**Assignment (8 July) part-2**

**Question 6 to 10**

**Q.6 What is template literals in ES6 and how do you use them?**

**Ans. ES6 Template Literals**

Template literals are a new feature introduced in ECMAScript 2015/ ES6. It provides an easy way to create multiline strings and perform string interpolation. Template literals are the string literals and allow embedded expressions.

Before [ES6](https://www.javatpoint.com/es6), template literals were called as **template strings**. Unlike quotes in strings, template literals are enclosed by the **backtick (` `)** character (key below the **ESC** key in QWERTY keyboard). Template literals can contain placeholders, which are indicated by the dollar sign and curly braces **($(expression})**. Inside the backticks, if we want to use an expression, then we can place that expression in the **($(expression})**.

**Syntax**

1. var str = `string value`;

Multiline strings

In normal strings, we have to use an escape sequence **\n**to give a new line for creating a multiline string. However, in template literals, there is no need to use **\n**because string ends only when it gets **backtick(`)**character.

Advertisement

Let us try to understand it with the following example.

**Example**

1. // Without template literal
2. console.log('Without template literal \n multiline string');
4. // With template literal
5. console.log(`Using **template** literal
6. multiline string`);

**Output**

Without template literal

multiline string

Using template literal

multiline string

String Interpolation

[ES6 template literals](https://www.javatpoint.com/es6-template-literals) support string interpolation. Template literals can use the placeholders for string substitution. To embed expressions with normal strings, we have to use the **${}** syntax.

**Example -1**

1. var name = 'World';
2. var cname = 'javaTpoint';
3. console.log(`Hello, ${name}!
4. Welcome to ${cname}`);

**Output**

Hello, World!

Welcome to javaTpoint

Let us see another example of string interpolation.

**Example-2**

1. var x = 10;
2. var y = 20;
3. console.log(`The product of the variables ${x} and ${y} is:
4. ${x\*y}`);

**Output**

The product of the variables 10 and 20 is:

200

Tagged templates

Tagged templates are one of the more advanced form of **template literals.** Tagged template literals allow us to parse template literals with a function.

The first argument of the tag function contains an array having string values, and the remaining arguments are related to the expression. The writing of tagged literal is similar to the function definition, but the difference occurs when the tagged literals get called. There are no parentheses **()** required to call a literal.

Let us see the illustration for the tagged templates.

**Example-1**

1. function TaggedLiteral(str) {
2. console.log(str);
3. }
5. TaggedLiteral `Hello World`;

**Output**

[ 'Hello World' ]

**Example-2**

We can also pass the values in a tagged literal. The value can be the result of some expression or the value fetched from the variable. We can see the illustration for the same in the following code:

1. function TaggedLiteral(str, val1, val2) {
2. console.log(str);
3. console.log(val1+"    "+val2);
4. }
6. let text = 'Hello World';
7. TaggedLiteral`Colors ${text} ${10+30}`;

**Output**

[ 'Colors ', ' ', '' ]

Hello World 40

Raw Strings

The template literal raw method allows the accessing of raw strings as they were entered. In addition to this, the **string.raw()** method exists for creating the raw strings as similar to the default template function. It allows us to write the backslashes as we would in a regular expression literal.

The **string.raw()** method is used to show the strings without any interpretation of backslashed characters. It is also useful to print windows sub-directory locations without require to use a lot of backslashes.

**Example**

1. var rawText = String.raw`Hello \n World \n Welcome back! `
2. console.log(rawText)

**Output**

Hello \n World \n Welcome back!

String.fromCodePoint()

This method returns a string, which is created by using the specified sequence of Unicode code points. It throws a **RangeError** if there is an invalid code point is passed.

**Example**

1. console.log(String.fromCodePoint(49))
2. console.log(String.fromCodePoint(80, 76))

**Output**

1

PL

**Q.7 What’s difference between map & forEach?**

**Ans.** JavaScript .forEach() and .map(): These are the methods that are used to iterate on an array, more technically they invoke the provided callback function for every element of an array.

**Differences between forEach() and map() methods:**

|  | **forEach()** | **map()** |
| --- | --- | --- |
| **1** | The forEach() method does not returns a  new array based on the given array. | The map() method returns an entirely new array. |
| **2** | The forEach() method returns “*undefined*“. | The map() method returns the newly created array according to the provided callback function. |
| **3** | The forEach() method doesn’t return anything hence the method chaining technique cannot be applied here. | With the map() method, we can chain other methods like, reduce(),sort() etc. |
| **4.** | It is not executed for empty elements. | It does not change the original array. |

**Q.8 How can you destructure objects and arrays in ES6?**

**Ans.** Destructuring in JavaScript refers to the process of unpacking values from an array or object. It provides a more concise way of getting values from arrays or objects without heavy lifting when you're interested in individual array items or values in an object.

