JavaScript ES6 Array Methods: forEach, map, reduce, filter

ES6 has introduced several array methods like `forEach()`, `map()`, `filter()`, `reduce()`, and more. These methods are directly available on array instances and provide elegant ways to manipulate arrays.

```
const array = [1, 2, 3, 4, 5];
array.filter(/* callback function */);
```

A crucial component of these methods is the callback function - an anonymous function passed as an argument to another function, to be executed later. This callback function dictates the behavior of the array method.

For example, consider `filter()`, which creates a new array with all elements that pass a test implemented by the provided callback function.

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

function isEven(number) {
  return number % 2 === 0;
}

const evenNumbers = array.filter(isEven);

const evenNumbers = array.filter(function(number){
  return number % 2 === 0
});

const evenNumbers = array.filter(number => number % 2 === 0);
```

In the above example, the callback function `number => number % 2 === 0` tests whether each number in the array is even. Only the even numbers pass this test and are included in the new `evenNumbers` array.

These callback-based array methods offer flexibility and power, allowing for simple to complex operations on arrays.

1. forEach

The `forEach()` method is a built-in JavaScript array method that executes a provided function once for each array element. It is a modern replacement of the `for` loop, the regular one, and the `for of` loop. The `forEach()` method is a way to run a function on each item in the array, in order. This method is extremely useful when you want to perform an operation using each element of an array. (It is crucial to note that the `forEach()` method **does not return anything**, unlike other ES6 array methods)

Consider this traditional `for` loop:

```
const emails = ["test1@example.com", "test2@example.com", "test3@example.com"];
for(const i = 0; i < emails.length; i++) {
    console.log(emails[i]);
}
```

This loop goes through each email in the `emails` array and prints an email to each. It's not too complex, but the syntax is a little clumsy. There's quite a bit of manual work involved: we need to initialize a loop counter (`i`), set up a condition that the loop continues as long as it's true, and increment `i` on each iteration.

Now let's accomplish the same task using `forEach` :

```
emails.forEach((email) => {
  console.log(email);
});
```

This code does the same thing as the `for` loop, but it's cleaner and more readable. We didn't have to manually manage a loop counter, making it more declarative in nature.

Utilizing Index in forEach Method

The `forEach()` method doesn't just give you access to the current item during each iteration, it also provides the index of that item and the original array itself. This can be quite useful in certain situations.

The syntax for `forEach` looks like this:

```
arr.forEach(callback(currentValue [, index [, array]])[, thisArg])
```

Here, `callback` is a function that you pass into `forEach()` , and this function can take in up to three arguments:

1. `currentValue` (required): The value of the current element in the array.
2. `index` (optional): The index of the current element in the array.
3. `array` (optional): The original array that `forEach` was called upon.

Now, let's see a practical example of using the `index` argument:

Imagine you have an array of product prices and you want to apply a special 10% discount to every third item. This is a case where knowing the index is essential:

```
const prices = [100, 200, 300, 400, 500, 600, 700];

prices.forEach((price, index) => {
  if ((index + 1) % 3 === 0) {
    // Apply 10% discount
    prices[index] = price * 0.9;
  }
});
```

In the code above, `forEach()` iterates through the `prices` array. For every third item (where `(index + 1) % 3 === 0`), it modifies the price to 90% of the original price, thus applying a 10% discount.

Remember, array indices in JavaScript start at 0, which is why we use `index + 1` to find every third item.

So, `forEach()` not only simplifies iterating over arrays, but also provides valuable information like the current index and even the original array, which you can use to achieve a variety of tasks.

Differences Between forEach and for

There are instances where a traditional `for` loop might be more suitable than `forEach`:

1. **Break**: In a `for` loop, you can break out of the loop using a `break` statement. This can be useful when you want to stop the loop as soon as a condition is met. However, `forEach` doesn't have this capability, it always iterates over all elements.

Here's a case where you might want to use a `for` loop instead of `forEach`. Let's say you have a list of users and you want to check if a user with a particular ID exists:

```
const users = [{ id: 1, name: 'John' }, { id: 2, name: 'Jane' }, { id: 3, name: 'Bob' }];

const targetId = 2;
const targetUser = null;
for (const i = 0; i < users.length; i++) {
  if (users[i].id === targetId) {
    targetUser = users[i];
    break; // Exit the loop as soon as the user is found
  }
}

if (targetUser) {
  // Perform some action with targetUser
}
```

2. **Async/Await**: `for` loops play nicely with async/await. If you have an asynchronous operation inside a loop that needs to be performed in order, `for` loops should be your go-to choice. The `forEach` method does not wait for promises:

