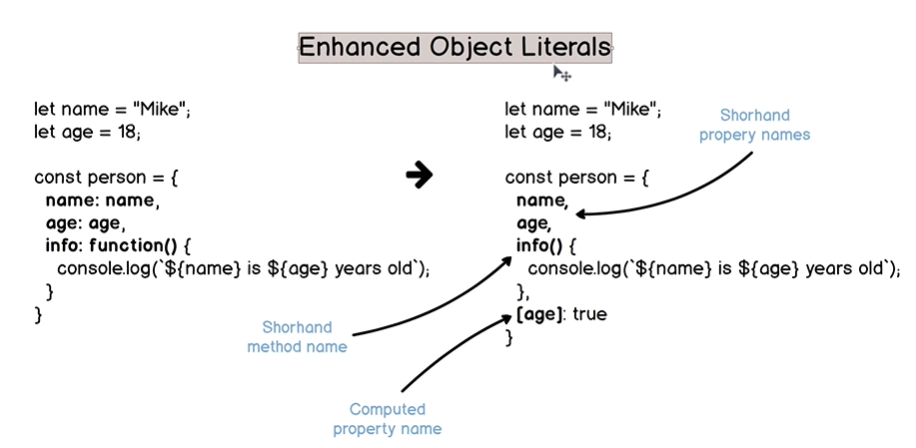
**Section 18**: ES6 Enhanced Object Literals



Enhanced object literals combine several features introduced in ES6.

These are shorthand property names, shorthand method names and computed property names.

Using computed property name, [age], a new property/key-value pair “18”: true will be added to the object.

18 is coerced to a string.

**Lecture 226**:

Object literal

Each property can hold value of any javascript variable type.

Javascript has 6 primitive types and 1 reference type, Object.

Shorthand method names do not work with arrow functions.

const num = 10

const str = "Hello world"

const myObject = {

num,

str,

info(){

return `${num} and ${str}`

}

}

const myCar = (make, year) => {

return {

make,

year,

info(price) { // shorthand method name

return `${make} costs ${price} $`

}

}

}

Cannot do this: info(price) **=>** {

We cannot shorten arrow function if it is value of a specific object property.

Traditional functions have own ‘this’ but arrow functions do not.

const BG\_COLOR\_PROP = "bgColor"

const btn = {

title: “Button 1”

[BG\_COLOR\_PROP] : "blue"

}

The second property name will be bgColor.

btn.bgColor

btn[BG\_COLOR\_PROP]

const list = {

options = ["Options 1", "Options 2"]

[BG\_COLOR\_PROP] : "grey"

}

This can help to update property names of objects.

**Lecture 228**:

Why “publishInfo” method still has access to the parameters of the “photoGallery” function ?

Because of Closure

const photoGallery = (title, dimension, date) => {

return {

title,

dimensions,

date,

[dimensions]: true

info(){

`Dimensions of the photo ${title} are ${date}`

},

publishInfo() {

console.log(`Photo ${title} was published ${Math.floor((new Date().getTime() - date.getTime())/1000 )} seconds ago`)

}

}

}

const photo1 = photoGallery("My dog", "1920x1080", new Date())

const testDimension1 = "1920x1080"

const testDimension2 = "1080x720"

photo1.info()

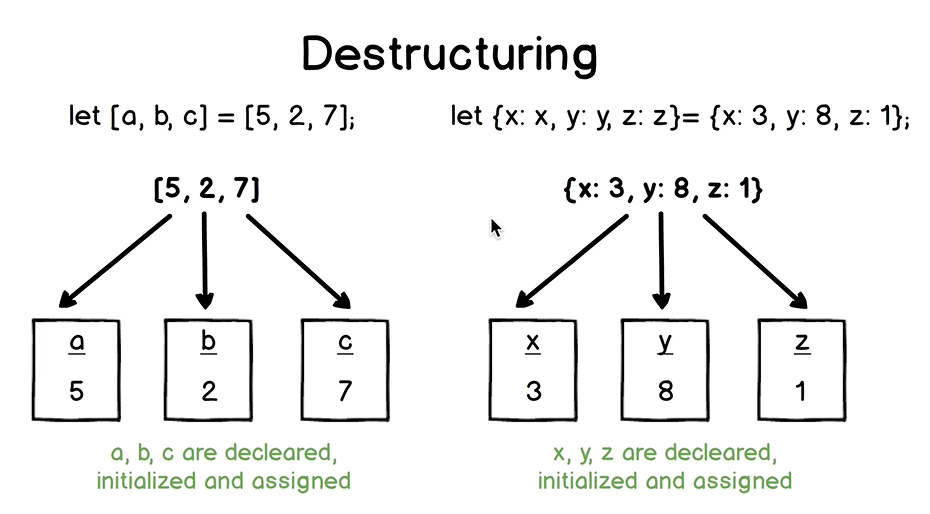
setTimeout( () => photo1.publishInfo(), 2000)