It's also helpful when processing returning values from a complex function or expression. This concept is one of the best practices you should follow while writing React code.

**How to Destructure Arrays**

Here's a sample code to get a sense of array destructuring:

const arr = [1, 2];

const [a, b] = arr;

console.log(a) // prints 1 on the console

console.log(b) // prints 2 on the console

This code uses destructuring to assign the values from arr—1 and 2—to the variables a and b, respectively. This is the fundamental concept behind destructuring. You have an array or object on the right-hand side, and you unpack items and assign them to individual variables on the left-hand side.

Under the hood, JavaScript copies values from arr and assigns them to the variables on the left-hand side like so:

const arr = [1,2];

const a = arr[0];

const b = arr[1];

As you can see, destructuring is a more concise way of doing this, unlike using the object or bracket notation. However, this syntax will likely be useful when working with complex arrays or functions that return arrays or strings.

**Check the sample code below:**

const [day, month, date, year, time, timezone] = Date().split(' ')

// Calling Date() returns the current date in the format:

// Mon Feb 20 2023 13:07:29 GMT+0000

console.log(day) // prints Mon

console.log(month) // prints Feb

console.log(date) // prints 20

**In this code sample**, we're creating a new string with the current date by calling Date(). Next, we use split(), a JavaScript string method, to separate items in the string using space as the separator. split(' ') returns an array, and we use destructuring to assign the values to individual variables.

Remember that if your array contains more items than you're unpacking, JavaScript will ignore the extra items.

const arr = [1, 2, 3, 4];

const [a, b] = arr;

console.log(a) // prints 1

console.log(b) // prints 2

// values 3 and 4 are not assigned to any variable; they are ignored

**In this case,** if you want to store the remaining items in a variable, use a rest parameter like so:

const arr = [1, 2, 3, 4];

const [a, b, ...rest] = arr;

console.log(rest) // prints [3,4]

Sometimes, you may not care about a specific item. JavaScript's destructuring pattern also allows you to skip certain elements by using an empty comma.

const arr = [1, 2, 3, 4];

const [a, , c] = arr;

console.log(c) // prints 3

The code above uses the empty space to ignore value 2 in the array arr. Instead of assigning value 2 to variable c, it skips to the next item in the array. It also ignores the fourth value because it doesn't specify a variable to store it in.

**In contrast**, if you need fewer items than you're unpacking, the process will assign the undefined value to those extra variables.

const arr = [1];

const [a, b] = arr;

console.log(a) // prints 1

console.log(b) // prints undefined

To prevent having variables undefined, you can set default values if you're not sure about the array length as follows (assigning default values is not a requirement):

const arr = [1];

const [a = '10', b = 'not provided'] = arr;

console.log(a) // prints 1

console.log(b) // prints "not provided"

This destructuring assigns the value 1 to variable a, overwriting its default value. Variable b keeps its default because a value is not provided.

**How to Destructure Objects**

Destructuring objects is not so different from arrays. The only difference is that arrays are iterable and objects are not, resulting in a slightly different way of doing things.

When working with objects, the variables on the left-hand side of the assignment operator are also initialized like objects:

const person = {name: 'Alvin', age: 10, height: 1};

const {name, age, height} = person;

console.log(name) // prints 'Alvin'

console.log(height) // prints 1

From the code, we're using property names from the person object. However, you don't have to use the exact property names in the object. You can destructure and store the values in different variable names as follows:

const person = {name: 'Alvin', age: 10, height: 1};

const {name: firstName, age: years, height: currentHeight} = person;

console.log(firstName) // prints 'Alvin'

console.log(currentHeight) // prints 1

You start by specifying the property value you want to destructure, then specify the variable name you will use to store the value after a colon. And like arrays, destructuring a property name that doesn't exist will be undefined.

To handle this, you can similarly specify default values in case the property name you want to assign to a variable is not available:

const person = {name: 'Alvin', age: 10, height: 1};

const {name, age, height, location='Worldwide'} = person;

console.log(name) // prints 'Alvin'

console.log(location) // prints 'Worldwide'

The order of variables on the left-hand side doesn't matter with an object since the objects store values in key-value pairs. As such, the following code will yield the same results:

const person = {name: 'Alvin', age: 10, height: 1};

const {age, height, name} = person;

console.log(name) // prints 'Alvin'

console.log(height) // prints 1

**Lastly, similar to arrays, you can also use the rest parameter to destructure several values in one variable like so:**

const person = {name: 'Alvin', age: 10, height: 1};

const {name, ...rest} = person;

console.log(name) // prints 'Alvin'

console.log(rest) // prints { age: 10, height: 1 }

Just note that the rest parameter must always come last. Otherwise, JavaScript will throw an exception.

