

Advanced JS features

Below are the few features we need to cover before stating React.js

- Arrow Functions
- built in functions like forEach, map, reduce, find, filter
- Spread operator / Rest operator
- Array de-structuring
- Object de-structuring
- JavaScript Modules

1. Arrow Functions

Arrow functions, also known as fat arrow functions, provide a concise way to write JavaScript functions.

They were introduced in ES6 (ECMAScript 2015) and have become popular due to their readability and simplicity.

Arrow functions in JavaScript provide a more concise syntax for writing functions compared to traditional function expressions. They are especially useful when working with functions in situations like callbacks or anonymous functions.

Basic syntax:

```
javascript

const functionName = (parameters) => {
  // function body
};
```

Key Features of Arrow Functions:

Shorter Syntax: Arrow functions remove the need to use the function keyword.

Implicit Return: If the function body is a single expression, you can omit the curly braces { } and the return keyword. The expression will automatically be returned.

Example:

javascript

```
const square = (x) => x * x;
```

2. Built In functions:

- **forEach method:**

The `forEach()` method in JavaScript is a built-in array method that allows you to iterate over all elements in an array and execute a provided function for each element. It's particularly useful when you need to perform an action on each element of an array without modifying the original array.

Basic Example:

javascript

```
const numbers = [1, 2, 3, 4, 5];

numbers.forEach((number) => {
  console.log(number); // Prints each number in the array
});
```

Example with Index and Array:

javascript

```
const fruits = ['Apple', 'Banana', 'Cherry'];


fruits.forEach((fruit, index, array) => {
  console.log(`${index}: ${fruit}`); // Prints each fruit with its index
  // Example of accessing the original array
  if (index === 1) {
    console.log(`The second fruit is: ${array[1]}`);
  }
});
```



Example for iterating object array:

javascript

```
const users = [  
  { name: 'John', age: 25 },  
  { name: 'Jane', age: 30 },  
  { name: 'Mike', age: 35 }  
];  
  
users.forEach((user) => {  
  console.log(`${user.name} is ${user.age} years old.`);  
});
```



forEach() is a simple and effective way to loop over arrays when you don't need to break the loop or accumulate values.

- **Find method:**

The find() method in JavaScript is used to search through an array and return the **first element** that satisfies the provided condition in the callback function. If no element satisfies the condition, it returns undefined.

Examples:

Finding a Number

If you're looking for the first number greater than 10 in an array:

javascript

```
const numbers = [5, 12, 8, 130, 44];  
  
const result = numbers.find((num) => num > 10);  
console.log(result); // 12
```

Finding an Object in an Array

If you're searching for an object in an array of objects based on a property:

javascript

```
const users = [
  { id: 1, name: 'John', age: 25 },
  { id: 2, name: 'Jane', age: 30 },
  { id: 3, name: 'Mike', age: 35 }
];

const user = users.find((user) => user.id === 2);
console.log(user); // { id: 2, name: 'Jane', age: 30 }
```

When to Use find():

Finding a unique value: Use find() when you need to find a specific value or object in an array based on a condition.

First match: Use it when you only care about the first matching element and don't need the rest.

- **Filter method**

The filter() method in JavaScript is used to create a **new array** that contains all the elements from the original array that pass a specific test provided by a callback function. The original array is not modified.

Key Points:

- **Does not modify the original array:** The filter() method creates a new array, and the original array remains unchanged.
- **Returns a new array:** It can return zero, one, or more elements that satisfy the condition.

Examples:

Filtering Even Numbers

javascript

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

const evenNumbers = numbers.filter((num) => num % 2 === 0);
console.log(evenNumbers); // [2, 4, 6, 8, 10]
```

Filtering Objects by Property

Suppose you have an array of objects, and you want to filter by a specific property:

javascript

```
const users = [
  { id: 1, name: 'John', age: 25 },
  { id: 2, name: 'Jane', age: 30 },
  { id: 3, name: 'Mike', age: 35 },
  { id: 4, name: 'Anna', age: 25 }
];

const youngUsers = users.filter((user) => user.age === 25);
console.log(youngUsers);
// [ { id: 1, name: 'John', age: 25 }, { id: 4, name: 'Anna', age: 25 } ]
```

Filtering Strings

You can filter strings based on a specific condition, such as strings with more than 5 characters:

javascript

