Chapter 3.3: JavaScript Advanced Topics

Welcome to this exciting intermediate JavaScript tutorial! Now that you know the basics and how to work with the DOM, we're diving into some cooler, more advanced stuff like array methods, object-oriented programming, modules, and asynchronous coding. We'll explore powerful array functions, classes, reusable code files, fetching data from the internet, saving data in the browser, and even getting the user's location. You can test these examples in your browser's console (F12 or Ctrl+Shift+I, then Console) or in a simple HTML file like this:

```
<!DOCTYPE html>
<html>
<head>
    <title>JS Advanced Tutorial</title>
    <script></script>
</head>
<body>
    <div id="output"></div>
</body>
</html>
```

Put your JavaScript code in the `<script>` tag or a separate `.js` file. Let's jump in and make your coding skills shine!

1. Array Functions

JavaScript arrays come with awesome built-in methods that make working with lists super easy, like looping, searching, transforming, and combining data. These are perfect for functional programming!

forEach()

Runs a function for every item in an array without returning anything.

```
let fruits = ['apple', 'banana', 'cherry'];
fruits.forEach((item, index) => {
    console.log(`${index}: ${item}`);  // 0: apple, 1: banana, 2: cherry
});
```

Explanation: This code loops through the `fruits` array and prints each fruit with its position (index), like "0: apple". Think of `forEach` as a friendly robot that visits every item in your list and does something for you! Try this in your console: change the message to `Item ${index} is ${item}` and see what happens. You can play with different arrays, like numbers or names, to practice.

Example with objects (e.g., items with name, price, quantity):

```
let items = [
  { name: 'Apple', price: 1.5, quantity: 10 },
  { name: 'Banana', price: 0.5, quantity: 20 },
  { name: 'Cherry', price: 2.0, quantity: 5 }
];
items.forEach(item => {
    console.log(`${item.name}: $${item.price * item.quantity}`);  // Apple:
$15, Banana: $10, Cherry: $10
});
```

Explanation: Here, we loop through an array of objects (like a shopping list) and calculate the total cost for each item (price × quantity). Try this in your console: add a new item to the `items` array, like `{ name: 'Orange', price: 1.0, quantity: 8 }`, and see the new total. It's like checking the price of everything in your cart!

find()

Grabs the first item in an array that matches a condition, or `undefined` if nothing fits.

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

Explanation: This code searches the `numbers` array for the first number bigger than 10 and logs it (12). It stops at the first match, like finding the first big treasure in a chest! Try this in your console: change the condition to `num > 100` and see what you get. You can also try finding a negative number to test `undefined`.

Example with objects:

```
let expensiveItem = items.find(item => item.price > 1);
console.log(expensiveItem.name);  // Apple
```

Explanation: This looks for the first item in the `items` array with a price over 1 and logs its name ("Apple"). It's like spotting the first expensive item in your shop. Try this in your console: change the price limit to 0.4 or 2.5 and see which item pops up. Experiment with different conditions, like `item.quantity < 10`.

filter()

Creates a new array with only the items that pass a test.

```
let evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers);  // [12, 8, 44]
```

Explanation: This code makes a new array with only the even numbers from `numbers` by checking if they divide by 2 with no remainder. It's like picking only the even-numbered cards from a deck! Try this in your console: filter for numbers bigger than 20 or odd numbers (`num % 2 !== 0`). See how the new array changes.

Example with objects:

```
let lowStockItems = items.filter(item => item.quantity < 10);
console.log(lowStockItems);  // [{ name: 'Cherry', price: 2.0, quantity: 5
}]
```

Explanation: This filters the `items` array to show only items with less than 10 in stock, logging the result. It's like checking which groceries you're running low on! Try this in your console: change the condition to `item.quantity > 15` or `item.price < 1`. Add more items to `items` to practice filtering.

map()

Makes a new array by transforming each item with a function.

```
let doubled = numbers.map(num => num * 2);
console.log(doubled);  // [10, 24, 16, 260, 88]
```

Explanation: This creates a new array where every number in `numbers` is doubled. It's like giving every number a twin! Try this in your console: change the transformation to `num + 5` or `num * num` (square the numbers). Check out the new array each time to see how `map` works its magic.