console.log(photo1[testDimension1])

console.log(photo1[testDimension2])

**Section 19**: ES6 Array and Object Destructuring

**Destructuring**:



Destructuring does not mutate original arrays and objects.

In array destructuring, order of elements is important.

In case of object literals, we can change the order of elements on the left hand side of the assignment.

We can combine default values and destructuring with both array and object destructuring.

// Example: More variables than elements in the array.

const myArray = ["a", "b"]

const [a, b, c] = myArray

console.log(a, b, c)

// c will be undefined.

// Example: Default values in array destructuring

const myArray = ["a"]

const [a, b, c = "c"] = myArray

console.log(a, b, c)

// This will print “a”, undefined and “c”

// Example: Skip elements during destructuring

const myArray = [1, 2, 3, 4, 5]

const [ , , a, , b] = myArray

console.log(a, b) // 3 5

// Rest operator in array destructuring

const myArray = [1, 2, 3, 4, 5]

const [a, b, ...c] = myArray

console.log(a, b, c)

// c will be an array with 3 elements.

If we use … to assign value to a variable which has …, then … acts as rest operator.

const d = […c]

In this case, … acts as spread operator.

// Example: Delete first element

let myArray = [1, 2, 3]

[ , ...myArray] = myArray

console.log(myArray) // [2, 3]

shift() method also deletes the first element.

myArray.shift()

let myArray = [1, 2, 3]

[ , ...anotherArray] = myArray

console.log(anotherArray) // [2, 3]

// myArray is not changed.

// Example: Swap values using destructuring

let x =5 , y=10

[y, x] = [x, y]

console.log(x, y) // 10 5

// Example: Destructuring in the function

const myPosts = [

["Post 1", 10],

["Post 2", 20]

]

myPosts.forEach( ( [title, likes] ) => console.log(`${title} has ${likes} likes`) )

Here destructuring is used directly in the parameters section.

// Example: Nested array destructuring

const myArray = [ 1, 2, [3, 4] ]

const [a, b, [c, d]] = myArray

console.log(a, b, c, d)

// 1, 2, 3, 4

**Lecture 233**:

Can use default value for the function parameter when passed in array does not have a value for that parameter.

**Lecture 235**:

const minMax = (...nums) => {

return [

nums.reduce( (min, num) => num < min ? num : min),

nums.reduce( (max, num) => num > max ? num : max)

]

}

let min, max

[min, max] = minMax(24, 5, 34, 10)

We can have implicit return above instead of explicit return.

If we return an object from a function then () are required.

In case of Array, they are not required.

const minMax = (...nums) => **[**

nums.reduce( (min, num) => num < min ? num : min),

nums.reduce( (max, num) => num > max ? num : max)

**]**

We have covered array destructuring so far.

**Lecture 236**: Object Destructuring

In object destructuring, order of elements is not important.

We can rename properties and can also use default values with object destructuring.

// Example: Declaration and assignment using object destructuring

const myObject = {

a: 10,

b: true

}

// const a = myObject.a

// const b = myObject.b

const { a: a, b: b } = myObject

console.log(a, b) // 10 true

// Example: Declaration and assignment using object destructuring and shorthand property names

const { a, b } = myObject

// Example: Assignment using object destructuring and shorthand property names

const myObject = {

a: 10,

b: true

}

let a, b

{ a, b } = myObject

console.log(a, b)

This gives an error. Here javascript engine interprets { } as a block and not as an object literal.

We can put this assignment in parentheses.

( { a, b } = myObject )

// Example: Destructure non-object value

const { a, b } = undefined

console.log(a, b)

// Uncaught TypeError: Cannot destructure property `a` of ‘undefined’ or ‘null’

We cannot perform destructuring when on the right hand side there is undefined or null.