```
const words = ['apple', 'banana', 'cherry', 'kiwi', 'mango'];

const longWords = words.filter((word) => word.length > 5);
console.log(longWords); // ['banana', 'cherry']
```

When to Use filter():

- **Filtering elements:** Use filter() when you need to create a new array based on a condition applied to the elements of the original array.
- **Working with objects or arrays:** It's especially useful for filtering arrays of objects by specific properties.
- **When you need to retain all matches:** Unlike find(), which only returns the first match, filter() returns all matching elements.

Differences Between filter() and find():

filter(): Returns **all matching elements** that satisfy the condition (it always returns an array, even if only one or no elements match).

find(): Returns **only the first matching element** or undefined if no elements match.

```
javascript
```

```
const numbers = [5, 10, 15, 20, 25, 30];

// filter() will return all numbers greater than 15
const filterResult = numbers.filter((num) => num > 15);
console.log(filterResult); // [20, 25, 30]

// find() will return the first number greater than 15
const findResult = numbers.find((num) => num > 15);
console.log(findResult); // 20
```

- **Map method**

The map() is used to transform each element in an array and return a new array of the transformed elements.

Key Points:

- Does not modify the original array.
- Returns a new array with the same length as the original.
- Ideal for data transformation tasks (e.g., modifying each element or performing calculations).

Examples:

Doubling the Numbers

Let's say you want to create a new array where each element is twice the value of the original element.

```
javascript
```

```
const numbers = [1, 2, 3, 4, 5];

const doubledNumbers = numbers.map((num) => num * 2);
console.log(doubledNumbers); // [2, 4, 6, 8, 10]
```

Extracting Specific Properties from Objects

You can use map() to extract specific properties from an array of objects:

javascript

```
const users = [  
  { id: 1, name: 'John', age: 25 },  
  { id: 2, name: 'Jane', age: 30 },  
  { id: 3, name: 'Mike', age: 35 }  
];  
  
const userNames = users.map((user) => user.name);  
console.log(userNames); // ['John', 'Jane', 'Mike']
```

Mapping to New Objects

You can map over objects to create new objects with modified properties:

javascript

```
const products = [  
  { name: 'Laptop', price: 1000 },  
  { name: 'Phone', price: 500 },  
  { name: 'Tablet', price: 300 }  
];  
  
const discountedProducts = products.map((product) => ({  
  name: product.name,  
  price: product.price * 0.9 // 10% discount  
}));  
  
console.log(discountedProducts);
```

```
// [  
//   { name: 'Laptop', price: 900 },  
//   { name: 'Phone', price: 450 },  
//   { name: 'Tablet', price: 270 }  
// ]
```

When to Use map():

- **Transformation:** When you need to apply a transformation (like mathematical operations, string manipulations, or property extraction) to every element of an array.

- **Creating a new array:** Use `map()` when you want to create a new array, while the original array remains unchanged.
- **Working with arrays of objects:** It's very common to use `map()` to extract certain properties or modify objects in arrays.
- **Reduce method**

The `reduce()` method in JavaScript is a powerful array method that allows you to **accumulate** or **reduce** all the elements of an array to a single value. This method takes a callback function that is executed for each element in the array, and the result is combined into a single output.

Key Points:

- **Accumulates values:** The `reduce()` method processes each element of the array and accumulates the result based on the logic you provide in the callback function.
- **Can be used for various operations:** It's very versatile, and you can use `reduce()` for tasks like summing values, counting occurrences, flattening arrays, or even performing more complex transformations.
- **Can provide an initial value:** You can specify an initial value for the accumulator (such as 0 for sum or an empty object for accumulating key-value pairs).

Examples:

Summing All Values in an Array

One of the most common uses of `reduce()` is to sum the values of an array:

javascript

```
const numbers = [1, 2, 3, 4, 5];

const sum = numbers.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
console.log(sum); // 15
```

Copy

Finding the Maximum Value

You can use `reduce()` to find the maximum value in an array:

javascript

```
const numbers = [1, 5, 3, 9, 2];

const max = numbers.reduce((accumulator, currentValue) => {
  return currentValue > accumulator ? currentValue : accumulator;
}, numbers[0]); // Start with the first element
console.log(max); // 9
```


Flattening an Array of Arrays

You can use `reduce()` to flatten a nested array (an array of arrays) into a single array:

javascript

