1. Arrow Functions:

Arrow functions allows a short syntax for writing function expressions.

You don't need the function keyword, the return keyword, and the curly brackets.

Example:

```
// ES5

var x = function(x, y) {

  return x * y;

}

// ES6

const x = (x, y) => x * y;
```

JavaScript Promises:

Promises are used for asynchronous execution

A Promise is a JavaScript object that links "Producing Code" and "Consuming Code".

"Producing Code" can take some time and "Consuming Code" must wait for the result.

Promise Syntax

```
const myPromise = new Promise(function(myResolve, myReject) {

// "Producing Code" (May take some time)

  myResolve(); // when successful

  myReject();  // when error

});

// "Consuming Code" (Must wait for a fulfilled Promise).

myPromise.then(

  function(value) { /* code if successful */ },

  function(error) { /* code if some error */ }

);

Example Using a Promise

const myPromise = new Promise(function(myResolve, myReject) {

  setTimeout(function() { myResolve("I love You !!"); }, 3000);

});
```

```
myPromise.then(function(value) {
  document.getElementById("demo").innerHTML = value;
});
```

Hoisting is JavaScript's default behavior of moving declarations to the top.

JavaScript only hoists declarations, not initializations.

To avoid bugs, always declare all variables at the beginning of every scope.

"use strict"; Defines that JavaScript code should be executed in "strict mode".

Strict mode makes it easier to write "secure" JavaScript.

You can use strict mode in all your programs. It helps you to write cleaner code, like preventing you from using undeclared variables.

"use strict" is just a string, so IE 9 will not throw an error even if it does not understand it.

Strict mode is declared by adding "use strict"; to the beginning of a script or a function.

```
"use strict";
x = 3.14;       // This will cause an error
```

The Difference Between **call() and apply()**

The difference is:

The call() method takes arguments separately.

The apply() method takes arguments as an array.

```
const person = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}
const person1 = {
  firstName:"John",
  lastName: "Doe"
}
const person2 = {
  firstName:"Mary",
  lastName: "Doe"
}

// This will return "John Doe":
person.fullName.call(person1);
```

Declared at the beginning of a script, it has global scope (all code in the script will execute in strict mode):

```
const person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + "," + city + "," + country;
  }
}
const person1 = {
  firstName:"John",
  lastName: "Doe"
}
person.fullName.apply(person1, ["Oslo", "Norway"]);
```

With the **bind()** method, an object can borrow a method from another object.

```
const person = {
  firstName:"John",
  lastName: "Doe",
  display1: function() {
    let x = document.getElementById("demo");
    x.innerHTML = this.firstName + " " + this.lastName;
  }
}
let display = person.display1.bind(person);
setTimeout(display, 3000); //output - John Doe
```

A closure is a function having access to the parent scope, even after the parent function has closed.

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

Js Currying:

Currying is when a funciton - instead of taking all arguments at one time - takes the the first one and returns a new functions,

which takes the second one and returns a new functions.

currying is checking method to make sure that you get everything you need before you proceed.

it helps you to avoid passing the same variable again and again.

It divides your function into multiple smaller functions that can handle one responsibility

```
const add = (a, b, c) =>{
return a+b+c;
}
console.log(add(2,3,4));
```

converting into currying:

```
const addCurry = (a) => {
return (b) =>{
return (c) =>{
return a+b+c;}}}
console.log(addCurry(2)(3)(5));
```

Uses of currying function

a) It helps to avoid passing same variable again and again.

b) It is extremely useful in event handling.

Currying is an advanced technique of working with functions. It’s used not only in JavaScript but in other languages as well.

Write little code modules that can be easily reused and configured.

Class:

JavaScript Classes are templates for JavaScript Objects.

A JavaScript class is not an object.

It is a template for JavaScript objects.

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
}
let myCar1 = new Car("Ford", 2014);
```

A class created with a class **inheritance** inherits all the methods from another class:

Use the "**extends**" keyword to inherit all methods from another class.

Use the "super" method to call the parent's constructor function.

I have a Ford, it is a Mustang

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return 'I have a ' + this.carname;
  }
}
```

```
class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a ' + this.model;
  }
}
let myCar = new Model("Ford", "Mustang");
document.getElementById("demo").innerHTML = myCar.show();
```

Getters and Setters

Classes also allows you to use getters and setters.