```
const userIds = [1, 2, 3, 4, 5];
for (const i = 0; i < userIds.length; i++) {
  const user = await fetchUserById(userIds[i]); // hypothetical async function to fetch
user
  // Perform some action with user
}
```

Despite these exceptions, `forEach` is generally a great choice for simpler, more readable code when iterating over arrays in JavaScript.

Real-world applications of forEach:

The `forEach()` method can be used in various real-world scenarios such as:

1. **Mutating the original array**: As shown above, if you need to modify the original array based on its elements, `forEach()` is an excellent choice.

2. **Performing side effects**: If you need to execute a function that has a side effect (like making an API call for each element), `forEach()` is an appropriate choice (this is an example for when we would exercise discretion and write an impure function), For example:

```
const postIds = [1, 2, 3, 4, 5];
postIds.forEach(id => {
    fetch(`/api/posts/${id}`).then(response => {
        // handle response
    });
});
```

3. **Interacting with the DOM**: Often, you need to manipulate multiple DOM elements in the same way. `forEach()` is perfect for these scenarios:

```
const buttons = document.querySelectorAll('button');
buttons.forEach(button => {
    button.addEventListener('click', handleButtonClick);
});
```

Remember, though, `forEach()` does not return a new array, unlike most other array methods. If you need to transform an array into a new array, you're better off with `map()`, `filter()`, or `reduce()`.

2. map

The `map()` method creates a new array with the results of calling a provided function on every element in the calling array. This is particularly useful when you want to perform data transformations.

Imagine you're running an e-commerce store where products are represented as an array of objects. Each product has a `price`, and you want to display the prices to the customer in a friendly format:

```
const products = [
  { id: 1, name: 'Jeans', price: 49.99 },
  { id: 2, name: 'T-Shirt', price: 19.99 },
  // more products...
];

const formattedPrices = products.map(product => {
  return { name: product.name, price: `$${product.price.toFixed(2)}` };
});

// formattedPrices is now:
// [
//   { name: 'Jeans', price: '$49.99' },
//   { name: 'T-Shirt', price: '$19.99' },
//   // more prices...
// ]
```

The original `products` array remains unchanged. The `formattedPrices` array holds the new, transformed data.

Understanding the map Method and Its Difference from forEach

The `map()` method, much like `forEach()`, is a method built into JavaScript that performs a function on each element in an array. The crucial difference, however, lies in the fact that `map()` creates a new array based on what is returned from the function, whereas `forEach()` doesn't return anything.

The syntax for `map` is as follows:

```
const newArray = arr.map(callback(currentValue[, index[, array]])[, thisArg])
```

Similar to `forEach`, `map` takes a callback function as an argument, which can take up to three parameters: `currentValue`, `index`, and `array`.

Here's a simple example:

```
const numbers = [1, 2, 3, 4, 5];
const doubledNumbers = numbers.map(num => num * 2);
console.log(numbers) // [1, 2, 3, 4, 5]
console.log(dumpedNumbers) // [2, 4, 6, 8, 10]
```

In this example, `map()` takes each number in the `numbers` array, multiplies it by 2, and then adds the result to the new `doubledNumbers` array. Notice that the original `numbers` array remains unchanged.

This ability to create a new array makes `map()` a more pure function in comparison to `forEach()`. In functional programming, pure functions are those that do not cause side effects (like mutating global state or the original array). Instead, they take inputs and return outputs without altering anything else.

Because of these characteristics, `map()` is more commonly used when the goal is to transform data. For example, it's perfect when you need to transform an array of objects, adding a new property to each object:

