Rest Parameter

- Rest parameters, introduced in ES6 (ECMAScript 2015), provide a way to work with functions that receive an arbitrary number of arguments.
- Rest parameters allow you to represent an indefinite number of arguments as an array.
- They provide a way to handle function parameters more flexibly, especially when you don't know in advance how many arguments will be passed.
- They offer a clean and concise way to collect all remaining arguments in an array, making it easier to handle functions with varying numbers of parameters.

Syntax

```
function functionName(...restParams) {
  // body of function
}
```

Where,

- ...restParams is the rest parameter, which collects the remaining arguments that are passed into the function.
- The rest parameter is represented by three dots (...) followed by the name of the parameter. This creates an array that contains the remaining arguments that weren't assigned to the preceding parameters.

Example: Handling Variable Arguments

```
function display(...numbers) {
  return numbers
}
console.log(display(1, 2, 3));
console.log(display(10, 20));
```

Example: You can combine rest parameters with regular parameters

```
function display(greeting, ...names) {
  return `${greeting} ${names}`;
}
console.log(display("Hello Students - ", "Akash", "Shailja", "Ishika"));
```

Rules for Rest Parameters

- **Rest Parameters Must Be the Last Parameter**
  - You can only use one rest parameter in a function.
  - It must appear at the end of the function's parameter list. Placing it before other parameters is not allowed.

  Example

```
//correct example
function display(a, b, ...rest) {
  // rest is an array of remaining arguments
}

//in-correct example (throws an error)
function display(...rest, a, b) {
  // Invalid syntax: rest parameter should be last
}
```

- **Rest Parameters Collect All Remaining Arguments**
  - Rest parameters gather all arguments that were not explicitly named in preceding parameters into an array.

  Example

```
function display(a, b, ...rest) {
  console.log(a);      // First argument
  console.log(b);      // Second argument
  console.log(rest);   // Array of remaining arguments
}
display(1, 2, 3, 4, 5);
```

- **Rest Parameters Work as an Array**
  - The rest parameter is an actual array. This means you can use common array methods such as map(), forEach(), reduce(), and others.

  Example

```
function sum(...numbers) {
  return numbers.reduce((total, current) => total + current, 0);
}
console.log(sum(10,20,30,40));
console.log(sum(5, 10, 15));
```

How Rest Parameters Differ from the arguments Object

- Before ES6, JavaScript provided the arguments object, which could be used to access all arguments passed to a function.
- Now, rest parameters are considered a more modern and preferable alternative.

1. Array-like vs. Array

- The arguments object is array-like but not a real array. You can access its values using indices, but it does not support array methods like map(), forEach(), etc.
- Rest parameters, on the other hand, are actual arrays, making them much more flexible and easier to manipulate.

Example

```
// Using the arguments object (not an actual array)
function displayArgs() {
  console.log(arguments);
  console.log(arguments instanceof Array);  // false
}
displayArgs(10, 20, 30);  // Output: [10, 20, 30] (array-like object)
// Using rest parameters (actual array)
function displayRest(...args) {
  console.log(args);
  console.log(args instanceof Array);  // true
}
displayRest(10, 20, 30);  // Output: [10, 20, 30] (actual array)
```

2. No Named Parameters in arguments

- The arguments object includes all passed arguments, even those assigned to named parameters.
- Rest parameters only capture the remaining arguments after the named parameters have been assigned.

Example

```
function displayArgs(a, b) {
  console.log(arguments);  // Includes all arguments
}
function displayRest(a, b, ...rest) {
  console.log(rest);  // Only includes remaining arguments after a and b
}
displayArgs(11, 12, 13, 14);
displayRest(11, 12, 13, 14);
```

3. Use in Arrow Functions

- The arguments object is not available in arrow functions, but rest parameters work seamlessly with arrow functions.

Example

```
const arrowFunc = (...args) => {
  console.log(args);
};

arrowFunc(10, 20, 30);
```

Examples & Exercises

Example -- Filtering Data with Datatype

```
function filterBy(type, ...args) {
            return args.filter(e => typeof e === "number");
}
let result = filterBy(10, 'Hi', null, undefined, 20);
console.log(result);
```

Example -- Rest parameters and arrow function

```
const combine = (...args) => {
      return args.reduce((prev, curr) => prev + ' ' + curr);
};
let message = combine('JavaScript', 'Rest', 'Parameters'); // =>
console.log(message); // JavaScript Rest Parameters
```

Example -- Rest parameter in a dynamic function

```
var showNumbers = new Function('...numbers', 'console.log(numbers)');
showNumbers(1, 2, 3) // [1,2,3]
```

Spread operator

- The spread operator in JavaScript (introduced in ES6) is a powerful feature that allows you to expand or spread iterable objects (like arrays or strings) into individual elements.
- It is denoted by three dots (...) and can be used in various contexts, such as function calls, array literals, and object literals.
- The spread operator in ES6 is a versatile feature that makes it easier to manipulate arrays and objects.
- It simplifies operations like combining arrays, copying objects, and passing elements as function arguments.
- Understanding how to use the spread operator effectively can greatly enhance your ability to write concise and readable JavaScript code.
- The spread operator is used with arrays, and its syntax is exactly the same as that of the rest parameter. It is used to expand the contents of an array.
- The spread operator allows us to spread elements of an iterable object, such as an array, map, or set.
- The spread operator can be used to clone an iterable object or merge multiple iterable objects into one.

