The fn calculateArea of Sphere is not a pure fn as it accesses global state value ie PI value. If PI value is changed, which is outside the scope of fn, it will impact the o/p of fn.

```
let PI = 3.142857142857143;
then CalculateAreaOfSphere(10, PI);
// returns 1257.1428571428571
```

Note:— Any function that does activities other than just returning a value can be considered as impure.

→ A pure function will not have observable side effect.

→ In programming language, mutability & immutability are concepts related to possibility of state of an object being changed after its creation.

```
let buttonClickCount = 0;
function buttonClick(value) {
  buttonClickCount = value + 1;
}
buttonClick(buttonClickCount);
Console.log(buttonClickCount); // 1
```

Here we have the buttonClickCount value. The preceding function is an impure function. It receives the clickCount value & it reassigns it to value by incrementing it by 1. Here we violate mutability feature of FP.

Benefits of pure fn: — Cache.

higher order fn: — Cache.

cities are them.

Immutability — mutability means liable or subject to change or alteration. It means object's state is allowed to undergo change where as immutability means, once an object is created it can never change its state.

— Immutability ensures that it won't change its original value i.e. value of object should return the new copy of state or value after processing.

— All Primitive datatypes in JS are immutable in nature.

For eg:— strings, numbers & soon.

```
Const simpleString = "Hello world";
Const otherString = simpleString.slice(6
Console.log(simpleString); // Hello worl
Console.log(otherString);
```

Hello blank world !

0 1 2 3 4 5 6 7 8 9 10 11

O/P:— world !

Note: — JS by default does not provide anythi to make the object & array immutable.

Var myArray = [1,2,3,4,5];

myArray.Push(6); // this statement mutates the array & adds value to it.

Console.log(myArray); [1,2,3,4,5,6]

Mutating fns — Push(), Pop(), splice(), sort() etc

Non-mutating methods: map(), filter().

Recursion: A Program in which a function calls itself is called recursion & the related function is known as Recursive function.

```
fib(n){
    if(n<=1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

Note: In functional Programming, there is no loops like for, while etc. this problem. n — recursion solves

```
int fact(int n){
    if(n<=1)
        return 1; // base case
    else
        return n*fact(n-1);
}
```

Stopwatch behaviour:

```
let countdownTime = (count) => {
    if (count === 0) {
        console.log("stop");
    }
    else {
        console.log(count);
        countdownTime(count - 1);
    }
}
countdownTime(4);
```

o/p:- 4
3
2
1
stop

Referential Transparency :- Expression is called referentially transparent if it can be replaced with its corresponding value without changing program's behaviour.

Pure fns + Immutable data = Referential transparency.

Advantages of Referential transparency —

- it eliminates side effect of code.
- makes us code context-independent that means us code can run in any order & any context. it will always return the same results.

Sum fn

```
Const Sum = (x, y) => x+y;
Sum (6, Sum (10, 20));
```

Sum (6, 30) resultant 36

this expression is replaced with 36.

First class functions — The functions are also treated as values & used as any other data.

Functions can be:

- stored in a variable
- Passed as an argument to any other function.
- Returned by any other function.

Stored in a variable — Functions can be stored in 3 different ways. — Variables, objects, Array.

```
Const foo = function () {
    Console.log ("Hello world!");
};
// Invoking the function using the variable
foo();
```

→

```
Const foo = function greetWorld () {
    Console.log ("Hello world");
}
foo();
```

Storing function in an Object

```
let obj = {
    greetWorld : function() {
        console.log("hello world");
    }
}
obj.greetworld();
    // hello world.
```

Storing function in an Array

```
let arr = [ function greetworld() { console.log("hello");
arr[0]();   o/p — hello.
```

Callback — A fn passed as an argument to another fn.

```
function greet() {
    return function() {
        console.log("Hi Ranjit!");
    }
}
greet()();   o/p — Hi Ranjit;
```

here the first Parentheses will return the function & the secondone will invoke the returned function.

Drawbacks of of FP: — See Book Pg 6/130

First-class function :- A programming language is said to have first-class functions, if functions in that language are treated like other variables. So treated functions as first-class citizens.

Usage of 1st class fn —

- It can be stored as a value in a variable
- It can be returned by another function.
- It can be passed into another function as parameter.
- It can be stored in an array, queue or stack.
- It can have its own methods & property

Eg
```
Const myfun = (a, b) => {
    return (a + " " + b);
}
console.log(myfun("Akshit", "Saneja"));
```
o/p — Akshit Saneja

Ch-3

Basics of CSS pg: 43, 44. CSS Syntax → Rule.
Pg 46, 47 eg. —
Store of CSS.
table of pg-51

Styling within Pseudo class —

Object oriented JS ch-6

code → objects ⇄ functions. objects can be connected with each other

class → collection of objects → have instances / properties
name, age, height } properties
method → talk, run, jump

Properties state — current state of obj.

Instance properties :- what they have name, age, height
Instance method :- talk, run, jump.

class definition → constructor.

<script type = "text/javascript

```
class Rectangle {
    constructor() {
        console.log ("the rectangle is being created")
    }
}
let myRectangle1 = new Rectangle();
```

→ classname.

↑ object created from class

} it calls the constructor msg.