```
const arrays = [[1, 2, 3], [4, 5], [6, 7, 8]];

const flattened = arrays.reduce((accumulator, currentValue) => {
  return accumulator.concat(currentValue);
}, []);

console.log(flattened); // [1, 2, 3, 4, 5, 6, 7, 8]
```

3. Rest operator / Spread operator

The **spread** and **rest** operators in JavaScript are both represented by the same syntax (`...`), but they are used in different contexts to perform different tasks.

- **Spread Operator (...)**

The **spread** operator is used to **expand** or **spread** elements from an array or object into individual elements. It's typically used when you want to spread the values of an array or object into another array or object.

Syntax:

javascript

```
// Array Spread
let newArray = [...array];

// Object Spread
let newObject = {...object};
```

Key Use Cases for Spread:

- Copying Arrays or Objects
- Merging Arrays or Objects
- Passing elements as arguments

Examples of the Spread Operator:

Copying Arrays

Using the spread operator to create a shallow copy of an array:

javascript

```
const arr = [1, 2, 3];
const copiedArr = [...arr];

console.log(copiedArr); // [1, 2, 3]
```

Merging Arrays

You can merge multiple arrays into one using the spread operator:

javascript

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const mergedArr = [...arr1, ...arr2];

console.log(mergedArr); // [1, 2, 3, 4, 5, 6]
```

Copying Objects

The spread operator can be used to make a shallow copy of an object:

javascript

```
const person = { name: 'John', age: 30 };
const copiedPerson = { ...person };

console.log(copiedPerson); // { name: 'John'
```

Combining Objects

You can also use the spread operator to combine objects:

javascript

```
const obj1 = { name: 'John' };
const obj2 = { age: 30 };
const combinedObj = { ...obj1, ...obj2 };

console.log(combinedObj); // { name: 'John', age: 30 }
```

Using Spread for Cloning and Modifying Objects

javascript

```
const person = { name: 'John', age: 30 };

const updatedPerson = { ...person, age: 31 };
console.log(updatedPerson); // { name: 'John', age: 31 }
```

- **Rest Operator (...)**

The **rest operator** is used to **collect** multiple elements into a single array. It is commonly used in function parameters, where it allows you to capture remaining arguments into an array.

Use Cases:

- **Function parameters:** Collect all remaining arguments into an array.
- **De-structuring arrays/objects:** Collect remaining elements from arrays or objects into a new variable.

Examples:

Function Parameters

You can use the rest operator in a function's parameter to collect multiple arguments into a single array.

javascript

```
const sum = (...numbers) => {
  return numbers.reduce((acc, num) => acc + num, 0);
};

console.log(sum(1, 2, 3, 4)); // 10
```

Destructuring Arrays

The rest operator can be used with destructuring to collect the remaining elements of an array.

javascript

```
const arr = [1, 2, 3, 4, 5];

const [first, second, ...rest] = arr;

console.log(first); // 1
console.log(second); // 2
console.log(rest); // [3, 4, 5]
```

De-structuring Objects

The rest operator can also be used with objects to collect the remaining properties into a new object.

javascript

```
const person = { name: 'John', age: 30, gender: 'male' };

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

console.log(name); // John
console.log(rest); // { age: 30, gender: 'male' }
```

Key Differences Between Spread and Rest:

Aspect	Spread Operator	Rest Operator
Purpose	Unpack elements or properties	Collect multiple elements or values into an array
Use Case	Expanding arrays or objects	Collecting arguments or remaining elements during destructuring
Position	Used in function calls, array/object literals	Used in function parameters or destructuring assignments

4. Array Destructuring

Array destructuring in JavaScript allows you to unpack values from arrays (or any iterable objects) into distinct variables in a clean, concise way. It's a shorthand method that improves readability and reduces boilerplate code.

Basic Syntax:

```
javascript

const array = [1, 2, 3];
const [a, b, c] = array;

console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

Skipping Items:

You can skip elements in the array if you don't need all of them.

```
javascript

const array = [1, 2, 3, 4];
const [, b, , d] = array;

console.log(b); // 2
console.log(d); // 4
```

Default Values:

You can assign default values in case an array element is undefined.