It can be smart to use getters and setters for your properties, especially if you want to do something special with the value before returning them, or before you set them.

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  get cnam() {
    return this.carname;
  }
  set cnam(x) {
    this.carname = x;
  }
}

let myCar = new Car("Ford");
```

```
document.getElementById("demo").innerHTML = myCar.cnam;
```

JavaScript Object Constructors

```
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}
```

In the example above, function Person() is an object constructor function.

Objects of the same type are created by calling the constructor function with the new keyword:

```
const myFather = new Person("John", "Doe", 50, "blue");
```

Prototype Inheritance:

it is the blueprint for the objects

All JavaScript objects inherit properties and methods from a prototype.

The Object.prototype is on the top of the prototype inheritance chain:

```
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyecolor;
}
```

To add a new property to a constructor, you must add it to the constructor function:

```
Person.prototype.nationality = "English";
Person.prototype.name = function() {
  return this.firstName + " " + this.lastName;
};
```

A Denial of Service (DoS) attack is a malicious attempt to affect the availability of a targeted system, such as a website or application,

Ensure High Levels of Network Security

Continuous Monitoring of Network Traffic

Look Out for the Warning Signs

Common signs of a DDoS are:

Poor connectivity.

Slow performance.

High demand for a single page or endpoint.

Crashes.

https://webdesign.tutsplus.com/tutorials/javascript-debounce-and-throttle--cms-36783

Debounce:

debouncing says it is a practice for improving browser performance by skipping a few repetitive executions of a piece of code during a given timeframe.

debounce function makes sure that your code is only triggered once per user input. Search box suggestions, text-field auto-saves,

and eliminating double-button clicks are all use cases for debounce

```
button.addEventListener('click', debounce(function() {
    alert("Hello\nNo matter how many times you" +
      "click the debounce button, I get " +
      "executed once every 3 seconds!!")
            }, 3000));
```

debounce calls a function when a user hasn't carried out an event in a specific amount of time,

while throttle calls a function at intervals of a specified amount of time while the user is carrying out an event

Throttle is useful for cases where the user is carrying out a smooth or continuous event such as scrolling or resizing.

throttle as a minimizing function; it minimizes the number of calls made within a certain time interval.

Defining the logic for a throttle, we have:

Initialize a variable to detect if the function has been called within the specified time

If the function has been called, pause the throttle function

If the function hasn't been called or is done running in the interval, rerun the throttle function

For example, if we debounce a scroll function with a timer of 250ms (milliseconds), the function is only called if the user hasn't scrolled in 250ms.

If we throttle a scroll function with a timer of 250ms, the function is called every 250ms while the user is scrolling.

Classes:

ES6 classes make it simpler to create objects, implement inheritance by using the "extends" keyword and also reuse the code efficiently.

```
class UserProfile {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  getName() {
    console.log(`The Full-Name is ${this.firstName} ${this.lastName}`);
  }
}
let obj = new UserProfile('John', 'Smith');
obj.getName(); // output: The Full-Name is John Smith
```

Modules:

We can use the "import" or "export" statement in a module to import or export variables, functions, classes or any other component from/to different files and modules

```
export var num = 50;
export function getName(fullName) {
  //data
};
import {num, getName} from 'module';
console.log(num); // 50
```

ES6 features: https://www.boardinfinity.com/blog/top-10-features-of-es6/

Multi-line Strings

Users can create multi-line strings by using back-ticks(`).

```
let greeting = `Hello World,
            Greetings to all,
            Keep Learning and Practicing!`
```

Default Parameters

```
//ES6
let calculateArea = function(height = 100, width = 50) {
   // logic
}
//ES5
var calculateArea = function(height, width) {
  height =  height || 50;
  width = width  || 80;
  // logic
}
```

Template Literals - string templates along with placeholders for the variables

```
let name = `My name is ${firstName} ${lastName}`
```

Destructuring Assignment: easy to extract values from arrays, or properties from objects, into distinct variables.

