**Section 16**: ES6 Template Literals

**Lecture 202**:

Pre-ES6- Template Strings

Traditional template strings

const myNumber = 10;

const myStr = "Hello";

const myArr = [1, 2];

const myObj = {x: 10}

const myBool = true

const templateStr = "Number is " + myNumber + " and string is " + myStr +

" and array is " + myArr + " and object is " + myObj + " and boolean is " + myBool

console.log(templateStr)

Here we use ‘+’ operator with strings and different variable types.

Each variable value is converted to string. This is called coercion.

Variables are coerced to string.

Pair of backticks represent template literals. **``**

Template literals do not introduce any new functionality.

Result of using template literals is the same as with using template strings.

Everything that must be interpretated by javascript as a javascript expression must be embedded in curly braces { } and we must add $ sign before it.

${..} will be interpretated as javascript expression.

const templateStr = `Number is + ${myNumber} and string is

${myStr} and array is ${myArr} and object is ${myObj} and boolean is ${myBool}`

Inside template literals, we can use any valid Javascript expression.

We can use a function call, IIFE, etc.

We have to embed the expression inside **${ }**

Another difference with template strings and template literals is with handling of carriage returns.

To add a newline in template strings, we have to use \n rather than just using ‘enter’ key.

We can split template literals into several lines using ‘enter’ key.

Using template literals, we can construct html in string format.

**Lecture 203**: Tagged Templates

Each template literal can be parsed using a tagged function.

const name = "Sophia"

const age = 20

console.log(**`**${name} is ${age} years old**`**)

This template literal is passed to the Javascript engine.

Javascript engine evaluates each expression in the template literal. Then result of each expression is coerced to string.

Javascript engine concatenates those strings.

Under the hood, there is default tag function that processes each template literal.

We can create custom tag functions.

We can then parse this template literal using custom tag function.

const checkAge = (arrayOfStrings, name, age) =>{

console.log(arrayOfStrings, name, age)

}

console.log(checkAge`${name} is ${age} years old`)

For arrayOfStrings, we will get an array of 3 elements.

0: “”

1: “ is ”

2: “ years old”

These represent strings that are already in string format in the template literal.

First one is the empty string located before the first expression.

If we parse a template literal using tag function,

First argument to checkAge function will be this array of strings.

2nd argument will be the result of the first expression.

And so on.

If we have a variable at the start or the end, then in the arrayOfString, first string or last string will be an empty string respectively.

For example,

console.log(checkAge`${name} is ${age} years old ${2+2}`)

Have to define checkAge with one more parameter.

checkAge is a tag function.

The first argument which is the array has a property called as “**raw**”.

This property is not present in a normal array.

This property also contains an array similar to the array elements.

If we add ‘\n’ in our template literal, then in the normal array it will be shown as carriage return (symbol of enter key) and in the ‘raw’ property it is shown as \n.

Each array in javascript is an Object. And thus it can have any properties it wants.

In this example, one more named property is added to the array.

We can add any property to any array and it won’t break anything.

Default tag function evaluates result of each expression, then coerces those results to strings, concatenates those strings and returns resulting string.

Built in default tag function parses the template literal and returns the result.

Number of other parameters will be equal to the length of the arrayOfStrings -1

**Lecture 205**:

const templateLiteral = num => **`**Number is ${num}

This number is ${num >=10 ? "greater" : "less"} than 10.

Square root of this number is ${Math.sqrt(num)}**`**

Any expression which has to do with operations with a variable is used in ${}

Behind the scenes, this template literal is parsed using default tag function.

**Lecture 207**:

1. Get all arguments from the function call, because the number of arguments can be variable.

Extract all results of the expressions to the separate array. “vals”

2. Loop through array of strings. “arrayOfStrings”

3. “arrayOfStrings”.length - 1 = “vals”.length

4. Concatenate elements in “arrayOfStrings” and “vals”.

(We have to iterate both of the arrays simulatenously concatenate the elements).

“vals” is 1 length smaller than “arrayOfStrings”. So when we reach the last element of “arrayOfString”, we are already done with “vals”.

For this, we can use reduce() helper method.

How can we pull out all arguments from a function, keeping in mind that we don’t about the number of expressions in the template literal?

We require arguments from position 2 because first argument will be “arrayOfStrings”.

Array.from(arguments)

We will delete first element from this array.

For this, we can use .slice() method.

.slice(1) deletes the first element and returns remaining number of elements in a new array.