Syntax

```
const newArray = [...existingArray];
const newObject = {...existingObject};
```

Spread Operator works with

- **Arrays:** When used with an array, the spread operator takes each element of the array and spreads it out into a new array.
  Example
  ```
  var arr = [..."123456"];
  console.log(arr);
  ```

- **Objects:** When used with an object, it takes each property of the object and spreads it out into a new object.
  Example
  ```
  var obj1 = { a: 1, b: 2 };
  var obj2 = { ...obj1, c: 3 };
  console.log(obj2);
  ```

Use Cases of Spread Operator

1. **Expanding Arrays:** The spread operator can be used to expand elements of an array into a new array.
   Example
   ```
   const numbers = [1, 2, 3];
   const newNumbers = [...numbers, 4, 5]; // [1, 2, 3, 4, 5]
   ```

2. **Combining Arrays:** You can use the spread operator to combine multiple arrays into one.
   Example
   ```
   const array1 = [1, 2];
   const array2 = [3, 4];
   const combinedArray = [...array1, ...array2]; // [1, 2, 3, 4]
   ```

3. **Copying Arrays:** The spread operator provides a concise way to create a shallow copy of an array.
   Example
   ```
   const originalArray = [1, 2, 3];
   const copiedArray = [...originalArray]; // [1, 2, 3]
   ```

4. **Expanding Strings:** The spread operator can also be used with strings to create an array of characters.
   Example
   ```
   const str = "Hello";
   const charArray = [...str]; // ['H', 'e', 'l', 'l', 'o']
   ```

5. **Spreading in Function Calls:** You can use the spread operator to pass elements of an array as arguments to a function.
   Example
   ```
   function add(a, b, c) {
     return a + b + c;
   }
   const nums = [1, 2, 3];
   console.log(add(...nums)); // 6
   ```

6. **Combining Objects:** The spread operator can be used to combine properties from multiple objects into one object.
   Example
   ```
   const obj1 = { a: 1, b: 2 };
   const obj2 = { b: 3, c: 4 };
   const combinedObj = { ...obj1, ...obj2 }; // { a: 1, b: 3, c: 4 }
   ```

7. **Shallow Copying Objects:** You can create a shallow copy of an object using the spread operator.
   A **shallow copy** is a copy of an object where the top-level properties are duplicated, but the nested objects or arrays inside the original object are still referenced by the copy. This means that changes to nested objects in the copied object will affect the original object as well.
   Example
   ```
   const originalObj = { a: 1, b: 2 };
   const copiedObj = { ...originalObj }; // { a: 1, b: 2 }
   ```

8. **Adding or Modifying Properties in Objects:** You can use the spread operator to add new properties or modify existing ones while creating a new object.
   Example
   ```
   const obj = { a: 1, b: 2 };
   const updatedObj = { ...obj, b: 3, c: 4 }; // { a: 1, b: 3, c: 4 }
   ```

9. **Using with Arrays of Objects:** When working with arrays of objects, you can use the spread operator to modify specific objects.
   Example
   ```
   const users = [{ id: 1, name: 'Priya' }, { id: 2, name: 'Karan' }];
   const updatedUsers = users.map(user =>
       user.id === 1 ? { ...user, name: 'Neha' } : user
   );
   console.log(updatedUsers);
   ```

Rules of Spread Operator

The spread operator in JavaScript has several important rules and guidelines for its use. Understanding these rules can help you avoid common mistakes and make the most of its versatility.