Example with objects:

```
let itemTotals = items.map(item => ({
    name: item.name,
    totalPrice: item.price * item.quantity
}));
console.log(itemTotals);  // [{name: 'Apple', totalPrice: 15}, {name:
'Banana', totalPrice: 10}, {name: 'Cherry', totalPrice: 10}]
```

Explanation: This transforms the `items` array into a new array of objects with just the name and total price (price × quantity). It's like making a summary list of costs! Try this in your console: modify the object to include `quantity`, like `{ name: item.name, totalPrice: item.price * item.quantity, quantity: item.quantity }`. Log it and see the new structure.

reduce()

Combines an array into a single value, like adding everything up.

```

```

```
let sum = numbers.reduce((total, num) => total + num, 0);
console.log(sum);  // 199 (5 + 12 + 8 + 130 + 44)
```

Explanation: This adds all numbers in the `numbers` array, starting with 0, to get 199. It's like tallying up your game score! Try this in your console: change it to multiply numbers (`total * num`) or start with 10 instead of 0. See how `reduce` builds one final value.

Example with objects:

```
let totalValue = items.reduce((total, item) => total + (item.price *
item.quantity), 0);
console.log(totalValue);  // 35 (15 + 10 + 10)
```

Explanation: This calculates the total value of all items by adding up their price × quantity. It's like finding the total cost of your shopping list! Try this in your console: add a new item to `items` and see how the total changes. You can also try reducing to count quantities (`total + item.quantity`).

sort()

Rearranges the array in order, either alphabetically or with a custom function.

```
let names = ['Charlie', 'Alice', 'Bob'];
names.sort();  // ['Alice', 'Bob', 'Charlie']

let nums = [100, 2, 30];
nums.sort((a, b) => a - b);  // [2, 30, 100] (numeric sort)
console.log(nums);
```

Explanation: This sorts the `names` array alphabetically and the `nums` array numerically using a comparison function. Without the function, numbers sort like strings (weirdly!). Try this in your console: change `nums` to include negative numbers or sort `names` in reverse (`b.localeCompare(a)`). Watch how the array changes—it's like organizing your bookshelf!

Example with objects:

```
items.sort((a, b) => a.price - b.price);
console.log(items.map(item => item.name));  // ['Banana', 'Apple',
'Cherry'] (sorted by price ascending)
```

Explanation: This sorts the `items` array by price (lowest to highest) and logs the names in order. It's like arranging your groceries by price! Try this in your console: sort by quantity (`a.quantity - b.quantity`) or reverse the price sort (`b.price - a.price`). Check the new order with `items.map`.

Tip: `sort()` changes the original array; others create new arrays.

2. Prototypes for Functions

Prototypes are JavaScript's way of sharing features between objects, like passing down family traits. Every function has a `prototype` that objects created with `new` inherit from.

Example

```
function Person(name) {
    this.name = name;
}

Person.prototype.greet = function() {
    return `Hello, ${this.name}!`;
};

let alice = new Person('Alice');
console.log(alice.greet());  // Hello, Alice!

// Check prototype
console.log(alice.__proto__ === Person.prototype);  // true
```

Explanation: This code creates a `Person` constructor to make objects with a `name`. We add a `greet` function to the `Person.prototype` so all `Person` objects can use it. We make an `Alice` object and call `greet` to say "Hello, Alice!". The last line checks that `Alice`'s prototype matches `Person`'s. Try this in your console: create another person, like `new Person('Bob')`, and call `greet`. It's like giving all your objects a shared superpower!

Note: Prototypes save memory by sharing methods across objects.

3. Using Class and super

ES6 `class` makes object-oriented programming easier, like building blueprints for objects. Use `extends` and `super` to create child classes that inherit from parents.

Example

```
class Animal {
    constructor(name) {
        this.name = name;
    }
    speak() {
        return `${this.name} makes a sound.`;
    }
}

class Dog extends Animal {
    constructor(name, breed) {
        super(name);  // Call parent constructor
        this.breed = breed;
```

```
    }
    speak() {
        return `${this.name} barks!`;
    }
}

let dog = new Dog('Rex', 'Labrador');
console.log(dog.speak());  // Rex barks!
console.log(dog.breed);    // Labrador
```

Explanation: This code creates an `Animal` class with a `name` and a generic `speak` method. The `Dog` class inherits from `Animal` using `extends`, calls the parent's constructor with `super`, and adds a `breed`. The `Dog`'s `speak` overrides the parent's to bark. Try this in your console: make a new `Dog` or create a `Cat` class that meows. It's like teaching your animals new tricks!

