ES6 Spread Operator

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

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

Output

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

Concatenating arrays

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

Example

```
let colors = ['Red', 'Yellow'];
let newColors = [...colors, 'Violet', 'Orange', 'Green'];
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

```
let colors = ['Red', 'Yellow'];
let newColors = [...colors];
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.

Example

Without using spread operator

```
let colors = ['Red', 'Yellow'];
let newColors = colors;
newColors.push('Green');
console.log(newColors);
console.log(colors);
```

Output

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

Using spread operator

```
let colors = ['Red', 'Yellow'];
let newColors = [...colors];
newColors.push('Green');
console.log(newColors);
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' ]
```

JavaScript Promise

Promises in real-life express a trust between two or more persons and an assurance that a particular thing will surely happen. In javascript, a Promise is an object which ensures to produce a single value in the future (when required). Promise in javascript is used for managing and tackling asynchronous operations.

Need for JavaScript Promise

Till now, we learned about events and callback functions for handling the data. But, its scope is limited. It is because events were not able to manage and operate asynchronous operations. Thus, Promise is the simplest and better approach for handling asynchronous operations efficiently.

There are two possible differences between Promise and Event Handlers:

1. A Promise can never fail or succeed twice or more. This can happen only once.
2. A Promise can neither switch from success to failure, or failure to success. If a Promise has either succeeded or failed, and after sometime, if any success/failure callback is added, the correct callback will be invoked, no matter the event happened earlier.

Constructor in Promise

new Promise(function(resolve, reject){});	Here, resolve(thenable) denotes that the promise will be resolved with then(). Resolve(obj) denotes promise will be fulfilled with the object Reject(obj) denotes promise rejected with the object.

Terminology of Promise

A promise can be present in one of the following states:

1. **pending:** The pending promise is neither rejected nor fulfilled yet.
2. **fulfilled:** The related promise action is fulfilled successfully.
3. **rejected:** The related promise action is failed to be fulfilled.
4. **settled:** Either the action is fulfilled or rejected.

Thus, a promise represents the completion of an asynchronous operation with its result. It can be either successful completion of the promise, or its failure, but eventually completed. Promise uses a **then()** which is executed only after the completion of the promise resolve.

Promises of Promise

A JavaScript Promise promises that:

1. Unless the current execution of the js event loop is not completed (success or failure), callbacks will never be called before it.
2. Even if the callbacks with then() are present, but they will be called only after the execution of the asynchronous operations completely.
3. When multiple callbacks can be included by invoking then() many times, each of them will be executed in a chain, i.e., one after the other, following the sequence in which they were inserted.

```
let pro=new Promise((resolve,reject)=>{
            setTimeout(()=>{
         let shadi="i promise u ill never leave you!!!!"
            //    resolve(shadi)
               reject("gharvale nhi manenge")
            },2000)
         })
         pro.then((shadiFinal)=>{
            console.log(shadiFinal);
         }).catch((error)=>{
                console.log("gharvale nhi manenge");
         })
```

1. **Resolve:** When the promise is executed successfully, the resolve argument is invoked, which provides the result.
2. **Reject:** When the promise is rejected, the reject argument is invoked, which results in an error.

It means either resolve is called or reject is called. Here, then() has taken one argument which will execute, if the promise is resolved. Otherwise, catch() will be called with the rejection of the promise.