```
//Array Destructuring
let fruits = ["Apple", "Banana"];
let [a, b] = fruits; // Array destructuring assignment
console.log(a, b);
```

```
//Object Destructuring
let person = {name: "Peter", age: 28};
let {name, age} = person; // Object destructuring assignment
console.log(name, age);
```

Enhanced Object Literals:

```
function getMobile(manufacturer, model, year) {
  return {
    manufacturer,
    model,
    year
  }
}
getMobile("Samsung", "Galaxy", "2020");
```

Spread Operator:

(...) allows us to quickly copy all or part of an existing array or object into another array or object

```
const numbers = [1, 2, 3, 4, 5, 6];
const [one, two, ...rest] = numbers;
```

Example of Rest operator in JavaScript:

Using the rest operator we can call a function using any number of arguments and can be accessed as an array. It allows the destruction of an array of objects.

```
function sum(...theArgs) {
  return theArgs.reduce((last, now) => {
    return last + now;
  });
}
console.log(sum(1, 2, 3, 4,5)); // expected output 15
```

Ternary Operator

The ternary operator is a simplified conditional operator like if / else.

Syntax: condition ? <expression if true> : <expression if false>

Overriding JavaScript Function

https://www.c-sharpcorner.com/blogs/override-function-inheritance-in-javascript

Javascript supports overriding but not overloading. When you define multiple functions in Javascript and include them in the file then the last one is referred by the Javascript engine.

shallow copy vs deep copy javascript:

https://dev.to/edwardluu/understanding-deep-and-shallow-copy-in-javascript-4im0

JavaScript Getters and Setters

Getters and setters allow you to get and set object properties via methods.

```
// Create an object:
const person = {
  firstName: "John",
  lastName: "Doe",
  language: "en",
  get lang() {
    return this.language;
  }
};

// Display data from the object using a getter:
document.getElementById("demo").innerHTML = person.lang;

const person = {
  firstName: "John",
  lastName: "Doe",
  language: "",
  set lang(lang) {
    this.language = lang;
  }
};
// Set an object property using a setter:
person.lang = "en";

// Display data from the object:
document.getElementById("demo").innerHTML = person.language;
```

JavaScript Callbacks

A callback is a function passed as an argument to another function

This technique allows a function to call another function

A callback function can run after another function has finished

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}

myCalculator(5, 5, myDisplayer);
```

Callback hell: When we develop a web application that includes a lot of code, then working with callback is messy. This excessive Callback nesting is often referred to as Callback hell.

https://www.javatpoint.com/es6-interview-questions

Set is the collection of new values. In Set, there shouldn't be any duplicate value. All the values should be unique.

```
var mySet = new Set();


mySet.add(1); // Set [1]

mySet.add(5); // Set [1, 5]

mySet.add(5); // Set [1, 5] --ignored
```

What is IIFEs (Immediately Invoked Function Expressions)?

It executes immediately after it's created

```
(function IIFE(){

            console.log( "Hello!" );

})();

// "Hello!"
```

JavaScript Async:JavaScript Async Functions

"async and await make promises easier to write"

```
async function myDisplay() {

  let myPromise = new Promise(function(resolve, reject) {

    resolve("I love You !!");

  });

  document.getElementById("demo").innerHTML = await myPromise;

}


myDisplay();
```

The Angular HTTP Interceptor is introduced along with the new HTTPClientModule. The Interceptor helps us to modify the HTTP Request by intercepting

it before the Request is sent to the back end.

The Interceptor can be useful for adding custom headers to the outgoing request, logging the incoming response, etc

The Angular HTTP interceptors sit between our application and the backend. When the application makes a request, the interceptor catches the request

before it is sent to the backend. By Intercepting requests,

we will get access to request headers and the body. This enables us to transform the request before sending it to the Server.

When the response arrives from the back end the Interceptors can transform it before passing it to our application.

One of the main benefits of the Http Interceptors is to add the Authorization Header to every request. We could do this manually,

but that is a lot of work and error-prone. Another benefit is to catch the errors generated by the request and log them

@angular/common/http package

https://www.tektutorialshub.com/angular/angular-httpclient-http-interceptor/

Summary

We learned how to intercept HTTP request & response using the new HttpClientModule.