Here, we cannot use arrow function.

const a =10; const b =5;

const sum = taggedTemplate`Sum of the two variables

a(${a}) and b(${b}) is ${a+b}`

function taggedTemplate(arrayOfStrings) {

const vals = Array.from(arguments).slice(1)

return arrayOfStrings.reduce(

(concatStr, str, index) => {

return concatStr + str + (vals[index] !== undefined ? vals[index] : "" )

}

, "")

}

taggedTemplate function works similar to the default in built tag function.

We can also convert the following function into an arrow function.

const taggedTemplate = (arrayOfStrings, ...vals) => {

return arrayOfStrings.reduce(

(concatStr, str, index) => {

return concatStr + str + (vals[index] !== undefined ? vals[index] : "" )

}

, "")

}

REST parameters (ES6 feature).

**Section 17**:

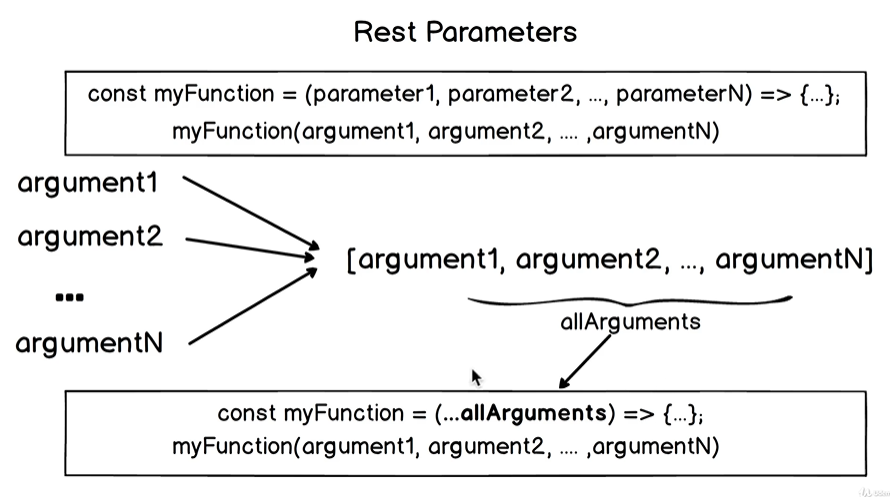
ES6 Rest/Spread Operators and Default Function Parameters

**Lecture 209**:

… (3 dots), depending on its usage, it is either called Rest operator or Spread operator.

Rest parameters

If … is used before a parameter name of a function, then it is called rest operator.



Separate parameter of a function is limited to its scope. And each time, when function will be called, brand new variable is created for each parameter.

If a function has 5 parameters and we call it with 3 parameters, then this means that the last 2 parameters in the function will be undefined.

If a function has 3 parameters and we pass in 5 arguments, then we would not be able to access last 2 arguments because we have defined only 3 parameters in the function.

In the anonymous arrow function, we would not be able to pull arguments using “arguments” variable.

In anonymous arrow function, we do not have access to ‘arguments’ variable.

If we use …, then all the passed in arguments are stored in an array. This array will be assigned to the parameter name.

const myFunction = (a, b, …restArguments) => {…}

restArguments is the rest parameter.

Rest operator can be used only at the end of the parameter list.

If we use rest parameter, then we must have parentheses.

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

console.log(nums)

return nums.filter(num => typeof num == "number")

.reduce((sum, num) => sum + num, 0)

}

console.log(sumNumbers("abc", 2, 3, 4))

Rest operator gathers elements in a single array.

**Lecture 212**: Spread Operator

Spread operator works opposite to rest operator.

It takes an array and spreads them into separate elements.

If we apply spread operator to an array with 5 elements, then we will get 5 separate elements.

Spread operator can be used in a function call.

If a function takes 3 arguments and we have an array with 3 elements, then we can pass this array with the spread operator and these 3 elements will be spread into 3 arguments of the function call.

Spread operator can be used in place of concat() function.

We can construct new arrays from other arrays using just spread operator.

const myGreeting = (name, city, age) => {

return `Hello from ${name} who is ${age} years old and lives in ${city}`

}

const myPerson = ["Alice", "Boston", 20] ;

console.log(myGreeting(...myPerson))

// Concatenate Arrays

We can chain concat() methods.

concat() method does not mutate the arrays.

const a = [1, 2, 3]

const b = [4, 5]

const c = a.concat(b)

console.log(c) // [1, 2, 3, 4, 5]

const d = [ ...a, ...b, 6 ]

console.log(d) // [1, 2, 3, 4, 5, 6]

Use spread operator to call built-in date() function.