```
let myRectangle2 = new Rectangle();
```

Eg-2

```
class Rectangle {
constructor() {
    this.width = 5;
    this.height = 3;
    this.color = "blue";
  }
}
```

this → it refers to current object being created by class.

← 3 instance properties defined within constructors.

← now write myRectangle1 in console ↵

let myRectangle1 = new Rectangle( );

</script>

Note-
now make myRectangle1 = (3, 5, "blue");
↑ Instance parameters of.

class Rectangle {
constructor (_width, _height, _color) {
console.log ("the rectangle is being created");
this.width = _width;
this.height = _height
this.Color = _color;
}
getArea ( ) { return this.width * this.height }

let myRectangle1 = new Rectangle (3, 5, "blue");
Console.log (myRectangle1.getArea ( ));
↑ Instance method.

* → add this.
pointDescription ( ) {
Console.log ("I am a rectangle of ${this.width} x ${this.height}
& and I am ${this.color}");

Class getters & setters → used to define things on a class &
can be used as properties.

class Square {
constructor (width) {
this.width = width;
this.height = width; } }

```
class Square {
    constructor(_width) {
    this.width = _width;
     this.height = _width;
     }
get area() {
   ↑ name of getter.
    return this.width * this.height;

    }

}

let square1 = new Square(25);
Console.log(square1.area);
```

O/P:- 625

~~Add this to end.~~

~~square1.area~~

```
Class Square {

Constructor (_width) {
  this.width = _width;
  this.height = _width;
   }
   get area() {
   return this.width * this.height;
      }
```

```
set area(area) {
this.width = Math.Sqrt
                (area
this.height = this.width

   }

}
```

let square1 = new Square(4);
Console.log(square1.area);
square1.area(25);
Console.log(square1.width);
Console.log(square1.height);

Static methods

class Square {
constructor(_width) {
this.width = _width.
this.height = _width;
}
}

let square1 = new Square(8);
Console.log(Square1);
let square2 = new Square(9);
Lets create static member.

instances

Static equals (a,b) {
return a.width * a.height ===
b.width * b.height ;
}

class Square {

constructor (width) {

this.width = width;

this.height = width;

}

static equals (a, b) {

return a.width * a.height === b.width * b.height;

}

let square1 = new Square(8);

let square2 = new Square(8);

Console.log (Square.equals (square1, square2));

Console.log (Square.isValidDimensions (6, 6));

* → static isValidDimensions (width, height) {

return width === height;

}

if * true *

Inheritance: — generic to child class. Creates

super () calls constructor function of the parent class.

O/P — Person { name: "Jeff", age: 45 }

Programmer { name: "Dom", age: 56, yearsOfExperience: 12 }

```
class Person {
    constructor(_name, _age) {
        this.name = _name;
        this.age = _age;
    }
    describe() {
        console.log('I am ${this.name} and I am
        ${this.age} years old');
    }
}

class Programmer extends Person {
    constructor(_name, _age, _yearsOfExperience) {
        super(_name, _age);
        this.yearsOfExperience = yearsOfExperience;
    }
    function developSoftware(programmer) {
        let person1 = new Person("Jeff", 45);
        let programmer1 = new Programmer("Adam",
                                         56, 12);
        console.log(person1);
        console.log(programmer1);
        function developSoftware(programmers) {
        }
```

Parent class

Calls constructor of parent person.

* → add this -

```
code()
{
console.log('${this.name} is coding');
}
```

~~Parent.code();~~ Programmer1.code();

O/P:- Tom is coding -

Polymorphism ——— redefining same method within child class

Eg-

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  makeSand() {
    console.log("Generic Animal sand");
  }
}
const a1 = new Animal("Tom");
a1.makeSand();
```

Add this -

```
class dog extends Animal {
constructor(name) {
  super(name);
}
```

```
makeSound() {
    { console.log("woof"); }
}
const a1 = new Animal("Dem");
const a2 = new Dog("Jeff");
a1.makeSound();
a2.makeSound();
```

O/P — Generic Animal Sound.
woof!

A Constructor in Java is a special method that is used to initialize objects. the constructor is called when an object of a class is created. It can be used to set initial values for object attributes

Java Script class :- It defines blueprints for creating objects with similar properties & methods. The class name is user defined. the constructor method is used to initialize the class.

Syntax:-
```
class classname {
    Constructor (Parameter)
    {
    this.Parameter = value;
    }
}
```

→ Constructor is automatically called when a new instance of the class is created.

Eg.-
```
class emp {
    Constructor (name, age) {
        this.name = name;
        this.age = age; } }
const empl = new emp ("Akash", 25);
console.log (empl.name);
console.log (empl.age);
```

JS Engine handles both synchronous & Asynchronous code.

Asynchronous methods include - Callback, Promises & async/await

Asynchronous

Start something now and finish it later.

