let var const

```
var someVal; // this variable is hoisted and prints undefined
var val1 = "123";
let val2 = "456";
const val3 = "789";

val1 = "asdas";
console.log(val1);

val2 = "new val";
console.log(val2);

// val3 = "new text"; // const does not allow for reassignment
// console.log(val3);
```

var has no block scope

```
if (true) {
// block scope starts
var temp = "value";
console.log("temp", temp);
let temp2 = "val";
// block scope ends
}

//var is visible through the block
console.log("outside block", temp);
// console.log("outside block let ", temp2);

function sayHi() {
// function scope starts
var newTemp = "Hello";
console.log("inside fn", newTemp);
// function scope ends
}
sayHi();
// console.log("outside fn, ", newTemp); // variables not accessible
```

HOISTING

```
function abc() {
// var declarations are processed when the
// function starts ( or script starts at global level)
var msg;
console.log("message variable", msg);
// console.log(msg2);
msg = "Hello";  // earlier this was var msg = ‘Hello’
let msg2 = "some text";
}
abc();

console.log("someval", someVal);
// console.log("someval2", someVal2);
```

let and const variables are hoisted but they are not accessible and cannot be used before the declarations

```
var someVal = "value";
let someVal2 = "value";
```

PROTOTYPES

# 1. COMPLIMENTARY DISHES

```
const obj = {}; // object literal syntax
console.log(obj.toString());
// const obj2 = new Object()

const arr = [1, 2, 3];
console.log(arr.join("->"));
```

Prototype is an object that has methods and properties that gets attached to our object

```
},
```

Prototypes are the mechanism by which JS objects inherits methods and properties from other objects

#2 Mom's Snacks

```
const user = {
name: " Simran",
};
console.log(user); // lookout for Object Prototype

const arr2 = [4, 5, 6];
const arr3 = new Array();
console.log(arr2); // lookout for Array prototype
```

#3 Family is Given but Friends are Chosen

```
const animal = {
eat: true,
sleep: true,
walk() {
console.log("the animla walk");
return "yay";
},
};
console.log(animal);

 const rabbit = {
 areCute: true,
 };
 rabbit.__proto__ = animal; // pointing prototype to a custom object
 console.log(rabbit);
```

[Protypal inhertitance]

```
// console.log(rabbit.walk());

const herbivore = {
eatMeat: "naah",
};

const carnivore = {
eatMeat: "yesss",
__proto__: animal,
};

herbivore.__proto__ = animal;

const rabbit = {
canJump: true,
__proto__: herbivore,
};

const tiger = {
canKill: true,
__proto__: carnivore,
};

console.log(tiger.eatMeat);
console.log(rabbit.eatMeat);
console.log(rabbit.dance);
// Prototype chain
console.log(rabbit.__proto__.__proto__.__proto__.__proto__);

// Prototypes hold either an object or a null value
```

#4 . God Element

```
const newArr = [1, 2, 3];
console.log(newArr); // Everything is JS derives from Object prototype
console.log(newArr.toString());

// Object constructor function

function User(name) {
```

```
this.name = name;
}
console.log(User.prototype);
const user2 = new User("Virat");
const user3 = new User("MSD");
console.log(user2);
console.log(user3);

let animal = {
eat: true,
sleep: true,
walk() {
console.log("animal walk");
},
};

function Rabbit(name) {
this.name = name;
}

Rabbit.prototype = animal; // setting prototype to a custom object

const rabbit = new Rabbit("Bruno");
console.log(rabbit.walk()); // now rabbit can access methods in
animal
```

```
// adding methods to prototypes

function User(name) {
this.name = name;
this.msg = function () {
console.log("hello");
};
}

const user1 = new User("Virat");
const user2 = new User("MSD");

console.log(user1);
console.log(user1.msg === user2.msg); // this is false as every
instance creates a separate copy of the functions

// memory wastage
// DRY

function BetterUser(name) {
this.name = name;
}

const userObj = {};
```

```
userObj.someProperty = "asd"; // adding custom methods and //
properties on object is same as adding some methods or properties //
on an object

console.log("userObj", userObj);

BetterUser.prototype.msg = function () { // adding method on prototype
console.log("Hello");
};


console.log("BetterUser", BetterUser.prototype);

const betterUser1 = new BetterUser("Shubman");
console.log("better user 1 ", betterUser1);
const betterUser2 = new BetterUser("Jadeja");

console.log(betterUser1.msg == betterUser2.msg);
// the above is true because both instance now point to the same
//function or memory reference


// Primitives
// new String()
console.log("scaler".toUpperCase());
console.log(String.prototype);
```

```
const str = "hello";
//str.crazyMethod("->") // O/p : s->c->a->

String.prototype.crazyMethod = function (pattern) {
return this.split("").join(pattern);
};

console.log("scaler".crazyMethod("->"));
console.log("scaler".crazyMethod(" :) "));
console.log("scaler".crazyMethod(" <3 "));
console.log(str.crazyMethod(" <3 "));
```

Learn about these beow

```
// __ proto__
// Object.create()
// Object.getPrototypeOf(rabbit) rabbit.__proto__
// Object.setPrototypeOf(rabbit, {}) rabbit.__proto__ = {}
```

Quiz

```
function Rabbit() {}
Rabbit.prototype = {
eats: true,
};
let rabbit = new Rabbit(); // protoype was attached to rabbit
```

```
Rabbit.prototype = {};
console.log(rabbit.eats); // Output ?

// //Options
// // 1. True - ans.
// // 2. Undefined
// // 3. Error

// // quiz 2
// let animal = {
// jumps: null,
// };
// let rabbit = {
// __proto__: animal,
// jumps: true,
// };

// console.log(rabbit.jumps); // O/P 1
// delete rabbit.jumps;

// console.log(rabbit.jumps); // O/P 2

// delete animal.jumps;
// console.log(rabbit.jumps); // O/P 3
```