Suppose we want to create a data string using an array.

const dateInfo = [2025, 5, 10];

const date = new Date(...dateInfo)

console.log(date)

Date will be June 10 2025

2nd parameter in the Date() function is month.

January starts from 0.

// **Copy Array**

const myArray = ["a", 5, [ ], true]

const newArray = [...myArray]

console.log(newArray)

“newArray” is a shallow copy of myArray.

Shallow copy only copies one level deep.

const myArray = ["a", 5, [], true]

const newArray = **[**...myArray**]**

newArray.push(10) // myArray is mutated

newArray[2].push(7) // myArray is not mutated

console.log(newArray) // ["a", 5, [7], true, 10]

console.log(myArray) // ["a", 5, [7], true]

All inner levels are copied as usually in shallow copy.

Only pointer is copied.

If we change a reference type element, then it will be changed in both the arrays.

Object variable type is a reference type in Javascript.

// **Copy Object** (this feature arrived in Ecmascript 2018)

const myObject = { a: 10, b: "c", d:[1, 2] }

const copiedObject = **{**...myObject**}**

copiedObject.a = 20 // myObject is not mutated

copiedObject.d.pop() // myObject is mutated

console.log(copiedObject)

console.log(myObject)

Here also, there is shallow copy.

Result is similar to the example of Copy Object.

All inner values are shallow copied.

**Example 6**: Spread String

We can apply spread operator to the string.

In Javascript, spread operator can be applied to any iterable element.

String is iterable, array is iterable.

Iterable elements have built in symbol iterators.

const myStr = "Hello World"

const letters = [...myStr]

console.log(letters)

// [“H”, “e”, …. ]

Single quotes and double quotes behave in exactly the same way in Javascript.

To spread a string into character array, we can also use split(“”) function on the string.

split() is a String helper method.

String.prototype.split

Spread operator can only split String in distinct characters and cannot split on a particular character such as space, etc.

**Lecture 214**:

Calculate mean of the given array.

To calculate the mean, we can use reduce() helper method.

const meanScore = (...scores) => {

if(scores.every(score => typeof score === "number")){

return scores.reduce( (avg, score) => {

return avg + score / scores.length

}, 0).toFixed(2)

} else {

throw new Error("Supplied arguments must only contain numbers")

}

}

To round off numbers, we can use toFixed() function.

Number.prototype.toFixed

toFixed() takes in number of digits after the decimal point as an argument.

toFixed() method returns String.

To convert this result to “number”, we can use unary ‘+’ operator.

Javascript takes a string and coerces it to a number.

Alternatively, we can use parseFloat() function.

const meanScore = (...scores) => {

if(scores.every(score => typeof score === "number")){

return parseFloat(scores.reduce( (avg, score) => {

return avg + score / scores.length

}, 0).toFixed(2))

} else {

throw new Error("Supplied arguments must only contain numbers")

}

}

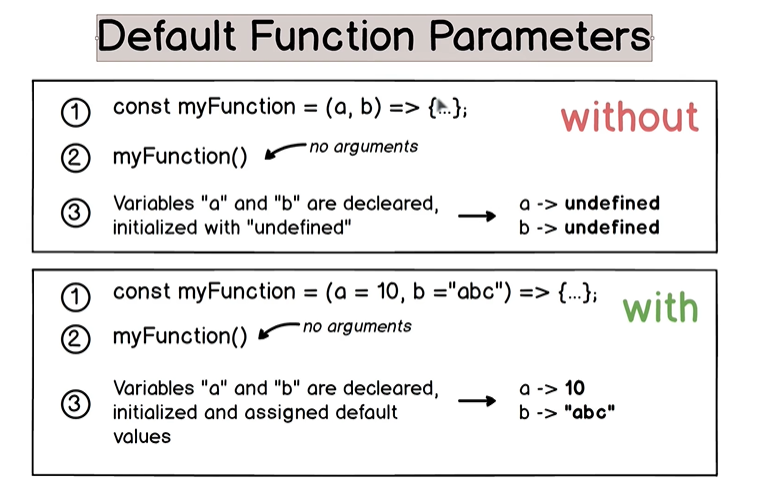
return **+**scores.reduce( (avg, score) => {

return avg + score / scores.length

}, 0).toFixed(2)

In Javascript, there is only one number type called “number”.