The Interceptor can be useful for adding custom headers to the outgoing request, logging the incoming response, etc

Subject -> behaviour subject, replay subject, async subject

https://www.tektutorialshub.com/angular/replaysubject-behaviorsubject-asyncsubject-in-angular/

The **NgRx** library is awesome if we want to use it in larger applications

Redux + RxJS = ngRX

https://dzone.com/articles/ngrx-with-redux-in-angular

A large application that uses the NgRx library will be easy to understand for a new team member.

It is easy to follow the data flow and to debug the application.

By using Redux in an Angular application, the NgRx library becomes more robust and flexible.

If you have an application with 20 components and you want to pass data from component 2 to component 18, I think the best way is to use Redux,

not two way data binding or shared services.

Redux is a reactive state management library developed by Facebook and used in the React library.

To use Redux in the Angular framework, we can use the NgRx library. This is a reactive state management library. With NgRx, we can get all events (data) from the

Angular app and put them all in the same place (Store). When we want to use the stored data, we need to get it (dispatch) from the store using the RxJS library.

RxJS (Reactive Extensions for JavaScript) is a library based on the Observable pattern that is used in Angular to process asynchronous operations.

How it Works

When data changes, the existing state is duplicated. Then, a new object is created with the updates. In Angular, this data is treated as an RxJS Observable,

allowing us to subscribe to it from anywhere in the app. When an event is emitted,

for example, a button is clicked, the action is sent to a reducer function to convert the old state into the new state:

New Features in ECMAScript 2018

Rest Properties

ECMAScript 2018 added rest properties.

This allows us to destruct an object and collect the leftovers onto a new object:

let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };

x; // 1

y; // 2

z; // { a: 3, b: 4 }

JavaScript Asynchronous Iteration

ECMAScript 2018 added asynchronous iterators and iterables.

With asynchronous iterables, we can use the await keyword in for/of loops.

```
for await () {}
```

Promise.finally

ECMAScript 2018 finalizes the full implementation of the Promise object with Promise.finally:

```
let myPromise = new Promise();


myPromise.then();

myPromise.catch();

myPromise.finally();
```

generators in ES6:

Generator functions are written using the function* syntax.

one can choose to jump out of a function and let outer code to determine when to jump back into the function.

the control of asynchronous call can be done outside of your code

can get the next value in only when we really need it, not all the values at once. And in some situations it can be very convenient.

```
function* makeRangeIterator(start = 0, end = Infinity, step = 1) {
   let iterationCount = 0;
   for (let i = start; i < end; i += step) {
      iterationCount++;
      yield i;
   }
   return iterationCount;
}
```

Temporal Dead Zone in ES6

```
//console.log(aLet) // would throw ReferenceError


let aLet;
console.log(aLet); // undefined
```

```
aLet = 10;

console.log(aLet); // 10
```

What is the easiest way to convert an array to an object

You can convert an array to an object with the same data using spread(...) operator.

```
var fruits = ["banana", "apple", "orange", "watermelon"];

var fruitsObject = {...fruits};

console.log(fruitsObject); // {0: "banana", 1: "apple", 2: "orange", 3: "watermelon"}
```

What is the easiest multi condition checking

You can use indexOf to compare input with multiple values instead of checking each value as one condition.

```
// Verbose approach

if (input === 'first' || input === 1 || input === 'second' || input === 2) {

  someFunction();

}

// Shortcut

if (['first', 1, 'second', 2].indexOf(input) !== -1) {

  someFunction();

}
```

What are the differences between promises and observables

Some of the major difference in a tabular form

Promises	**Observables**
Emits only a single value at a time	Emits multiple values over a period of time(stream of values ranging from 0 to multiple)
Eager in nature; they are going to be called immediately	Lazy in nature; they require subscription to be invoked

Promise is always asynchronous even though it resolved immediately   Observable can be either synchronous or asynchronous

Doesn't provide any operators   Provides operators such as map, forEach, filter, reduce, retry, and retryWhen etc

Cannot be canceled   Canceled by using unsubscribe() method

What is an observable

An Observable is basically a function that can return a stream of values either synchronously or asynchronously to an observer over time. The consumer can get the value by calling subscribe() method. Let's look at a simple example of an Observable

```
import { Observable } from 'rxjs';


const observable = new Observable(observer => {

  setTimeout(() => {

    observer.next('Message from a Observable!');

  }, 3000);

});


observable.subscribe(value => console.log(value));

const arr = [10,20,30,40];

Math.max.apply(null, arr)

40
```

Writing a simpler code of Memoization function

Memoization is a programming technique that attempts to increase a function's performance by caching its previously computed results

if it finds similar function parameters received.

