#CODING

JavaScript Rest vs Spread Operator – What's the Difference?

spread operators. But these two operators are not the same.

The main difference between rest and spread is that the rest operator puts the rest of some specific user-supplied values into a JavaScript array. But the spread syntax expands iterables into individual elements.

For instance, consider this code that uses rest to enclose some values into an array:

```
// Use rest to enclose the rest of specific user-supplied values into an
function myBio(firstName, lastName, ...otherInfo) {
  return otherInfo;
}

// Invoke myBio function while passing five arguments to its parameters:
myBio("Oluwatobi", "Sofela", "CodeSweetly", "Web Developer", "Male");

// The invocation above will return:
["CodeSweetly", "Web Developer", "Male"]
```

Try it on StackBlitz

In the snippet above, we used the `...otherInfo` rest parameter to put `"CodeSweetly"`, `"Web Developer"`, and `"Male"` into an array.

Now, consider this example of a spread operator:

Forum Donate

Learn to code — free 3,000-hour curriculum

```
  return `${firstname} ${lastname} runs ${company}`;
}

// Use spread to expand an array's items into individual arguments:
myBio(...["Oluwatobi", "Sofela", "CodeSweetly"]);

// The invocation above will return:
"Oluwatobi Sofela runs CodeSweetly"
```

Try it on StackBlitz

In the snippet above, we used the spread operator (`...`) to spread `["Oluwatobi", "Sofela", "CodeSweetly"]` 's content across `myBio()` 's parameters.

Don't worry if you don't understand the rest or spread operators yet. This article has got you covered!

In the following sections, we will discuss how rest and spread work in JavaScript.

So, without any further ado, let's get started with the rest operator.

What Exactly Is the Rest Operator?

The **rest operator** is used to put the rest of some specific user-supplied values into a JavaScript array.

So, for instance, here is the rest syntax:

Forum Donate

Learn to code — free 3,000-hour curriculum

The three dots (`...`) in the snippet above symbolize the rest operator.

The text after the rest operator references the values you wish to encase inside an array. You can only use it before the last parameter in a function definition.

To understand the syntax better, let's see how rest works with JavaScript functions.

How Does the Rest Operator Work in a Function?

In JavaScript functions, rest gets used as a prefix of the function's last parameter.

Here's an example:

```
// Define a function with two regular parameters and one rest parameter:
function myBio(firstName, lastName, ...otherInfo) {
  return otherInfo;
}
```

The rest operator (`...`) instructs the computer to add whatever `otherInfo` (arguments) supplied by the user into an array. Then, assign that array to the `otherInfo` parameter.

As such, we call `...otherInfo` a rest parameter.

Forum Donate

Learn to code — free 3,000-hour curriculum

Here's another example:

```
// Define a function with two regular parameters and one rest parameter:
function myBio(firstName, lastName, ...otherInfo) {
  return otherInfo;
}

// Invoke myBio function while passing five arguments to its parameters:
myBio("Oluwatobi", "Sofela", "CodeSweetly", "Web Developer", "Male");

// The invocation above will return:
["CodeSweetly", "Web Developer", "Male"]
```

Try it on StackBlitz

In the snippet above, notice that `myBio` 's invocation passed five arguments to the function.

In other words, `"Oluwatobi"` and `"Sofela"` got assigned to the `firstName` and `lastName` parameters.

At the same time, the rest operator added the remaining arguments (`"CodeSweetly"` , `"Web Developer"` , and `"Male"`) into an array and assigned that array to the `otherInfo` parameter.

Therefore, `myBio()` function correctly returned `["CodeSweetly", "Web Developer", "Male"]` as the content of the `otherInfo` rest parameter.

Forum Donate

Learn to code — free 3,000-hour curriculum

Parameter

Keep in mind that you *cannot* use the `"use strict"` directive inside any function containing a rest parameter, default parameter, or destructuring parameter. Otherwise, the computer will throw a syntax error.

For instance, consider this example below:

```
// Define a function with one rest parameter:
function printMyName(...value) {
  "use strict";
  return value;
}

// The definition above will return:
"Uncaught SyntaxError: Illegal 'use strict' directive in function with nc
```

Try it on CodeSandbox

`printMyName()` returned a syntax error because we used the `"use strict"` directive inside a function with a rest parameter.

But suppose you need your function to be in strict mode while also using the rest parameter. In such a case, you can write the `"use strict"` directive outside the function.

Here's an example:

Forum Donate

Learn to code — free 3,000-hour curriculum

```
// Define a function with one rest parameter:
function printMyName(...value) {
  return value;
}

// Invoke the printMyName function while passing two arguments to its par
printMyName("Oluwatobi", "Sofela");

// The invocation above will return:
["Oluwatobi", "Sofela"]
```

Try it on CodeSandbox

Note: Only place the `"use strict"` directive outside your function if it is okay for the entire script or enclosing scope to be in strict mode.