**Lecture 216**: Default Function Parameters



We can use default function parameters in both arrow and regular function expressions.

If we don’t supply a value to a function parameter, it is initialized as ‘undefined’.

const multiplier = (num=2, mult) => { console.log(num\*mult)

}

multiplier(3) // this gives NaN as 3 is passed in as the value for parameter ‘num’.

const groceryItem = (title, qty, item) => {

item = {

itemTitle: title,

itemQty: qty

}

console.log(title, qty, item)

}

groceryItem("Banana", 5) // 2 arguments in the function call.

We can also specify default value of ‘item’

For the default values, we can refer to previous parameters.

‘title’ and ‘qty’ will get declared, initialized and assigned before assigning value of item parameter.

**IMP**:

Suppose there is a case where we need to pass in a value for a parameter, else it will lead to an Exception.

const warning = () = {

throw new Error("btn function must be called with 2 arguments - title and color")

}

const btn = (title = warning(), color = warning()) => {

console.log(title)

console.log(color)

}

btn("Button 1", "grey")

btn("Button 2") // Exception

warning() is used when no value is specified for that particular parameter.

**Lecture 218**:

Default value without default parameters

Empty string in Javascript is treated as false value.

0, undefined and null are also treated as false value.

a || b will always return b if a is false.

const weatherForecast = (city, weather) => {

weather = weather !== undefined ? weather : "Great Weather"

return `Weather forecast for ${city}: ${weather}`

}

Here, we only treat undefined as absence for one of the parameters.

Default function parameter behaves exactly as the above assignment.

**Lecture 220-223**: Generate Missing Unique IDs

const tasks = [

{ title: “abc”, taskId: 334 },

{ title: “def”, taskId: 91 },

{ title: “xyz” }

]

Generate missing task ids automatically.

Generate 4 digit number

[1000, 10000)

Math.random(), [0, 1)

1000+ Math.floor( Math.random() \* 9000) )

map() returns a brand new array and does not mutate original array.

const tasks = [

{ title: "Meeting with John", taskId: 4621 },

{ title: "Visit Gym", taskId: 6821 },

{ title: "Buy new phone" },

{ title: "Clean the room", taskId: 2721 },

{ title: "Plan a trip" }

]

const generateTasksIds = tasks => {

return tasks.map(task => {

if(!task.hasOwnProperty("taskId")){

let newTaskId = 1000 + Math.floor(Math.random() \* 9000)

task.taskId = newTaskId

}

return task

});

}

const tasksSortedByIds = (tasks, tasksWithIds = generateTasksIds(tasks)) => {

console.log(`Quantity of the missing tasks

is ${tasks.filter(task => !task.hasOwnProperty("taskId")).length}`)

return tasksWithIds

}

To prevent mutation of the array reference, we can create a copy of the array by using … rest operator.

This prevents mutation of the original array.

const generateTasksIds = tasks => {

return tasks.map(task => {

**task = {...task}**

if(!task.hasOwnProperty("taskId")){

let newTaskId = 1000 + Math.floor(Math.random() \* 9000)

task.taskId = newTaskId

}

return task

});

}

const generateTasksIds = tasks => {

return tasks.map( **({...task})** => {

if(!task.hasOwnProperty("taskId")){

let newTaskId = 1000 + Math.floor(Math.random() \* 9000)

task.taskId = newTaskId

}

return task

});

}

One element will be represented as {…tasks}

const tasksSortedByIds = (tasks, tasksWithIds = generateTasksIds(tasks)) => {

console.log(`Quantity of the missing tasks

is ${tasks.filter(task => !task.hasOwnProperty("taskId")).length}`)

return tasksWithIds.sort((a, b) => a.taskId - b.taskId)

}

// return array of task ids in sorted order.

Bonus Task: Ensure that new unique 4-digit “taskId” is not the same as any “taskId” of the existing tasks.

do-while loop

We can only use primitive types with includes() method.

We cannot compare Javascript objects directly.

To compare JS objects, we have to use parse() or stringify() function.

const generateTasksIds = tasks => {

let newTaskIds = []

return tasks.map( ({...task}) => {

if(!task.hasOwnProperty("taskId")){

do{

newTaskId = 1 + Math.floor(Math.random() \* 9)

} while( tasks.find(task => task.taskId === newTaskId )

|| newTasksIds.includes(newTaskId) );

newTaskIds.push(newTaskId)

task.taskId = newTaskId

}

return task

} );

}