 4){
      callback('coud not fetch data', undefined);
    }
  });

  request.open('GET', 'todos.json');
  request.send();
};

getTodos((err, data) => {
  console.log('callback fired');
  if(err){
    console.log(err);
  } else {
    console.log(data);
  }
});
```

This is defined inside function se if we want to use the same function for different JSON file or API.
So instead we pass in the filename as a parameter

```javascript
const getTodos = (resource, callback) => {
  const request = new XMLHttpRequest();

  request.addEventListener('readystatechange', () => {
    if(request.readyState === 4 && request.status === 200){
      const data = JSON.parse(request.responseText);
      callback(undefined, data);
    } else if(request.readyState === 4){
      callback('coud not fetch data', undefined);
    }
  });

  request.open('GET', resource);
  request.send();
};

getTodos('todos/luigi.json', (err, data) => {
  console.log('callback fired');
  if(err){
    console.log(err);
  } else {
    console.log(data);
  }
});
```

Callback Hell

Callback Hell: Callback Hell is essentially nested callbacks stacked below one another forming a pyramid structure. Every callback depends/waits for the previous callback, thereby making a pyramid structure that affects the readability and maintainability of the code

```javascript
getTodos('todos/luigi.json', (err, data) => {
  console.log(data);
  getTodos('todos/mario.json', (err, data) => {
    console.log(data);
    getTodos('todos/shaun.json', (err, data) => {
      console.log(data);
    });
  });
});
```

Promises in JS

Nesting of callback function in callback hell was messy and hard to read.
Instead we use PROMISES

```
const getSomething = () => {

  return new Promise((resolve, reject) => {
    // fetch something
    //resolve('some data');
    reject('some error');
  });

};

getSomething().then((data) => {
  console.log(data);
}, (err) => {
  console.log(err);
});
```

A normal function that returns a **Promise**

These two are functions that are built into the promise API

On Success

On Failure

Fired when resolve() is called in promise

Fired when reject() is called in promise

Better way of doing it

```
getSomething().then(data => {
  console.log(data);
}).catch(err => {
  console.log(err);
})
```

Applying Promise to Previous Example

```javascript
const getTodos = (resource, callback) => {
  const request = new XMLHttpRequest();

  request.addEventListener('readystatechange', () => {
    if(request.readyState === 4 && request.status === 200){
      const data = JSON.parse(request.responseText);
      callback(undefined, data);
    } else if(request.readyState === 4){
      callback('coud not fetch data', undefined);
    }
  });

  request.open('GET', resource);
  request.send();
};

getTodos('todos/luigi.json', (err, data) => {
  console.log('callback fired');
  if(err){
    console.log(err);
  } else {
    console.log(data);
  }
});
```

```javascript
const getTodos = (resource) => {

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

    request.addEventListener('readystatechange', () => {
      if(request.readyState === 4 && request.status === 200){
        const data = JSON.parse(request.responseText);
        resolve(data);
      } else if(request.readyState === 4){
        reject('error getting resource');
      }
    });

    request.open('GET', resource);
    request.send();
  });

};
```

```javascript
getTodos('todos/luigi.json').then(data => {
  console.log('promise resolved:', data);
}).catch(err => {
  console.log('promise rejected:', err);
});
```

```javascript
getTodos('todos/luigi.json', (err, data) => {
  console.log(data);
  getTodos('todos/mario.json', (err, data) => {
    console.log(data);
    getTodos('todos/shaun.json', (err, data) => {
      console.log(data);
    });
  });
});
```

```javascript
getTodos('todos/luigi.json').then(data => {
  console.log('promise 1 resolved:', data);
  return getTodos('todos/mario.json');
}).then(data => {
  console.log('promise 2 resolved:', data);
  return getTodos('todos/shaun.json');
}).then(data => {
  console.log('promise 3 resolved:', data);
}).catch(err => {
  console.log('promise rejected:', err);
});
```

BY SHIVKARAMAN

Fetch API

Lot of developers use XMLHttpRequest( ) to fetch data from an API. Thats not bad but a new function is added in modern JS which is much faster then that
which is the **fetch('API')** which fetches data from the API given as parameter. It returns a promise