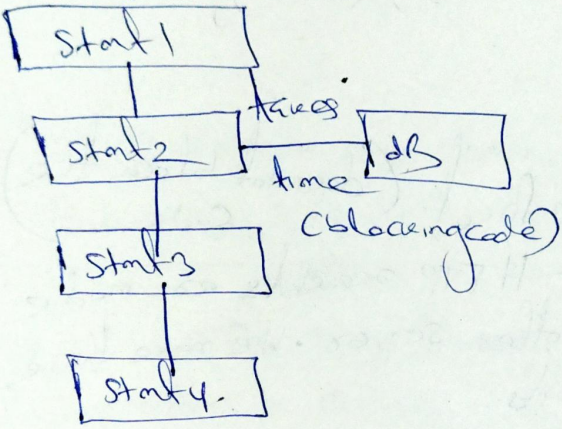
JS can run one statement at a time.

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

(synchronous) → takes time

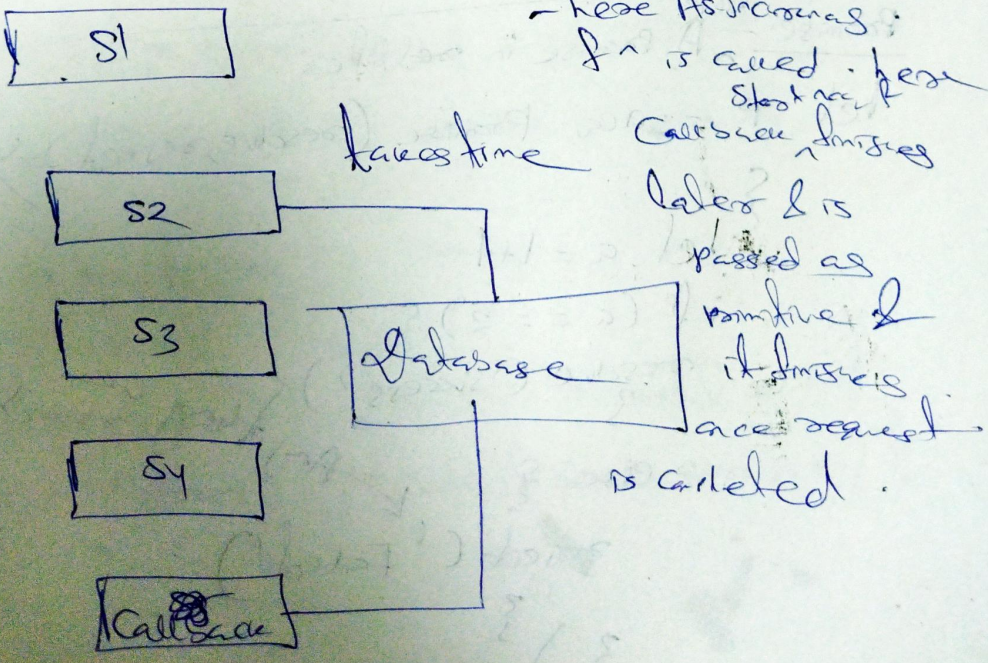
ordered sequence of steps.

(single threaded)



So need of Asynchronous code.

Asynchronous to Rescue



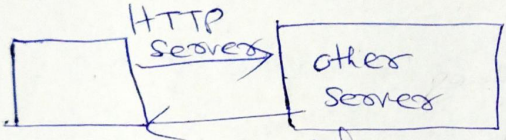
- here Asynchronous f^ is called. here Start new, here Callback triggers later & is passed as parameter & it triggers once request is completed.

```
c.log(1),
c.log(2) ---> SetTimeout(()=>{
c.log(3),
c.log(4);

                Console.log("Callback fn fired");
                },2000);
```

O/P:- 1
2
3
4
Callback function fired (does not block the code)

HTTP requests :— HTTP requests are made to get data from another servers. We make these requests to API end points.



Promise:— A Promise in real life (like)

```
let p = new Promise((resolve,reject)=>
{
  let a = 1+1
  if (a == 2) {
    resolve('success')
    }
    else {
      reject('Failed')
    }
})
```

(Any thing u can reject)

```
p.then ( (message) => {
    // it will run the resolve.
        console.log ('This is in the then' +
                                    message)
    }).catch ((message) => {
        // any error in rejected case.
    console.log ('This is in Catch' + message)
    })
```

→ P.then is called when promise is resolved successfully.
& .catch is called to catch the error when it is rejected.

Async :- The keyword async before a function makes the function return a Promise

Eg. —

```
async function myFunction() {
return "Hello"; }   is same as:
    function myFunction() {
    return Promise.resolve ("Hello");
        }
```

here is how to use the promise:

```
myFunction().then(
function (value) { /* code if successful */ },
function (error) { /* code if some error */ }
);
```

Await syntax — This can only be used inside an async function. It makes function pause the execution and wait for a resolved promise before it continues.

```
let value = await promise;
```

Note: — The two arguments : resolve and reject are predefined by JS. We will not create them, but call one of them when the executor function is ready. Most often we will not need a reject function.

Arrow function — introduced in ES6.

Allows us to write shorter function syntax.

Arrow functions Return value by default —

```
hello = () => "Hello world";
```

Arrow function with Parameters : —

```
hello = (val) => "hello" + val;
```

In case of only one parameter, u can skip the parentheses.