To avoid getting this error when trying to destructure a variable which could be undefined or null, we can do the following:

const val = undefined

const {a, b} = val **||** {}

console.log(a, b) // undefined undefined

Here we will not get any error.

const myArray = [1, 2]

const {c, d} = myArray

console.log(c, d) // undefined undefined

The above expression in Javascript is fully correct.

const {length, d} = myArray

console.log(length, d) // 2 undefined

Each array has in-built property called as ‘length’.

// Example: Change order of the properties

const myObject = {

a: 10,

b: true

}

const { b, a } = myObject

console.log(a, b) // 10 true

When we destructure an object, then order of properties in the object literal is not important because we refer to the properties in the object by their names.

// Example: Rest operator in object destructuring

const myObject = {

a: 10,

b: true,

c: [],

d: "abc",

e: 20

}

const { a, b, ...rest } = myObject

console.log(a, b, rest)

‘rest’ will be an object with 3 key-value pairs, c, d and e.

const { a, d, ...rest } = myObject

console.log(a, d, rest)

// Here ‘rest’ will be an object with 3 key-value pairs b, c and e.

Rest element must be the last element.

“Object” is the prototype of myObject.

If we try to add a property to Object, then it will be added to all the variables which have Object as their prototype.

Object.prototype.newProp = 1;

myProp.newProp gives 1.

“rest” variable will not contain the properties of the prototype of myObject.

// Example: Destructure non-existing properties.

const myObject = {

a: 10,

b: true

}

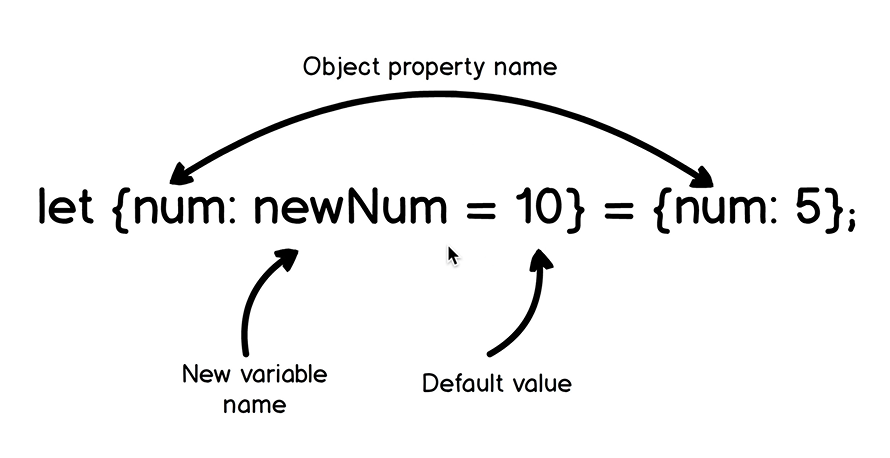
const { a, b, c } = myObject

console.log(a, b, c) // 10 true undefined

If the property ‘c’ is absent in the object, then its value is undefined.

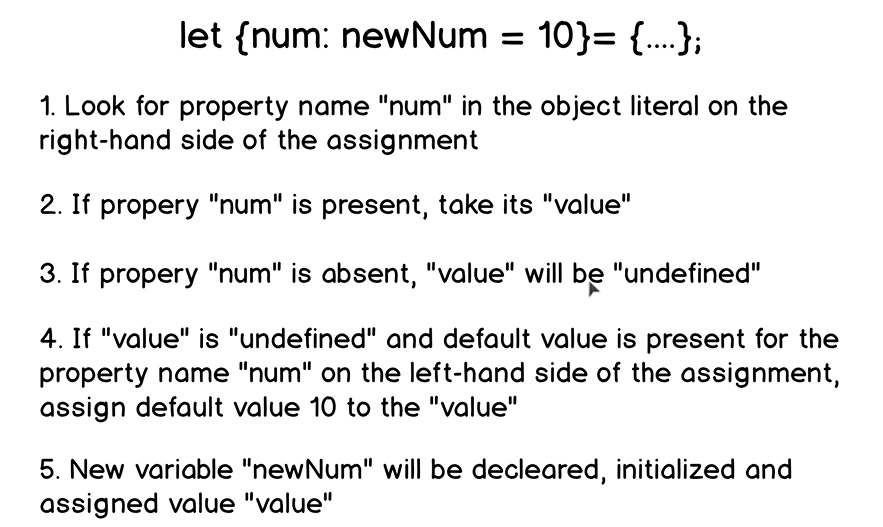
**Lecture 237**:

New variable names and default values in Object destructuring



“num” on the left and right side are property names.