```
fetch('todos/luigi.json').then((response) => {
  console.log('resolved', response);
}).catch((err) => {
  console.log('rejected', err);
});
```

The fetch API shows ERROR (reject( )) if and only if there was some **network error** . If we mistype the URL while requesting for API then we dont get a rejection **It still resolves instead of reject** To solve this we check if status = 200 which means success

```
response.json();
```

This contains the data that was requested from the API and this cannot be stored in some constant as we did for **response.parse() in XMLHttpRequest( )**

Since this returns a promise, we can chain an another **.then()** to the fetch method to take the promise returned by the fetch() method and use

```
fetch('todos/luigi.json').then((response) => {
  console.log('resolved', response);
  return response.json();
}).then(data => {
  console.log(data);
}).catch((err) => {
  console.log('rejected', err);
});
```

asynch and await

These are methods recently added into JS which helps chain promises in a better way

aynch : It is a keyword used along with a function to make that that function an asunchronous function

```
const getTodos = () => {

};
```

Normal func

```
const getTodos = async () => {

};
```

Asynchronous Func

await: This is used along with fetch( ) method. What we normally do is pack on a .then() method after fetch() which is fired when the promise has resolved (resolve( )). Instead we use await keyword like below.

```
const response = await fetch('todos/luigi.json');
```

The await( ) keyword stops JS right there until the promise returned by fetch( ) is resolved

```
fetch('todos/luigi.json').then((response) => {
  console.log('resolved', response);
  return response.json();
}).then(data => {
  console.log(data);
}).catch((err) => {
  console.log('rejected', err);
});
```

same as

```
const getTodos = async () => {

  const response = await fetch('todos/luigi.json');
  const data = await response.json();

  return data;

};

getTodos()
  .then(data => console.log('resolved:', data));
```

throw and catch error

```
const getTodos = async () => {

  const response = await fetch('todos/luigis.json');
  const data = await response.json();

  return data;

};

getTodos()
  .then(data => console.log('resolved:', data))
  .catch(err => console.log('rejected:', err.message));
```

In response.json(), if there is some error in the json data it will result in an error and the catch( ) method is fired.

But if there is an error in fetch() method we have already seen that only network errors are detected by fetch( ), hence even if there is some error in fetch( ) (except network error) fetch( ) doesnot reject the promise.
The error messgae that will be printed is the there is some error in JSON file (as promise that was returned by response.json() was rejected)
This is not right

```
const getTodos = async () => {

  const response = await fetch('todos/luigis.json');

  if(response.status !== 200){
    throw new Error('cannot fetch the data');
  }

  const data = await response.json();

  return data;

};

getTodos()
  .then(data => console.log('resolved:', data))
  .catch(err => console.log('rejected:', err.message));
```

To resolve the issue, we have to check manually and throw our own error

Local Storage

Application Data

We ahve seen previousluy that all the input fields and the data is erased from the browser when we refresh the page. But we can store the data somewhere and retrieve that back when we open the page again usoing the two following ways

- Set up a database to store & retrieve data
- Use local storage to store and retrieve data

The `localStorage` object allows you to save key/value pairs in the browser.

Note

The localStorage object stores data with no expiration date.

The data is not deleted when the browser is closed, and are available for future sessions.

DATA WE STORE IN THE LOCAL STORAGE HAS TO BE A STING

```
// store data in local storage
localStorage.setItem('name', 'mario');
localStorage.setItem('age', 50);

// get data from local storage
let name = localStorage.getItem('name');
let age = localStorage.getItem('age');

console.log(name, age);

// updating data
localStorage.setItem('name', 'luigi');
localStorage.age = '40';
```

Save Data to Local Storage

```
localStorage.setItem(key, value);
```

Read Data from Local Storage

```
let lastname = localStorage.getItem(key);
```

Remove Data from Local Storage

```
localStorage.removeItem(key);
```

Remove All (Clear Local Storage)

```
localStorage.clear();
```

Anyting that we store in the local storage has to be a string
If data is not a string, we convert it to string