So now that we know how rest works in a function, we can talk about how it works in a destructuring assignment.

How the Rest Operator Works in a Destructuring Assignment

The rest operator typically gets used as a prefix of the destructuring assignment's last variable.

Here's an example:

```
// Define a destructuring array with two regular variables and one rest v
const [firstName, lastName, ...otherInfo] = [
  "Oluwatobi", "Sofela", "CodeSweetly", "Web Developer", "Male"
];
```

Forum Donate

Learn to code — free 3,000-hour curriculum

```
// The invocation above will return:
["CodeSweetly", "Web Developer", "Male"]
```

Try it on StackBlitz

The rest operator (`...`) instructs the computer to add the rest of the user-supplied values into an array. Then, it assigns that array to the `otherInfo` variable.

As such, you may call `...otherInfo` a rest variable.

Here's another example:

```
// Define a destructuring object with two regular variables and one rest
const { firstName, lastName, ...otherInfo } = {
  firstName: "Oluwatobi",
  lastName: "Sofela",
  companyName: "CodeSweetly",
  profession: "Web Developer",
  gender: "Male"
}

// Invoke the otherInfo variable:
console.log(otherInfo);

// The invocation above will return:
{companyName: "CodeSweetly", profession: "Web Developer", gender: "Male"}
```

Try it on StackBlitz

Forum Donate

Learn to code — free 3,000-hour curriculum

In other words, whenever you use rest in a destructuring object, the rest operator will produce a properties object.

However, if you use rest in a destructuring array or function, the operator will yield an array literal.

Before we wrap up our discussion on rest, you should be aware of some differences between JavaScript arguments and the rest parameter. So, let's talk about that below.

Arguments vs Rest Parameters: What's the Difference?

Here are some of the differences between JavaScript arguments and the rest parameter:

Difference 1: The `arguments` object is an array-like object — not a real array!

Keep in mind that the JavaScript arguments object is not a real array. Instead, it is an array-like object that does not have the comprehensive features of a regular JavaScript array.

The rest parameter, however, is a real array object. As such, you can use all array methods on it.

So for instance, you can call the `sort()`, `map()`, `forEach()`, or `pop()` method on a rest parameter. But you cannot do the same on the arguments object.

Forum Donate

Learn to code — free 3,000-hour curriculum

The `arguments` object is not available within an arrow function, so you can't use it there. But you can use the rest parameter within all functions — including the arrow function.

Difference 3: Let rest be your preference

It is best to use rest parameters instead of the `arguments` object — especially while writing ES6 compatible code.

Now that we know how rest works, let's discuss the `spread` operator so we can see the differences.

What Is the Spread Operator and How Does `spread` work in JavaScript?

The **spread operator** (`...`) helps you expand iterables into individual elements.

The spread syntax works within array literals, function calls, and initialized property objects to spread the values of iterable objects into separate items. So effectively, it does the opposite thing from the rest operator.

Note: A spread operator is effective only when used within array literals, function calls, or initialized properties objects.

So, what exactly does this mean? Let's see with some examples.

Forum Donate

Learn to code — free 3,000-hour curriculum

```
const myName = ["Sofela", "is", "my"];
const aboutMe = ["Oluwatobi", ...myName, "name."];

console.log(aboutMe);

// The invocation above will return:
[ "Oluwatobi", "Sofela", "is", "my", "name." ]
```

Try it on StackBlitz

The snippet above used spread (`...`) to copy the `myName` array into `aboutMe`.

Note:

- Alterations to `myName` will not reflect in `aboutMe` because all the values inside `myName` are primitives. Therefore, the spread operator simply copied and pasted `myName`'s content into `aboutMe` without creating any reference back to the original array.
- As mentioned by @nombrekeff in a comment here, the spread operator only does shallow copy. So, keep in mind that supposing `myName` contained any non-primitive value, the computer would have created a reference between `myName` and `aboutMe`. See info 3 for more on how the spread operator works with primitive and non-primitive values.
- Suppose we did not use the spread syntax to duplicate `myName`'s content. For instance, if we had written `const aboutMe = ["Oluwatobi", myName, "name."]`. In such a case,

Forum Donate

Learn to code — free 3,000-hour curriculum

reflect in the duplicated one.

Spread Example 2: How to Use Spread to Convert a String into Individual Array Items

```
const myName = "Oluwatobi Sofela";

console.log([...myName]);

// The invocation above will return:
[ "O", "l", "u", "w", "a", "t", "o", "b", "i", " ", "S", "o", "f", "e", "
```

Try it on StackBlitz

In the snippet above, we used the spread syntax (`...`) within an array literal object (`[...]`) to expand `myName` 's string value into individual items.