```
const products = [
  { id: 1, price: 100 },
  { id: 2, price: 200 },
  { id: 3, price: 300 },
];

const discountedProducts = products.map(product => ({
  ...product,
  discountedPrice: product.price * 0.9
}));

console.log(products)
/*
  [
    { id: 1, price: 100 },
    { id: 2, price: 200 },
    { id: 3, price: 300 },
  ]
*/
console.log(discountedProducts)
/*
  [
    { id: 1, price: 100, discountedPrice: 90 },
    { id: 2, price: 200, discountedPrice: 180 },
    { id: 3, price: 300, discountedPrice: 270 }
  ]
*/
```

In the code above, we create a new array `discountedProducts` where each product has an additional property `discountedPrice`, which is 90% of the original price. This leaves the original `products` array unchanged, which is often what you want to achieve in many real-world programming scenarios.

Remember, the key point to understand about `map()` is that it does not mutate the original array but instead creates a new one based on the result of the provided function. This makes `map()` especially useful when you want to avoid side effects and keep your code pure and predictable.

3. reduce

Unlocking the Power of the reduce Method in JavaScript

The `reduce()` method in JavaScript is a powerful array method that allows you to process all the elements in an array and return a single output. It is important to note that the `reduce()` method always returns a single value. It's a bit more complex than the other array methods, but it's also incredibly flexible once you understand how it works.

At its core, `reduce()` executes a reducer function on each element in the array, resulting in a single output value. It can be used for simple tasks, like summing numbers in an array, or more sophisticated operations, like transforming an array into an object.

The syntax for `reduce()` looks like this:

```
arr.reduce(callback(accumulator, currentValue[, index[, array]])[, initialValue])
```

Here, the `callback` function is the reducer that you provide. It gets called on each element in the array and has four parameters:

1. `accumulator`: Accumulates the callback's return values. It's the accumulated value previously returned in the last invocation of the callbackâ€”or `initialValue`, if it was supplied.
2. `currentValue`: The current element being processed in the array.
3. `index` (optional): The index of the current element being processed in the array.
4. `array` (optional): The array `reduce()` was called upon.

`initialValue` (optional): A value to use as the first argument to the first call of the callback.

For example, to sum numbers in an array:

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((acc, curr) => acc + curr, 0);
// sum will be: 15
```

In this example, `reduce()` starts with an `accumulator` of `0` (our `initialValue`) and then adds each `currentValue` to the `accumulator` for each iteration through the array, resulting in the sum of all numbers.

But `reduce()` isn't limited to simple tasks. We can use `reduce()` to transform an array into an object. For example:

```
const pets = ['dog', 'cat', 'fish', 'dog', 'cat', 'dog'];
const petCounts = pets.reduce((obj, pet) => {
  if (!obj[pet]) {
    obj[pet] = 1;
  } else {
    obj[pet]++;
  }
  return obj;
}, {});
// petCounts will be: { dog: 3, cat: 2, fish: 1 }
```

In this example, `reduce()` starts with an empty object `{}`. For each iteration, it checks if the `pet` already exists as a key in the object. If it doesn't, it adds the `pet` as a key with a value of `1`. If it does, it increments the value. The result is an object with each `pet` from the array as a key and the number of times each `pet` appeared in the array as its value.

> ### Side note
>
> Note that can simplify that `'if'` statement of this reducer function using a ternary operator. It would look like this:
>
> ```
> const pets = ['dog', 'cat', 'fish', 'dog', 'cat', 'dog'];
> const petCounts = pets.reduce((obj, pet) => {
>   obj[pet] = obj[pet] ? obj[pet] + 1 : 1;
>   return obj;
> }, {});
> ```
>
> In this one-liner version, `obj[pet] ? obj[pet] + 1 : 1;` is a ternary operation. It checks if `obj[pet]` exists. If it does, it increments `obj[pet]` by 1. If it doesn't, it sets `obj[pet]` to 1. This line does the same job as the previous if-else statement but in a more concise way. #

As you can see, `reduce()` provides a way to handle both simple and complex tasks involving arrays, making it one of the most powerful tools in your JavaScript toolbox.

4. filter

The `filter()` method in JavaScript creates a new array with all elements that pass a certain **condition** implemented by the provided function.

The `filter()` method takes in a callback function that has access to each element, its index, and the original array. It returns a new array consisting of only those elements for which the callback function returns a **truthy value**.

Here is the syntax:

```
const newArray = arr.filter(callback(element, index, array), thisArg)
```

- The callback is a function to test each element of the array. The function must return a boolean value, `true` to keep the element, or `false` otherwise.
- The `thisArg` argument is optional and used as `this` when executing callback.