```
javascript

const array = [1, 2];
const [a, b, c = 3] = array;

console.log(a); // 1
console.log(b); // 2
console.log(c); // 3 (default value since there is no third element)
```

Rest Syntax:

You can use the rest (...) syntax to collect the remaining items of the array into another array.

javascript

```
const array = [1, 2, 3, 4];
const [a, ...rest] = array;

console.log(a); // 1
console.log(rest); // [2, 3, 4]
```

Destructuring with Function Returns:

You can also destructure values returned from functions.

javascript

```
function getValues() {
  return [10, 20, 30];
}

const [x, y, z] = getValues();
console.log(x, y, z); // 10 20 30
```

5. Object Destructuring

Object destructuring in JavaScript allows you to extract values from objects and assign them to variables in a more concise and readable way. Instead of accessing properties one by one using the dot notation, you can "destructure" the object directly.

Note: Object keys and variable names must be same.

Basic Syntax:

javascript

```
const person = { name: 'John', age: 30, country: 'USA' };

// Destructuring assignment
const { name, age, country } = person;

console.log(name);    // 'John'
console.log(age);     // 30
console.log(country); // 'USA'
```

Renaming Variables:

You can rename the variables while destructuring using a colon (:).

```
javascript

const person = { name: 'John', age: 30 };

// Destructuring with renaming
const { name: fullName, age: yearsOld } = person;

console.log(fullName); // 'John'
console.log(yearsOld); // 30
```

Default Values:

You can provide default values for properties that may not exist in the object.

```
javascript

const person = { name: 'John' };

// Destructuring with default value
const { name, age = 25 } = person;

console.log(name); // 'John'
console.log(age); // 25 (since 'age' is not in the object, default value is used)
```

Nested Destructuring:

Destructuring works with nested objects too.

```
const person = {
  name: 'John',
  address: { city: 'New York', zip: '10001' }
};

// Nested destructuring
const { name, address: { city, zip } } = person;

console.log(name); // 'John'
console.log(city); // 'New York'
console.log(zip); // '10001'
```

6. Modules

JavaScript modules allow you to split your code into smaller, reusable pieces. This makes your code more maintainable and organized. With modules, you can export functions, objects, or variables from one file and import them into another. This is especially useful in large applications where you want to divide your code into logical sections.

How to use JavaScript Modules:

1. Exporting in a Module:

You can export values from a module using the export keyword.

Named Export:

You can export multiple values from a module using named exports.

```
javascript

// math.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
```

Default Export:

You can export a single value or function from a module using export default.

```
javascript

// greet.js
const greet = (name) => `Hello, ${name}!`;
export default greet;
```

2. Importing in Another File:

To use the exported values in another file, you can import them using the import keyword.

Import Named Exports:

You can import named exports using { } and the exact name of the export.


```
javascript

// app.js
import { add, subtract } from './math.js';

console.log(add(2, 3));    // 5
console.log(subtract(5, 3)); // 2
```

Import Default Export:

You import the default export without `{ }` and you can give it any name.

```
javascript

// app.js
import greet from './greet.js';

console.log(greet('Alice')); // 'Hello, Alice!'
```

Renaming Imports:

You can rename imports using the `as` keyword.

```
// app.js
import { add as addition, subtract as difference } from './math.js';

console.log(addition(2, 3));    // 5
console.log(difference(5, 3));  // 2
```

Importing All Exports from a Module:

You can import all exports from a module as an object.

```
javascript

// app.js
import * as math from './math.js';

console.log(math.add(2, 3));    // 5
console.log(math.subtract(5, 3)); // 2
```

File Extension for Modules:

When using modules in JavaScript, the file extension is important. Typically, you will use .js for JavaScript modules. If you're using ES modules in the browser, make sure your `<script>` tag includes the `type="module"` attribute.

```
html
```

```
<script type="module" src="app.js"></script>
```

3. Advantages of JavaScript Modules:

- **Code Organization:** You can break your code into logical chunks, making it easier to maintain.
- **Reusability:** Once a module is written, it can be reused across different parts of the application.
- **Namespace Management:** Modules help avoid polluting the global namespace by encapsulating variables and functions.
- **Performance:** Modern JavaScript bundlers (like Webpack, Rollup) can optimize the loading of modules, improving application performance.

Note: To create this handout, references are taken from different sources. Use it within organization. Do not share.