As such, `"Oluwatobi Sofela"` got expanded into `[ "O", "l", "u", "w", "a", "t", "o", "b", "i", " ", "S", "o", "f", "e", "l", "a" ]`.

Spread Example 3: How the Spread Operator Works in a Function Call

```
const numbers = [1, 3, 5, 7];

function addNumbers(a, b, c, d) {
  return a + b + c + d;
```

Forum Donate

Learn to code — free 3,000-hour curriculum

```
// The invocation above will return:
16
```

Try it on StackBlitz

In the snippet above, we used the spread syntax to spread the `numbers` array's content across `addNumbers()`'s parameters.

Suppose the `numbers` array had more than four items. In such a case, the computer will only use the first four items as `addNumbers()` argument and ignore the rest.

Here's an example:

```
const numbers = [1, 3, 5, 7, 10, 200, 90, 59];

function addNumbers(a, b, c, d) {
  return a + b + c + d;
}

console.log(addNumbers(...numbers));

// The invocation above will return:
16
```

Try it on StackBlitz

Here's another example:

Forum Donate

Learn to code — free 3,000-hour curriculum

```
  return a + b + c;
}

console.log(spellName(...myName));      // returns: "Olu"

console.log(spellName(...myName[3]));   // returns: "wundefinedundefined"

console.log(spellName([...myName]));    // returns: "O,l,u,w,a,t,o,b,i, ,

console.log(spellName({...myName}));    // returns: "[object Object]undef
```

Try it on StackBlitz

Spread Example 4: How Spread Works in an Object Literal

```
const myNames = ["Oluwatobi", "Sofela"];
const bio = { ...myNames, runs: "codesweetly.com" };

console.log(bio);

// The invocation above will return:
{ 0: "Oluwatobi", 1: "Sofela", runs: "codesweetly.com" }
```

Try it on StackBlitz

In the snippet above, we used spread inside the `bio` object to expand `myNames` values into individual properties.

What to Know About the Spread Operator

Forum Donate

Learn to code — free 3,000-hour curriculum

Info 1: Spread operators can't expand object literal's values

Since a properties object is not an iterable object, you cannot use the spread operator to expand its values.

However, you can use the spread operator to clone properties from one object into another.

Here's an example:

```
const myName = { firstName: "Oluwatobi", lastName: "Sofela" };
const bio = { ...myName, website: "codesweetly.com" };

console.log(bio);

// The invocation above will return:
{ firstName: "Oluwatobi", lastName: "Sofela", website: "codesweetly.com"
```

Try it on StackBlitz

The snippet above used the spread operator to clone `myName`'s content into the `bio` object.

Note:

- The spread operator can expand iterable objects' values only.

Forum Donate

Learn to code — free 3,000-hour curriculum

- Array, TypedArray, String, Map, and Set are all built-in iterable types because they have the `@@iterator` property by default.
- A properties object is not an iterable data type because it does not have the `@@iterator` property by default.
- You can make a properties object iterable by adding `@@iterator` onto it.

Info 2: The spread operator does not clone identical properties

Suppose you used the spread operator to clone properties from object A into object B. And suppose object B contains properties identical to those in object A. In such a case, B's versions will override those inside A.

Here's an example:

```
const myName = { firstName: "Tobi", lastName: "Sofela" };
const bio = { ...myName, firstName: "Oluwatobi", website: "codesweetly.cc

console.log(bio);

// The invocation above will return:
{ firstName: "Oluwatobi", lastName: "Sofela", website: "codesweetly.com"
```

Try it on StackBlitz

Forum Donate

Learn to code — free 3,000-hour curriculum

Info 3: Beware of how spread works when used on objects containing non-primitives!

Suppose you used the spread operator on an object (or array) containing only primitive values. The computer will not create any reference between the original object and the duplicated one.

For instance, consider this code below:

```
const myName = ["Sofela", "is", "my"];
const aboutMe = ["Oluwatobi", ...myName, "name."];

console.log(aboutMe);

// The invocation above will return:
["Oluwatobi", "Sofela", "is", "my", "name."]
```

Try it on StackBlitz

Observe that every item in `myName` is a primitive value. Therefore, when we used the spread operator to clone `myName` into `aboutMe`, the computer did not create any reference between the two arrays.

As such, any alteration you make to `myName` will not reflect in `aboutMe`, and vice versa.