Let's look at some practical examples:

Example 1: Filtering out negative numbers

Let's say we have an array of numbers, and we want to create a new array with only the positive numbers. Here's how we could do it:

```
const numbers = [-3, 2, -8, 12, 5];
const positiveNumbers = numbers.filter(number => number > 0);
// positiveNumbers = [2, 12, 5]
```

Example 2: Filtering based on complex conditions

Imagine we have an array of objects where each object represents a product in an e-commerce store. We want to create a new array with only the products that are in stock and cost less than $10.

```
const products = [
  { name: 'Apple', price: 5, inStock: true },
  { name: 'Banana', price: 15, inStock: true },
  { name: 'Cherry', price: 10, inStock: false },
  { name: 'Date', price: 20, inStock: true },
  { name: 'Elderberry', price: 8, inStock: true },
];

const affordableInStockProducts = products.filter(product => product.inStock &&
product.price < 10);

/*
affordableInStockProducts = [
  { name: 'Apple', price: 5, inStock: true },
  { name: 'Elderberry', price: 8, inStock: true },
]
*/
```

In this example, our condition in the `filter()` method is `product.inStock && product.price < 10`, which will only return `true` for items that are both in stock and cost less than $10.

Remember, the `filter()` method does not mutate the original array but creates a new one. This helps us to maintain data integrity by not altering our original data.

Chaining Array Methods in JavaScript

Chaining array methods like `filter`, `map`, and `reduce` allows us to perform complex data processing in a readable and concise manner. The fact that `filter` and `map` always return an array makes them chainable, as we can simply call another array method on the result.

Let's examine a more detailed example:

```
const movements = [200, 450, -400, 3000, -650, -130, 70, 1300];

const calcDisplaySummary = (movements, interestRate) => {
  const incomes = movements
    .filter(mov => mov > 0)
    .reduce((acc, mov) => acc + mov, 0);
  // The sum of all positive movements, i.e., incomes

  const out = movements
    .filter(mov => mov < 0)
    .reduce((acc, mov) => acc + mov, 0);
  // The sum of all negative movements, i.e., withdrawals

  const interest = movements
    .filter(mov => mov > 0) // filter out negative movements
    .map(deposit => (deposit * interestRate) / 100) // Mapping each deposit to its
corresponding interest
    .filter((int) => int >= 1) // Filter out interests that are less than 1
    .reduce((acc, int) => acc + int, 0); // The total interest earned from positive
movements (incomes)

  return {incomes, out, interest};
};

const summary = calcDisplaySummary(movements, 1.2);
// This will return an object like:
// { incomes: 4020, out: -1180, interest: 40.2 }
```

Here, the `calcDisplaySummary` function calculates and returns an object containing the total incomes, withdrawals, and interests from an array of financial movements.

We start with filtering the movements to separate incomes (positive values) and withdrawals (negative values). Then, we use `reduce` to sum up the respective values.

Finally, we calculate the interests earned from the incomes by first filtering the positive movements, then using `map` to calculate the interest for each income, filtering out the interests less than 1, and finally summing up all the interests using `reduce`.

Each operation naturally follows the previous one, thanks to chaining. This makes the code both elegant and easy to understand, emphasizing the power of JavaScript's array methods.

Conclusion: Mastering JavaScript Array Methods and Method Chaining

In this lecture, we've covered some of the most powerful and widely-used array methods in JavaScript: `forEach`, `map`, `filter`, and `reduce`. These ES6 methods, along with the principles of callback functions, provide us with efficient and elegant tools to manipulate and transform arrays.

Remember:

- `forEach` is a method that helps us loop over an array and execute a function on each element. It doesn't return any value, and unlike `map`, `filter`, and `reduce`, it may lead to "side effects", such as modifying the array itself or affecting external variables. This makes `forEach` suitable for tasks like triggering side effects or making API calls for each element in an array.

- `map` is a non-mutating method that creates a new array based on the results of applying a function to every element in the array. It's an indispensable tool for data transformation.
- `filter` gives us an efficient way to extract elements that meet a certain condition from an array, creating a new array with only the elements that satisfy the provided testing function.
- `reduce` is a highly versatile method that lets us boil down an array into a single output value - a number, a string, or even a new object or array.

On top of these, we also explored method chaining, a powerful technique that makes our code more readable and efficient. By leveraging the fact that `map`, `filter`, and `reduce` all return arrays, we can chain these methods together to perform complex data manipulation tasks in a single, fluid expression.

Becoming proficient with these methods and understanding when to use them and when to chain them will significantly improve your data handling capabilities in JavaScript. These tools form the backbone of data manipulation in many real-world applications, and mastering them is a major step towards becoming an expert JavaScript programmer.