“newNum” is the new variable name declared using “let” keyword and 10 is the default value.



const myObject = {

a: 10,

b: true

}

const { a, b, c = "default value" } = myObject

console.log(a, b, c) // 10 true “default value”

const myObject = {

a: 10,

b: true

}

const {

a,

b = "default value",

c = "default value"

} = myObject

console.log(a, b, c) // 10 true “default value”

const myObject = {

a: 10,

b: true

}

const {

a: newA,

b: newB

} = myObject

console.log(a, b)

Uncaught ReferenceError: a is not defined

// Example: Default values and new variable names

const myObject = {

a: 10,

b: true

}

const {

a: newA,

b: newB,

c: newC = "default value"

} = myObject

console.log(newA, newB, newC)

// 10 true “default value”

// Example: Nested Object destructuring

const myObject = {

a: 1,

b: 2,

nestedObject: {

c: 3,

d: 4

}

}

const {a: a, b: b, nestedObject: nestedObject} = myObject

console.log(a, b, nestedObject)

// 1 2 {c: 3, d: 4}

const { c: c, d: d } = nestedObject

console.log(a, b, c, d)

const {

a: a,

b: b,

nestedObject: { c: c, d: d }

} = myObject

console.log(a, b, c, d)

// 1 2 3 4

const {

a,

b,

nestedObject: { c, d }

} = myObject

console.log(a, b, c, d)

We can also easily use new variable names and default values when we perform nested object destructuring.

const {

a,

b,

nestedObject: { c, d: newD }

} = myObject

console.log(a, b, c, newD)

“d” is the property of the object, so we cannot use it standalone in place of ‘newD’.

**Lecture 240**:

When using shorthand property names, value name is equal to the property name.

Property name is always on the left while variable name is always on the right.

const personInfo = (

{

name: name,

age: personAge,

location: { country: origin, city: homeCity },

friendsQty = 0,

recordCreatedAt = new Date().getFullYear()

} ) => {

return {

name,

personAge,

origin,

homeCity,

friendQty,

recordCreatedAt

}

}

const person = {

name: "Alice",

age: 19,

location: {

country: "England",

city: "London"

}

}

console.log(personInfo(person))

**Lecture 242**: Object destructuring with array helper methods.

const posts = [

{ postId: 234, author: "robd", commentsQty: 5 },

{ postId: 823, author: "sady" },

{ postId: 161, author: "merryl", commentsQty: 8 }

]

const processPosts = posts => {

return posts.map( ( // Object destructuring

{

postId,

author: postAuthor,

commentsQty: postCommentsQty = 0

} )

=> {

({

postAuthor,

postCommentsQty,

postId: postId + 1000

})

} )

}

console.log(processPosts(posts))

**Lecture 244**: Delete Object properties

We have to mutate the object.

These tasks are common in real world applications.

Suppose we receive an object from the database.

let person = {

\_id = "12343",

index: 4,

processed: false,

cart: ["item1", "item2", "item3"],

email: "abc@gmail.com",

name: "abc",

cartId: 912

}

console.log(person)

/\*

We need to return the following:

{

index: 4,

email: "abc@gmail.com",

name: "abc",

cartId: 912

}

\*/

We can use delete operator individually for properties to be deleted.

We can also achieve this using object destructuring.

If we do this:

let { \_id, processed, cart, ...person } = person, then ‘person’ variable will be re-declared again.

With let, we cannot declare same variable again in the same scope.

The following does the job:

let \_id, processed, cart

( { \_id, processed, cart, ...person } = person )

“person” is re-assigned.

If we try to do this:

let person = {

\_id = "12343",

index: 4,

processed: false,

cart: ["item1", "item2", "item3"],

email: "abc@gmail.com",

name: "abc",

cartId: 912

}

{

let { \_id, processed, cart, ...person } = person

// This throws an error of

Uncaught ReferenceError: person is not defined

}

console.log(person)

Here the js engine does not look for ‘person’ variable in the global scope because there is block scoped local variable ‘person’.

‘person’ variable in the inner scope is in the temporal dead zone.