Tip: `super` connects to the parent class's code.

4. JavaScript Modules (type="module")

Modules let you split your code into separate files, like organizing your notes into folders. Use `export` to share code and `import` to use it, with `<script type="module">`.

Example

math.js:

```
export function add(a, b) {
    return a + b;
}
export const PI = 3.14159;
```

Explanation: This `math.js` file shares an `add` function and a `PI` constant for other files to use. It's like lending your math tools to a friend! Save this as `math.js` and try the next example to use it.

index.html:

```
<script type="module">
    import { add, PI } from './math.js';
    console.log(add(2, 3));  // 5
    console.log(PI);         // 3.14159
</script>
```

Explanation: This HTML uses a module script to import `add` and `PI` from `math.js`, then logs their results. You'll need a local server (like VS Code's Live Server) to test this because of browser security. Try this: create `math.js`, add this to your HTML, and open it via a server. Change `add(2, 3)` to `add(10, 20)` or add a new export to `math.js`. It's like borrowing tools from another file!

Tip: Modules use strict mode and need a server for local testing (e.g., `http://localhost`).

5. Asynchronous Coding

Asynchronous code handles tasks that take time, like waiting for data from the internet, without freezing your program.

Promises: resolve, reject, Promise.all()

A `Promise` is like a ticket for a future result—it either works (`resolve`) or fails (`reject`).

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve('Done!'), 1000);  // Resolves after 1s
    // reject('Error!');  // Uncomment to test rejection
});

promise
    .then(result => console.log(result))  // Done!
    .catch(error => console.log(error));

// Promise.all: Run multiple promises in parallel
let p1 = Promise.resolve(1);
let p2 = Promise.resolve(2);
Promise.all([p1, p2]).then(values => console.log(values));  // [1, 2]
```

Explanation: This creates a promise that says "Done!" after 1 second and logs it. If you uncomment `reject`, it'll log an error instead. `Promise.all` runs two promises at once and logs their results together. Try this in your console: change the timeout to 2000 (2 seconds) or add a third promise to `Promise.all`. It's like waiting for multiple deliveries at once!

async/await

Makes async code look like regular code, using `async` functions and `await`.

```
async function fetchData() {
    try {
        let response = await new Promise(resolve => {
            setTimeout(() => resolve('Data received'), 1000);
        });
        console.log(response);  // Data received
    } catch (error) {
        console.log(error);
    }
}
fetchData();
```

Explanation: This `async` function waits for a promise to finish (after 1 second) and logs "Data received". The `try/catch` handles errors. Try this in your console: call `fetchData()` and change the timeout or message.

You can also make the promise `reject` an error to test the `catch`. It's like waiting for a package but writing the code as if it's already here!

Tip: Always use `try/catch` with `await` to catch errors.

6. Fetch API Using JSON Data

The Fetch API grabs data (like JSON) from the internet, returning a promise.

Example

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
    .then(response => response.json())  // Parse JSON
    .then(data => console.log(data.title))  // Log post title
    .catch(error => console.log('Error:', error));
```

Explanation: This code fetches a post from a free test API, turns the response into JSON, and logs the post's title. If something goes wrong, it logs an error. Try this in your console: change the post number (e.g., `/posts/2`) and see different titles. It's like ordering a specific book from an online library!

```
// Using async/await
async function getPost() {
    try {
        let response = await
fetch('https://jsonplaceholder.typicode.com/posts/1');
        let data = await response.json();
        console.log(data.title);
    } catch (error) {
        console.log('Error:', error);
    }
}
getPost();
```

Explanation: This does the same thing as above but uses `async/await` for cleaner code. It waits for the fetch, parses the JSON, and logs the title, with error handling. Try this in your console: call `getPost()` with different post numbers or test with a bad URL to see the error. It's like waiting for your book order in a simple way!