1. **Must be Used with Iterable Objects:** The spread operator only works with iterable objects such as arrays, strings, maps, sets, or objects that have an @@iterator method.
   Example
   ```
   const array = [1, 2, 3];
   const newArray = [...array];
   ```
   Non-iterables: Objects, in general, are not iterables (unless they are made iterable with specific properties).
   But since ES2018, the spread operator works with objects, although objects themselves aren't truly iterable. JavaScript supports spreading their enumerable properties.
   Example (Object Spread)
   ```
   const obj1 = { a: 1, b: 2 };
   const obj2 = { ...obj1 }; // Object spreading is supported
   ```

2. **Spread Operator Can Only Appear in Certain Contexts:** In Array Literals: It spreads out the elements of an iterable into a new array.
   Example
   ```
   const arr1 = [1, 2];
   const arr2 = [...arr1, 3, 4]; // [1, 2, 3, 4]
   ```
   In Function Calls: It can be used to pass the elements of an array (or any iterable) as individual arguments to a function.
   Example
   ```
   function add(a, b, c) {
     return a + b + c;
   }
   const nums = [1, 2, 3];
   const sum = add(...nums); // Spreads the array, equivalent to add(1, 2, 3)
   ```
   In Object Literals (Post-ES2018): It spreads properties of an object into a new object.
   ```
   const obj = { a: 1, b: 2 };
   const newObj = { ...obj, c: 3 }; // { a: 1, b: 2, c: 3 }
   ```

3. **Cannot Use Spread with Non-Iterable Objects (Except Objects in ES2018):** The spread operator will throw an error if you try to spread something that isn't iterable in a context that requires an iterable.
   For example, using a non-iterable object in an array literal will cause an error.
   ```
   const obj = { a: 1, b: 2 };
   const arr = [...obj]; // Error: obj is not iterable
   ```
   However, you can use the spread operator with objects in an object literal context, as objects have been made "spreadable" (but not iterable) since ES2018.

   Example (Valid Object Spread)
   ```
   const obj = { a: 1, b: 2 };
   const newObj = { ...obj }; // Works fine for objects
   ```

4. **Shallow Copy with Spread Operator:** The spread operator only creates a shallow copy of arrays or objects. For arrays or objects containing primitive types (like numbers, strings, etc.), this works as expected.
   However, for nested objects or arrays of objects, the copy will only be shallow, meaning changes to nested elements will affect both the original and the copy.
   Example
   ```
   const original = [1, 2, { a: 1 }];
   const copy = [...original];
   copy[2].a = 2; // Changing the nested object affects the original as well
   console.log(original[2].a); // Output: 2
   ```
   If you need a deep copy, you'll need to use other methods like JSON.parse(JSON.stringify()) or structured cloning.

5. **Order of Properties in Object Spread:** When using the spread operator with objects, if there are duplicate properties, the one spread last takes precedence.
   Example:
   ```
   const obj1 = { a: 1, b: 2 };
   const obj2 = { b: 3, c: 4 };
   const merged = { ...obj1, ...obj2 }; // { a: 1, b: 3, c: 4 }
   ```
   The property b from obj2 overrides the one from obj1 because obj2 is spread later in the new object.

6. **Function Call with Spread Operator:** The spread operator can be used to pass an array or any iterable as arguments to a function.
   Example
   ```
   function sum(a, b, c) {
     return a + b + c;
   }
   const nums = [1, 2, 3];
   const result = sum(...nums); // Equivalent to sum(1, 2, 3)
   console.log(result); // Output: 6
   ```
   Note that the number of elements in the array must match the number of parameters the function expects, or JavaScript will pass undefined for any missing values.

7. **Spreading in the Middle of Arrays:** You can use the spread operator at any position in an array: at the beginning, middle, or end.
   Example
   ```
   const arr1 = [1, 2];
   const arr2 = [3, 4];
   const combined = [0, ...arr1, ...arr2, 5]; // [0, 1, 2, 3, 4, 5]
   ```

8. **Rest Parameter and Spread Operator are Opposites:** While both the rest and spread operators use the ... syntax, they serve opposite purposes:
   - The spread operator is used to expand an iterable into individual elements.
   - The rest parameter is used to collect multiple elements into a single array or object.

   Example (Spread and Rest)
   ```
   // Spread operator:
   const nums = [1, 2, 3];
   console.log(...nums); // Outputs: 1 2 3

   // Rest parameter:
   function add(...args) {
     return args.reduce((sum, n) => sum + n, 0);
   }
   console.log(add(1, 2, 3)); // Output: 6
   ```

9. **Cannot Spread Non-enumerable Properties of Objects:** The spread operator does not copy non-enumerable properties of objects, which means only the properties that can be iterated over are copied.
   Example
   ```
   const obj = {};
   Object.defineProperty(obj, 'hidden', {
     value: 'secret',
     enumerable: false
   });
   const newObj = { ...obj };
   console.log(newObj.hidden); // Output: undefined
   ```

Spread Operator Vs Rest Parameter

While the spread operator and rest parameter look similar (both use ...), they serve different purposes:

- **Spread Operator (...):** Used to expand or spread elements from an iterable (like an array or object) into individual elements.
  ```
  const arr = [1, 2, 3];
  const newArr = [...arr]; // Spread
  ```

- **Rest Parameter (...):** Used to collect a variable number of arguments into an array when defining a function.
  ```
  function func(...args) { } // Rest
  ```