As an example, let's add more content to `myName`:

Forum Donate

Learn to code — free 3,000-hour curriculum

Now, let's check the current state of `myName` and `aboutMe` :

```
console.log(myName); // ["Sofela", "is", "my", "real"]

console.log(aboutMe); // ["Oluwatobi", "Sofela", "is", "my", "name."]
```

Try it on StackBlitz

Notice that `myName` 's updated content did not reflect in `aboutMe` — because spread created no reference between the original array and the duplicated one.

What if `myName` contains non-primitive items?

Suppose `myName` contained non-primitives. In that case, spread will create a reference between the original non-primitive and the cloned one.

Here is an example:

```
const myName = [["Sofela", "is", "my"]];
const aboutMe = ["Oluwatobi", ...myName, "name."];

console.log(aboutMe);

// The invocation above will return:
[ "Oluwatobi", ["Sofela", "is", "my"], "name." ]
```

Forum Donate

Learn to code — free 3,000-hour curriculum

Therefore, using the spread operator to clone `myName`'s content into `aboutMe` caused the computer to create a reference between the two arrays.

As such, any alteration you make to `myName`'s copy will reflect in `aboutMe`'s version, and vice versa.

As an example, let's add more content to `myName`:

```
myName[0].push("real");
```

Now, let's check the current state of `myName` and `aboutMe`:

```
console.log(myName); // [["Sofela", "is", "my", "real"]]

console.log(aboutMe); // ["Oluwatobi", ["Sofela", "is", "my", "real"], "r
```

Try it on StackBlitz

Notice that `myName`'s updated content is reflected in `aboutMe` — because spread created a reference between the original array and the duplicated one.

Here's another example:

Forum Donate

Learn to code — free 3,000-hour curriculum

```
const bio = { ...myName };

myName.firstName = "Tobi";

console.log(myName); // { firstName: "Tobi", lastName: "Sofela" }

console.log(bio); // { firstName: "Oluwatobi", lastName: "Sofela" }
```

Try it on StackBlitz

In the snippet above, `myName` 's update did not reflect in `bio` because we used the spread operator on an object that contains primitive values only.

Note: A developer would call `myName` a **shallow object** because it contains only primitive items.

Here's one more example:

```
const myName = {
  fullName: { firstName: "Oluwatobi", lastName: "Sofela" }
};

const bio = { ...myName };

myName.fullName.firstName = "Tobi";

console.log(myName); // { fullName: { firstName: "Tobi", lastName: "Sofel

console.log(bio); // { fullName: { firstName: "Tobi", lastName: "Sofela"
```

Forum Donate

Learn to code — free 3,000-hour curriculum

used the spread operator on an object that contains a non-primitive value.

Note:

- We call `myName` a **deep object** because it contains a non-primitive item.
- You do **shallow copy** when you create references while cloning one object into another. For instance, `...myName` produces a shallow copy of the `myName` object because whatever alteration you make in one will reflect in the other.
- You do **deep copy** when you clone objects without creating references. For instance, I could deep copy `myName` into `bio` by doing `const bio = JSON.parse(JSON.stringify(myName))`. By doing so, the computer will clone `myName` into `bio` without creating any reference.
- You can break off the reference between the two objects by replacing the `fullName` object inside `myName` or `bio` with a new object. For instance, doing `myName.fullName = { firstName: "Tobi", lastName: "Sofela" }` would disconnect the pointer between `myName` and `bio`.

Wrapping it up

This article discussed the differences between the rest and spread operators. We also used examples to see how each operator works.

Forum Donate

Learn to code — free 3,000-hour curriculum

Oluwatobi Sofela

Read more posts.

If you read this far, thank the author to show them you care.

Say Thanks

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

ADVERTISEMENT

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) charity organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public.

Forum Donate

Learn to code — free 3,000-hour curriculum

You can make a tax-deductible donation here.

Trending Guides

Learn CSS Transform
What is Programming?
HTTP Networking in JS
How to Write Clean Code
Learn Swift
Learn CSS Grid
Learn JS Modules
Front-End JS Development
Command Line for Beginners
OSS Security Best Practices
Build a Static Blog
Python Code Examples
Write React Unit Tests
Learn PHP
Learn Golang
Learn Solidity
Learn Apache Kafka
Learn to Build REST APIs
Intro to Operating Systems
Distributed Systems Patterns
Build an AI Chatbot
Open Source for Devs
Learn Algorithms in JS
Learn Java
Learn Node.js
Learn Express.js
REST API Best Practices
Intermediate TS and React
Learn to Build GraphQL APIs
Software Architecture Patterns

Mobile App

Download on the App Store
GET IT ON Google Play

Our Charity

About Alumni Network Open Source Shop Support Sponsors Academic Honesty
Code of Conduct Privacy Policy Terms of Service Copyright Policy