```
const todos = [
  {text: 'play mariokart', author: 'shaun'},
  {text: 'buy some milk', author: 'mario'},
  {text: 'buy some bread', author: 'luigi'}
];

//console.log(JSON.stringify(todos));
localStorage.setItem('todos', JSON.stringify(todos))
```

Converts the data into a JSON stirng

```
const stored = localStorage.getItem('todos');

console.log(JSON.parse(stored));
```

Converts the data in local storage back to array

OOP

```
class User {
  constructor(username, email){
    // set up properties
    this.username = username;
    this.email = email;
  }
}

const userOne = new User('mario', 'mario@thenetninja.co.uk');
const userTwo = new User('luigi', 'luigi@thenetninja.co.uk');

console.log(userOne, userTwo);

// the 'new' keyword
// 1 - it creates a new empty object {}
// 2 - it binds the value of 'this' to the new empty object
// 3 - it calls the constructor function to 'build' the object
```

```
class User {
  constructor(username, email){
    // set up properties
    this.username = username;
    this.email = email;
    this.score = 0;
  }
  login(){
    console.log(`${this.username} just logged in`);
  }
  logout(){
    console.log(`${this.username} just logged out`);
  }
  incScore(){
    this.score += 1;
    console.log(`${this.username} has a score of ${this.score}`);
  }
}

const userOne = new User('mario', 'mario@thenetninja.co.uk');
const userTwo = new User('luigi', 'luigi@thenetninja.co.uk');

console.log(userOne, userTwo);

userOne.login();
userOne.logout();
userOne.incScore();
userOne.incScore();
userOne.incScore().incScore().incScore();
```

```
login(){
  console.log(`${this.username} just logged in`);
  return this;
}
logout(){
  console.log(`${this.username} just logged out`);
  return this;
}
incScore(){
  this.score += 1;
  console.log(`${this.username} has a score of ${this.score}`);
  return this;
}
```

username.login()
username.logout()

username.login().logout()

```
Uncaught TypeError: Cannot read property
'incScore' of undefined
    at sandbox.js:27
```

this is because the methods dont return any value and to chain methods the methods must return some value on which the chained method will be called

BY SHIVKARAMAN

Inheritence

```
class Admin extends User{
  deleteUser(user){
    users = users.filter((u) => {
      return u.username !== user.username
    })
  }
}

const userOne = new User('mario', 'mario@thenetninja.co.uk');
const userTwo = new User('luigi', 'luigi@thenetninja.co.uk');
const userThree = new Admin('shaun', 'shaun@thenetninja.co.uk');

console.log(userOne, userTwo, userThree);

let users = [userOne, userTwo, userThree];
console.log(users);

userThree.deleteUser(userTwo);
```

SUPER KEYWORD

```
class Admin extends User{
  constructor(username, email, title){
    super(username, email);
    this.title = title;
  }
  deleteUser(user){
    users = users.filter(u => u.username !== user.username);
  }
}

const userOne = new User('mario', 'mario@thenetninja.co.uk');
const userTwo = new User('luigi', 'luigi@thenetninja.co.uk');
const userThree = new Admin('shaun', 'shaun@thenetninja.co.uk', 'black-belt-nin

console.log(userThree);
```

Calls the constructor of the user class. If not used, it shows error

Constructor Underhood

```
class User {
  constructor(username, email){
    // set up properties
    this.username = username;
    this.email = email;
    this.score = 0;
  }
  login(){
    console.log(`${this.username} just logged in`);
  }
```

Class is a new keyword added to JS recently and it is just acts as a mast and it is not how actually objects are created underneath

The methods are stored in the prototype when we define an object using class

This is the convention used to represent an object
FIRST LETTER CAPITAL

```
function User(username, email){
  this.username = username;
  this.email = email;
  this.login = function(){
    console.log(`${this.username} has logged in`)
  }
}
```

This is the actual code that runs underneath

Methods are stored as a part of the object itself and not as a prototype

Prototypes

- Every object in JavaScript has a prototype
- Prototypes contain all the methods for that object type

Date prototype
getDay( )
getMonth( )
etc ...

Array prototype
sort( )
filter( )
etc ...

The advantage of prototye is that pototype contains list of metods say for an array common methods are sort, filter etc. So instead of keeping methods in each array object, it is kept in prototype instead of the object itself which is accesable by all the array objects saving a lot of memory