Example for displaying posts:

```
<!DOCTYPE html>
<html>
<head>
    <title>Display Posts</title>
</head>
<body>
    <div id="posts"></div>
```

```
    <script>
        async function displayPosts() {
            try {
                let response = await
fetch('https://jsonplaceholder.typicode.com/posts');
                let posts = await response.json();
                let postsDiv = document.getElementById('posts');
                posts.forEach(post => {
                    let titleElem = document.createElement('h3');
                    titleElem.textContent = post.title;
                    let bodyElem = document.createElement('p');
                    bodyElem.textContent = post.body;
                    postsDiv.appendChild(titleElem);
                    postsDiv.appendChild(bodyElem);
                });
            } catch (error) {
                console.log('Error fetching posts:', error);
            }
        }
        displayPosts();
    </script>
</body>
</html>
```

Explanation: This HTML and JavaScript fetches a list of posts from the API, creates `<h3>` titles and `<p>` bodies, and adds them to a `<div>` with `id="posts"`. If there's an error, it logs it. Try this in an HTML file: save it, open it in a browser, and see the posts appear. Try limiting to the first 5 posts (`posts.slice(0, 5).forEach`) or add styling with `titleElem.style.color = 'blue'`. It's like building a blog page from the internet!

Note: Make sure the API works (this one is public and CORS-compliant).

7. localStorage

`localStorage` saves data in the browser, like a notebook that doesn't disappear when you close the browser (up to ~5MB).

Example

```
// Save data
localStorage.setItem('username', 'Alice');
localStorage.setItem('settings', JSON.stringify({ theme: 'dark' }));

// Retrieve data
let username = localStorage.getItem('username');
console.log(username);  // Alice

let settings = JSON.parse(localStorage.getItem('settings'));
console.log(settings.theme);  // dark

// Remove or clear
```

```
localStorage.removeItem('username');
localStorage.clear();  // Clears all
```

Explanation: This code saves a username (“Alice”) and a settings object (as a JSON string) in `localStorage`. It retrieves and logs them, parsing the JSON to get the object’s `theme`. Then it removes the username and clears everything. Try this in your console: save a new key-value pair, like `localStorage.setItem('score', 100)`, and retrieve it. Check it stays after refreshing the page! It’s like a sticky note in your browser.

Tip: Use `JSON.stringify` and `JSON.parse` for objects or arrays.

8. navigator.geolocation

Gets the user’s location (latitude and longitude) if they allow it.

Example

```
if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(
        position => {
            console.log(`Lat: ${position.coords.latitude}, Lon:
${position.coords.longitude}`);
        },
        error => {
            console.log('Error:', error.message);
        }
    );
} else {
    console.log('Geolocation not supported');
}
```

Explanation: This checks if the browser supports geolocation, asks for the user’s location, and logs their latitude and longitude if permitted. If they say no or it fails, it logs an error. Try this in your console on a secure site (HTTPS): accept the location prompt and see the coordinates. Try denying permission to test the error. It’s like asking your browser, “Where am I?”—but you need permission!

Note: Works only on secure sites (HTTPS) and needs user consent.

Cheat Sheet

Topic	Description	Key Points/Examples
Array: forEach	Executes a function for each array element, no return.	`array.forEach((item, index) => console.log(item))`

Topic	Description	Key Points/Examples
Array: find	Returns first element matching a condition, else `undefined`.	`array.find(num => num > 10)`
Array: filter	Creates new array with elements passing a test.	`array.filter(num => num % 2 === 0)`
Array: map	Creates new array with results of function on each element.	`array.map(num => num * 2)`
Array: reduce	Reduces array to a single value using a function.	`array.reduce((total, num) => total + num, 0)`
Array: sort	Sorts array in place, optional comparison function.	`array.sort((a, b) => a - b)` (numeric sort)
Prototypes	Mechanism for sharing methods/properties across objects.	`Function.prototype.method = func; new Function().method()`
Class & super	ES6 syntax for object-oriented programming, inheritance.	`class Child extends Parent { constructor() { super(); } }`
Modules (type="module")	Split code into reusable files with `import`/`export`.	`<script type="module"> import { fn } from './file.js'</script>`
Promises	Handle async operations with resolve/reject states.	`new Promise((resolve, reject) => resolve('Done'))` `Promise.all([p1, p2])`
async/await	Simplify promise handling with synchronous-like syntax.	`async function fn() { let x = await promise; }`

Topic	Description	Key Points/Examples
Fetch API	Fetch resources (e.g., JSON) from a URL.	`fetch(url).then(res => res.json())` `async () => await (await fetch(url)).json()`
localStorage	Store key-value pairs in browser, persists across sessions.	`localStorage.setItem('key', 'value')` `localStorage.getItem('key')`
navigator.geolocation	Access user's location with permission.	`navigator.geolocation.getCurrentPosition(pos => pos.coords)`