This is a very well-known and beneficial approach considered to improve performance.

What are the possible ways to create objects in JavaScript?

a.) **Object constructor:** The simpliest way to create an empty object is using Object constructor. Currently this approach is not recommended.

```
var object = new Object();
```

b.) Object create method: The create method of Object creates a new object by passing the prototype object as a parameter

```
var object = Object.create(null);
```

c.) Object literal syntax: The object literal syntax is equivalent to create method when it passes null as parameter

```
var object = {};
```

d.) Function constructor: Create any function and apply the new operator to create object instances,

```
function Person(name) {
 var object = {};
 object.name = name;
 object.age = 26;
 return object;
}
var object = new Person("Alex");
```

e.) Function constructor with prototype: This is similar to function constructor but it uses prototype for their properties and methods,

```
function Person(){}
Person.prototype.name = "Alex";
var object = new Person();
```

This is equivalent to an instance created with an object create method with a function prototype and then call that function with an instance and parameters as arguments.

```
function func {};

new func(x, y, z);

// **(OR)**
// Create a new instance using function prototype.

var newInstance = Object.create(func.prototype)


// Call the function

var result = func.call(newInstance, x, y, z),


// If the result is a non-null object then use it otherwise just use the new instance.

console.log(result && typeof result === 'object' ? result : newInstance);
```

f.) **ES6 Class syntax**: ES6 introduces class feature to create the objects

```
class Person {

 constructor(name) {

   this.name = name;

 }

}


var object = new Person("Alex");
```

g.) **Singleton pattern:** A Singleton is an object which can only be instantiated one time. Repeated calls to its constructor return the same instance and this way one can ensure that they don't accidentally create multiple instances.

```
var object = new function() {

  this.name = "Alex";

}
```

What is a higher order function

A Higher-Order function is a function that receives a function as an argument or returns the function as output.

```
const arr1 = [1, 2, 3];

const arr2 = arr1.map(function(item) {
  return item * 2;
});
console.log(arr2);
------------
```

Service creation

components communications

interceptors

Redux

current project architecture

JS objects, constructors and its core concepts

js currying

https://www.interviewbit.com/javascript-interview-questions/

https://codersera.com/blog/advanced-javascript-interview-questions/

https://github.com/sudheerj/javascript-interview-questions

IWAN:

Get faster, easier deployment and operation of your WAN, and faster performance using less bandwidth

To deliver enterprise networking solutions of the future

reduces WAN costs and time to deploy new services

control over the network

gives you visibility to see your network traffic

Deploy branch offices quickly

Simplify WAN management

Automation and centralized management

customers:

fedex, Arab bank, flex, JCB

----

manage multiple projects in one folder

```
ng g application backend
ng serve --project=backend
```

```
ng new angular-shop --create-application=false
ng g application backend
```

angular libraries:

angular material is angular library

Angular is ecosystem

Observables:

subject - active, special type of observables

Typescript:

extending js, superset of js,

add more features to js (for, while and others)

adds static typing

tightly coupled programming language

type checking

easily catch the erros during development(writing code)

package.json creation using npm init -y

npm i typescript

npx tsc typescript.ts -- to run

tsconfig.json to create -- npx tsc --init

type inference:

let course = 'Angular';

course = 1234 // will throw an error

union types:

```
let course: string | number = 'Angular
course =1234 // will not be the error
```

Assigning type aliases:

```
type person = {
name: string;
age:number;
}

let people: Person[];
```

diving into function and fn. types:

```
function add(a: number, b: number): number{
return a+b;
}
```

Generics:

```
function showOutput<T>(array: T[], value:T){
const newArray = [value, ...array];
return newArray;
}
showoutput([1,2,3], -1);
showoutput(['a','b','c'], 'd')
```

classes and interfaces:

class - blueprint for the objects

```
class student{
name:string;
age:number;

constructor(first: string, age: number){
this.name = first;
this.age = age;}}

const student = new student('venkat', 32);
```

interfaces:

```
interface Human {
name: string;
age: number;
greet: () => void;
}
```