```
function User(username, email){
  this.username = username;
  this.email = email;
  this.login = function(){
    console.log(`${this.username} has logged in`)
  }
}
```

```
function User(username, email){
  this.username = username;
  this.email = email;
}

User.prototype.login = function(){
  console.log(`${this.username} has logged in`)
}
```

This creates a method in the prototype instead of the object

This is what exactly is happening under the hood when we use classes

Built in Object

```
> new Object()
<· ▼{}
    ▼__proto__:
      ▶constructor: ƒ Object()
      ▶hasOwnProperty: ƒ hasOwnProperty()
      ▶isPrototypeOf: ƒ isPrototypeOf()
      ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶toLocaleString: ƒ toLocaleString()
      ▶toString: ƒ toString()
      ▶valueOf: ƒ valueOf()
      ▶__defineGetter__: ƒ __defineGetter__()
      ▶__defineSetter__: ƒ __defineSetter__()
      ▶__lookupGetter__: ƒ __lookupGetter__()
      ▶__lookupSetter__: ƒ __lookupSetter__()
      ▶get __proto__: ƒ __proto__()
      ▶set __proto__: ƒ __proto__()
```

Each and every method in JS is inherited fom this basic object

So every object contains its own methods in the prototype along with the above methods

Storing Data

- Websites work with data
  - blog posts, todos, comments, user info, scores etc

- We can store this data in a database
  - Firebase (by Google)

Storing Data

SQL databases

- Tables
- Rows
- Columns

NoSQL databases

- Collections
- Documents
- Properties

Firebase

NoSQL Databases

Database Instance

Data Collection

Blogs

Comments

Users

Todos

Pizzas

Blogs Collection

- document (xgdfer)
- document (fjdfghi)
- document (lophka)
- document (wiqum)
- document (zxtsvfl)

```
Document {
  title: 'ninjas!',
  likes: 50,
  author: 'ryu'
}
```

Each collection contains many documents

BY SHIVKARAMAN

Linking to Firebase

```
<script src="https://www.gstatic.com/firebasejs/5.9.1/firebase.js"></script>
<script>
  // Initialize Firebase
  var config = {
    apiKey: "AIzaSyA0WswXetYgLJ_ieH1wuejNprljdyiMfK4",
    authDomain: "udemy-modern-javascript.firebaseapp.com",
    databaseURL: "https://udemy-modern-javascript.firebaseio.com",
    projectId: "udemy-modern-javascript",
    storageBucket: "udemy-modern-javascript.appspot.com",
    messagingSenderId: "40056642736"
  };
  firebase.initializeApp(config);
</script>
```

Copied from the firebase website
This loads all the libraries of the firebase ie the whole firebase package

```
index.cdn.ts:18
It looks like you're using the development build of the Firebase
JS SDK.
When deploying Firebase apps to production, it is advisable to
only import
the individual SDK components you intend to use.

For the CDN builds, these are available in the following manner
(replace <PACKAGE> with the name of a component - i.e. auth,
database, etc):