**Q.9 How can you define default parameter values in ES6 functions?**

**Ans.** Function parameters with default values are initialized with default values if they contain no value or are undefined. **JavaScript function parameters are defined as undefined by default.** However, it may be useful to set a different default value. That is where default parameters come into play.

**Syntax:**

function name(parameter=value,...parameters) {

}

**Example 1:**If we multiply two numbers in this example without passing a second parameter and without using the default parameter, the answer that this function will return is NAN(Not a Number), since if we do not pass the second parameter, the function will multiply the first number with undefined.

* Javascript

|  |
| --- |
| **function** multiply(a, b) {  **return** a \* b;  }  let num1 = multiply(5);  console.log(num1);  let num2 = multiply(5, 8);  console.log(num2); |

**Output:**

NaN

40

**Example 2:**If we do not pass a number as the second parameter and take the default parameter as the second parameter, it will multiply the first number with the default number, and if we pass two numbers as parameters, it will multiply the first number with the second number.

* Javascript

|  |
| --- |
| **function** multiply(a, b = 2) {  **return** a \* b;  }  let num1 = multiply(5);  console.log(num1);  let num2 = multiply(5, 8);  console.log(num2); |

**Output:**

10

40

**Example 3:** Default Parameter with Constructor: we can use the default parameter concept with the constructor of a class.

* Javascript

|  |
| --- |
| class Geeks {      constructor(a, b = 3) {      console.log(a \* b);      }  }  let obj = **new** Geeks(5);  let obj1 = **new** Geeks(5, 4); |

**Output:**

15

20

**Q.10 What is the purpose of the spread operator (...) in ES6?**

**Ans.** [ES6](https://www.javatpoint.com/es6) introduced a new operator referred to as a spread operator, which consists of three dots (...). It allows an iterable to expand in places where more than zero arguments are expected. It gives us the privilege to obtain the parameters from an array.

Spread operator syntax is similar to the rest parameter, but it is entirely opposite of it. Let's understand the syntax of the spread operator.

**Syntax:**

1. var variablename1 = [...value];

The three dots (...) in the above syntax are the spread operator, which targets the entire values in the particular variable.

Spread Operator and Array Manipulation

Here, we are going to see how we can manipulate an array by using the spread operator.

Constructing array literal

When we construct an array using the literal form, the spread operator allows us to insert another array within an initialized array.

**Example**

1. let colors = ['Red', 'Yellow'];
2. let newColors = [...colors, 'Violet', 'Orange', 'Green'];
3. console.log(newColors);

**Output**

[ 'Red', 'Yellow', 'Violet', 'Orange', 'Green' ]

Concatenating arrays

Spread operator can also be used to concatenate two or more arrays.

**Example**

1. let colors = ['Red', 'Yellow'];
2. let newColors = [...colors, 'Violet', 'Orange', 'Green'];
3. console.log(newColors);

**Output**

[ 'Red', 'Yellow', 'Violet', 'Orange', 'Green' ]

Copying an array

We can also copy the instance of an array by using the spread operator.

**Example**

1. let colors = ['Red', 'Yellow'];
2. let newColors = [...colors];
3. console.log(newColors);

**Output**

[ 'Red', 'Yellow' ]

If we copy the array elements without using the spread operator, then inserting a new element to the copied array will affect the original array.

But if we are copying the array by using the spread operator, then inserting an element in the copied array will not affect the original array.

Let's understand the illustration for the same.

**Example**

**Without using spread operator**

1. let colors = ['Red', 'Yellow'];
2. let newColors = colors;
3. newColors.push('Green');
4. console.log(newColors);
5. console.log(colors);

**Output**

[ 'Red', 'Yellow', 'Green' ]

[ 'Red', 'Yellow', 'Green' ]

**Using spread operator**

1. let colors = ['Red', 'Yellow'];
2. let newColors = [...colors];
3. newColors.push('Green');
4. console.log(newColors);
5. console.log(colors);

**Output**

[ 'Red', 'Yellow', 'Green' ]

[ 'Red', 'Yellow' ]

Note: Instead of elements, the spread operator only copies the array itself to the new one. It means that the operator can do a shallow copy instead of a deep copy.

Spread operator and Strings

Let's see how the spread operator spreads the strings. The illustration for the same is given below.

**Example**

Here, we have constructed an array **str** from individual strings.

1. let str = ['A', ...'EIO', 'U'];
2. console.log(str);

In the above example, we have applied the spread operator to the string **'EIO'**. It spreads out each specific character of the **'EIO'** string into individual characters.

We will get the following output after the execution of the above code.

**Output**

[ 'A', 'E', 'I', 'O', 'U' ]