****

The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

**Array Destructuring**

**Example (With Destructuring)**let Me = ['Chandan', 'Kumar', 20];  
let [fName, lName, Age, Branch="CSFE"] = Me;  
console.log(fName);  
console.log(Branch);

**Output:**   
Chandan  
CSFE

**Example (Without Destructuring)**let Me = ['Chandan', 'Kumar', 20];  
let fName = Me[0];  
let lName = Me[1];  
let Age = Me[2];   
Me.push("CSFE")

console.log(fName);  
console.log(Me[3]);

**Output:**   
Chandan  
CSFE

**Object Destructuring**

**Example (With Destructuring)**const Me = {  
 fName:"Chandan",  
 lName:"Kumar",  
 Age:20  
}  
let {fName, lName, Age, Branch="CSFE"}= Me;

console.log(fName);  
console.log(Branch);

**Output:**   
Chandan  
CSFE

**Example (Without Destructuring)**const Me = {  
 fName:”Chandan”,  
 lName:”Kumar”,  
 Age:20  
}

console.log(fName);

**Output:**   
Chandan

**Object Properties**

We can now use **Dynamic Properties** using **[]**,

**Example:**

let myName = "Chandan";  
const Me = {  
[myName]:`Hey, ${myName} How are you!`,  
age:`My Age is ${[10+10]}`,  
[10+10]:`Is my Age`  
}  
console.log(Me);

**Output:**{20: 'Is my Age', Chandan: 'Hey, Chandan How are you!', age: 'My Age is 20'}

No need to write Key and value, **if both are same**,

**Example: //with writting**

let **myName**= "Chandan";  
let **age** = 20;  
const Me = {  
myName: **myName**,  
age: **age**  
}  
console.log(Me);

**Output:**{myName: 'Chandan', age: 20}

**Example: //without writting**

let myName = "Chandan";  
let age = 20;  
const Me = {   
myName,  
age   
}  
**//** We can also write it as, **const Me = {myName,age}** which is exactly same as **Object Destructuring**  
console.log(Me);

**Output:**{myName: 'Chandan', age: 20}

**Spread Operators**

**#1- Example: (Without Spread Operators)**

const colors = ['red', 'blue', 'green'];  
**// After some time, I realise I need an array having 2 more colors along with the above colors**

const favColors = ['red', 'blue', 'green', 'black', 'yellow'];

console.log(favColors);

**Output:**['red', 'blue', 'green', 'black', 'yellow']

**#1- Example: (With Spread Operators)**

const colors = ['red', 'blue', 'green'];  
**// After some time, I realise I need an array having 2 more colors along with the above colors**

const favColors = […colors, 'black', 'yellow']; **//…colors (there shouldn’t be spaces)**

console.log(favColors);

**Output:**['red', 'blue', 'green', 'black', 'yellow']

**Rest Operators**

Rest Operator is an improved way to handle function parameter, allowing us to more easily handle various input as parameters in a function.  
The rest Operator syntax allows us to represent an indefinite number of arguments as an array.

**#1-Example (Without Rest Operators i.e. introduced in ES5):**

((a,b,c,d,e)=>{   
**//Second version of *Anonumous Function* i.e. *FatArrow Function* which here act as a *CallBack Function* because it is under *Immediately Invoked Function***

console.log(a+b+c+d+e);

})  
(1,2,3,5,6);

**Output:**17

**#2-Example (With Rest Operators i.e. introduced in ES6):**

((...a)=>{   
**// …a shouldn’t be anyspace and a is an arbitrary variable. (…is Rest Operator)**

let total = 0;  
for (let i of a) **//using for of loop**  
total += i;

console.log(total);

})   
(1,2,3,5,6);

**Output:**17

**Q.1. Find the Output:**

((...a)=>{   
**// …a shouldn’t be anyspace and a is an arbitrary variable. (…is Rest Operator)**

console.log(a);  
console.log(a);

let total = 0;  
for (let i of a) **//using for of loop**  
total += i;

console.log(total);

})   
(1,2,3,5,6);

**Output:**[1, 2, 3, 5, 6]  
 1 2 3 5 617

**Q.2. Find the Output:**

((b, c, ...a)=>{

console.log(`${b} ${c}`);

console.log(a);

console.log(a[0]);

console.log(a.length);

console.log(a.indexOf(2));

})

(1,2,3,5,6);

**Output:**1 2  
[3, 5, 6]  
 33  
-1