https://www.gstatic.com/firebasejs/5.0.0/firebase-<PACKAGE>.js
```

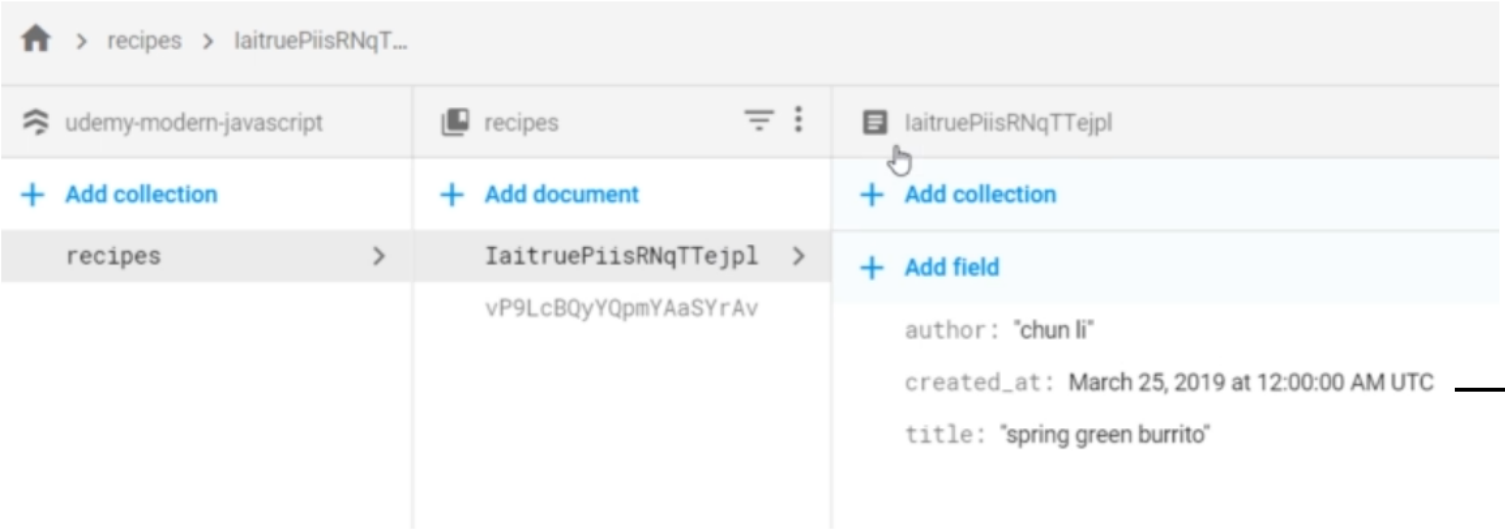
We see this error in the browser coz we dont actually need the whole firebase package now
We just need the
firestore and firebase-app libraries for now

```
<script src="https://www.gstatic.com/firebasejs/5.9.1/firebase-app.js"></scri
<script src="https://www.gstatic.com/firebasejs/5.9.1/firebase-firestore.js">
```

```
<script src="https://www.gstatic.com/firebasejs/5.9.1/firebase-app.js"></scri
<script src="https://www.gstatic.com/firebasejs/5.9.1/firebase-firestore.js">
<script>
  // Initialize Firebase
  var config = {
    apiKey: "AIzaSyA0WswXetYgLJ_ieH1wuejNprljdyiMfK4",
    authDomain: "udemy-modern-javascript.firebaseapp.com",
    databaseURL: "https://udemy-modern-javascript.firebaseio.com",
    projectId: "udemy-modern-javascript",
    storageBucket: "udemy-modern-javascript.appspot.com",
    messagingSenderId: "40056642736"
  };
  firebase.initializeApp(config);
  const db = firebase.firestore();
```

Now db has a reference to Data base

GETTING COLLECTIONS



This is not just a simple date and time, it is a timestamp object of database

Gives reference to 'recepies' collection

used to retrieve the data from the collection

This rerurns a promise

```
db.collection('recipes').get().then((snapshot) => {
  // when we have the data
  snapshot.docs.forEach(doc => {
    console.log(doc.data())
  })
}).catch(err => {
  console.log(err);
});
```

Gives reference to all the documens of colllection

Add Documents

```
// add documents
form.addEventListener('submit', e => {
  e.preventDefault();

  const now = new Date();
  const recipe = {
    title: form.recipe.value,
    created_at: firebase.firestore.Timestamp.fromDate(now)
  };

  db.collection('recipes').add(recipe).then(() => {
    console.log('recipe added');
  })
```

Creating a date object to know the time of creating of document

As we already know that document in database contains **timestamp** object, we need to create a timestamp object from using current time and date

Adds the Object into database as a document in the collection 'recipe'

NOTE : JS OBJECTS ARE STORED AS DOCUMENTS IN THE DATABASE SO WE NEED TO PASS JS OBJECTS TO 'add' FUNCTION

DELETING A DOCUMENT

In order to delete a document, we need to know the id of the collection.
As we have set the id of each document in 'data-id' attdibute of ,<li> tag we can get it from there

```
<li data-id="6j9AN4DdxA7WPwFTr2bm">
```

```
// deleting data
list.addEventListener('click', e => {
  //console.log(e);
  if(e.target.tagName === 'BUTTON'){
    const id = e.target.parentElement.getAttribute('data-id');
    db.collection('recipes').doc(id).delete().then(() => {
      console.log('recipe deleted');
    });
  }
});
```

Deletes the document with this id

```javascript
const list = document.querySelector('ul');
const form = document.querySelector('form');

const addRecipe = (recipe, id) => {
  let time = recipe.created_at.toDate();
  let html = `
    <li data-id="${id}">
      <div>${recipe.title}</div>
      <div>${time}</div>
      <button class="btn btn-danger btn-sm my-2">delete</button>
    </li>
  `;

  list.innerHTML += html;
}

// get documents
db.collection('recipes').get().then(snapshot => {
  // when we have the data
  snapshot.docs.forEach(doc => {
    addRecipe(doc.data(), doc.id);
  });
}).catch(err => {
  console.log(err);
});

 // add documents
 form.addEventListener('submit', e => {
   e.preventDefault();

   const now = new Date();
   const recipe = {
     title: form.recipe.value,
     created_at: firebase.firestore.Timestamp.fromDate(now)
   };

   db.collection('recipes').add(recipe).then(() => {
     console.log('recipe added');
   }).catch(err => {
     console.log(err);
   });

 });
// deleting data
list.addEventListener('click', e => {
  //console.log(e);
  if(e.target.tagName === 'BUTTON'){
    const id = e.target.parentElement.getAttribute('data-id');
    db.collection('recipes').doc(id).delete().then(() => {
      console.log('recipe deleted');
    });
  }
});
```

As we can clearly see
This is the section responsible to update the html page according to data present in the database.
But since this is present in the beginning of the JS page, this runs initially and when we add the document or delete a document from the database, database is updated instantly but its not visible to us as HTML IS NOT CHANGED

So the changes made to database is visible in the html only when we refresth the page

TO OVERCOME THIS WE USE REALTIME LISTNERS

Realtime Listners

This is used to keep the database and UI in sync

The realtime listner actively listens to any change in database and helps update UI accordingly

```
// get documents
db.collection('recipes').onSnapshot(snapshot => {
  console.log(snapshot.docChanges());
});
```

Actively listens to any change in database

What this does it it lists the changes in database in comparision with prev snapshot

```
▼(3) [{…}, {…}, {…}]
  ▶0: {type: "added", doc: t, oldIndex: -1, newIndex: 0}
  ▶1: {type: "added", doc: t, oldIndex: -1, newIndex: 1}
  ▶2: {type: "added", doc: t, oldIndex: -1, newIndex: 2}
   length: 3
  ▶__proto__: Array(0)
```

This is the first snapshot -> type : "added" => means these documents are added to database

```
// get documents
db.collection('recipes').onSnapshot(snapshot => {
  snapshot.docChanges().forEach(change => {
    const doc = change.doc;
    if(change.type === 'added'){
      addRecipe(doc.data(), doc.id);
    } else if (change.type === 'removed'){
      deleteRecipe(doc.id);
    }
  })
});
```

GIVES THE DOCUMENT

BY SHIVKARAMAN

```
this.chats
  .where('room', '==', this.room)
  .onSnapshot(snapshot => {
    snapshot.docChanges().forEach(change => {
      if(change.type === 'added'){
        // update the ui
        callback(change.doc.data());
      }
    });
  });
```

without this finction when we attach a real time listener, we get a list of every document on which changae has happened.
This method helps us create a condition based upon which documents are listed

Here in this example,
Only those documents whose 'room' peoperty == this.room will be added to list

FIREBASE DOESNT HAVE ===

```
this.chats
  .where('room', '==', this.room)
  .orderBy('created_at')
  .onSnapshot(snapshot => {
    snapshot.docChanges().forEach(change => {
      if(change.type === 'added'){
        // update the ui
        callback(change.doc.data());
      }
    });
  });
```

When we get data from the database, it is not in any order.

If we want to order the data in wrt any of the property of the doocument we use **.orderBy('field_name')** method

```
const list = document.querySelector('ul');
const form = document.querySelector('form');

const addRecipe = (recipe, id) => {
  let time = recipe.created_at.toDate();
  let html = `
    <li data-id="${id}">
      <div>${recipe.title}</div>
      <div><small>${time}</small></div>
      <button class="btn btn-danger btn-sm my-2">delete</button>
    </li>
  `;

  list.innerHTML += html;
};

const deleteRecipe = (id) => {
  const recipes = document.querySelectorAll('li');
  recipes.forEach(recipe => {
    if(recipe.getAttribute('data-id') === id){
      recipe.remove();
    }
  });
};

// real-time listener
db.collection('recipes').onSnapshot(snapshot => {
  console.log(snapshot.docChanges());
  snapshot.docChanges().forEach(change => {
    const doc = change.doc;
    if(change.type === 'added'){
      // console.log(doc);
      addRecipe(doc.data(), doc.id)
    } else if (change.type === 'removed'){
      deleteRecipe(doc.id);
    }
  });
});
```

BY SHIVKARAMAN

UNSUBSCRIBE FROM CHANGES

We had real-time listener to listen to any change in data-base using which we update the UI

This is called unsubscribing from changes

Now if we dont want to update UI whenever we change the database , we do the following

```
const unsub = db.collection('recipes').onSnapshot(snapshot => {
  snapshot.docChanges().forEach(change => {
    const doc = change.doc;
    if(change.type === 'added'){
      addRecipe(doc.data(), doc.id);
    } else if (change.type === 'removed'){
      deleteRecipe(doc.id);
    }
  })
});

// unsub from database changes
button.addEventListener('click', () => {
  unsub();
  console.log('unsubscribed from collection changes');
});
```

```
db.collection('recipes').onSnapshot(snapshot => {
```

⇩

This returns a function which wehn called, THE ABOVE FUNCTION IS NOT INVOKED ie the real-time listener is not invoked

When no of arguements that we will pass is not sure, we use this
The arguements sent to the function wuill be bundled up as an array

```
// rest parameter
const double = (...nums) => {
  console.log(nums);
  return nums.map(num => num*2);
}

const result = double(1,3,5,7,9,2,4,6,8);
console.log(result);
```

```
▸ (9) [1, 3, 5, 7, 9, 2, 4, 6, 8]        sandbox.js:3
▸ (9) [2, 6, 10, 14, 18, 4, 8, 12, 16]   sandbox.js:8
```

```
// spread syntax (arrays)

const people = ['shaun', 'ryu', 'crystal'];
const members = ['mario', 'chun-li', ...people];
console.log(members);
```

```
                                          sandbox.js:14
▸ (5) ["mario", "chun-li", "shaun", "ryu", "crystal"]
```

This has a similar syntax like REST PARAMETER but this is used to **spread** an array into individual components

In this example, all elements of 'people' are put in 'members' along with other elements of 'members'

SPREAD SYNTAX IN USE

```
// spread syntax (objects)

const person = { name: 'shaun', age: 30, job: 'net ninja' };
const personClone = person;
```

This doesnot create a copy of person
This just creates a new reference variable pointing to people

```
const personClone = {...person}
```

Now this creates a brand new object with same values as that of person instead of pointer to same object

BY SHIVKARAMAN

Sets

```
// sets
const namesArray = ['ryu', 'chun-li', 'ryu', 'shaun'];
console.log(namesArray);

//const namesSet = new Set(['ryu', 'chun-li', 'ryu', 'shaun']);
const namesSet = new Set(namesArray);
console.log(namesSet);

const uniqueNames = [...namesSet];
console.log(uniqueNames);
```

Sets are normal containers in which duplicate elements are not allowed

CONTAINER WHICH CONTAINS ONLY UNIQUE ELEMENTS -> SETS

Creates a new array from set

```
ages.add(20);
```

```
ages.delete(25);
```

```
ages.size
```

```
ages.has(30)
```

```
ages.clear();
```