Lists

Week 3

> Adib Sakhawat
> IUT CSE ‘21

Using Arrays as Lists in JavaScript

JavaScript arrays are dynamic and versatile, making them ideal for handling lists. They can store elements of any type, including numbers, strings, objects, and even other arrays. This guide will cover how to use arrays as lists, including adding, removing, and iterating over items using array methods.

Creating Arrays

You can create arrays using:

Array Literals

The most common way to create an array is by using square brackets `[]`.

```
const emptyArray = [];
const numbers = [1, 2, 3];
const mixedArray = [1, 'two', true, { key: 'value' }];
```

Array Constructor

You can also use the `Array` constructor, though it's less common due to some quirks.

```
const numbers = new Array(1, 2, 3);
```

Edge Case: Single Numeric Argument

If you pass a single number to the `Array` constructor, it creates an empty array with that length.

```
const arr = new Array(3);
console.log(arr.length); // Output: 3
console.log(arr); // Output: [ <3 empty items> ]
```

Adding Items to Arrays

1. Using `push()`

Adds one or more elements to the **end** of an array.

```
const fruits = ['apple', 'banana'];
fruits.push('orange');
console.log(fruits); // Output: ['apple', 'banana', 'orang
e']
```

Adding Multiple Items:

```
fruits.push('grape', 'pineapple');
console.log(fruits); // Output: ['apple', 'banana', 'orang
e', 'grape', 'pineapple']
```

2. Using `unshift()`

Adds one or more elements to the **beginning** of an array.

```
const fruits = ['banana', 'orange'];
fruits.unshift('apple');
console.log(fruits); // Output: ['apple', 'banana', 'orang
e']
```

3. Using `splice()`

Inserts elements at a specific index.

```
const fruits = ['apple', 'banana', 'orange'];
fruits.splice(1, 0, 'grape'); // At index 1, delete 0 item
s, then insert 'grape'
```

```
console.log(fruits); // Output: ['apple', 'grape', 'banan
a', 'orange']
```

Edge Case: Negative Indices

Negative indices count from the end of the array.

```
fruits.splice(-1, 0, 'lemon'); // Insert 'lemon' before the
last item
console.log(fruits); // Output: ['apple', 'grape', 'banan
a', 'lemon', 'orange']
```

Removing Items from Arrays

1. Using `pop()`

Removes the last element and returns it.

```
const fruits = ['apple', 'banana', 'orange'];
const lastFruit = fruits.pop();
console.log(lastFruit); // Output: 'orange'
console.log(fruits); // Output: ['apple', 'banana']
```

2. Using `shift()`

Removes the first element and returns it.

```
const fruits = ['apple', 'banana', 'orange'];
const firstFruit = fruits.shift();
console.log(firstFruit); // Output: 'apple'
console.log(fruits); // Output: ['banana', 'orange']
```

3. Using `splice()`

Removes elements at a specific index.

```
const fruits = ['apple', 'banana', 'orange'];
const removedFruits = fruits.splice(1, 1); // At index 1, r
emove 1 item
```

```
console.log(removedFruits); // Output: ['banana']
console.log(fruits); // Output: ['apple', 'orange']
```

4. Using `filter()`

Creates a new array without the items that match the condition.

```
const numbers = [1, 2, 3, 4, 5];
const oddNumbers = numbers.filter(num => num % 2 !== 0);
console.log(oddNumbers); // Output: [1, 3, 5]
```

Iterating Over Arrays

1. Using `for` Loop

```
const fruits = ['apple', 'banana', 'orange'];
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}
// Output:
// 'apple'
// 'banana'
// 'orange'
```

2. Using `for...of` Loop

```
for (const fruit of fruits) {
  console.log(fruit);
}
// Output:
// 'apple'
// 'banana'
// 'orange'
```

3. Using `forEach()`

```
fruits.forEach((fruit, index) => {
  console.log(`${index}: ${fruit}`);
});
// Output:
// '0: apple'
// '1: banana'
// '2: orange'
```

4. Using `map()`

Transforms each element and returns a new array.

```
const uppercasedFruits = fruits.map(fruit => fruit.toUpperCase());
console.log(uppercasedFruits); // Output: ['APPLE', 'BANANA', 'ORANGE']
```

5. Using `reduce()`

Reduces the array to a single value.

```
const numbers = [1, 2, 3, 4, 5];
const total = numbers.reduce((sum, num) => sum + num, 0);
console.log(total); // Output: 15
```

Common Array Methods

1. `concat()`

Combines two or more arrays.

```
const array1 = [1, 2];
const array2 = [3, 4];
const combinedArray = array1.concat(array2);
console.log(combinedArray); // Output: [1, 2, 3, 4]
```

2. `slice()`

Returns a shallow copy of a portion of an array.

```
const fruits = ['apple', 'banana', 'orange', 'grape'];
const citrus = fruits.slice(1, 3); // From index 1 up to bu
t not including index 3
console.log(citrus); // Output: ['banana', 'orange']
```

Edge Case: Negative Indices

```
const lastTwoFruits = fruits.slice(-2);
console.log(lastTwoFruits); // Output: ['orange', 'grape']
```

3. `indexOf()` and `lastIndexOf()`

Finds the index of an element.

```
const animals = ['cat', 'dog', 'cow', 'dog'];
console.log(animals.indexOf('dog')); // Output: 1
console.log(animals.lastIndexOf('dog')); // Output: 3
```

4. `includes()`

Checks if an array includes a certain value.

```
const numbers = [1, 2, 3];
console.log(numbers.includes(2)); // Output: true
console.log(numbers.includes(4)); // Output: false
```

5. `find()` and `findIndex()`

Finds an element or its index based on a test function.

```
const people = [{ name: 'Alice' }, { name: 'Bob' }, { name:
'Charlie' }];
const person = people.find(p => p.name === 'Bob');
console.log(person); // Output: { name: 'Bob' }
```

```
const index = people.findIndex(p => p.name === 'Charlie');
console.log(index); // Output: 2
```

6. `some()` and `every()`

Tests whether some or all elements pass a test.

```
const numbers = [1, 2, 3, 4, 5];
const hasEven = numbers.some(num => num % 2 === 0);
console.log(hasEven); // Output: true

const allPositive = numbers.every(num => num > 0);
console.log(allPositive); // Output: true
```

7. `sort()`

Sorts elements in place.

```
const letters = ['b', 'c', 'a'];
letters.sort();
console.log(letters); // Output: ['a', 'b', 'c']
```

Edge Case: Numeric Sorting

```
const numbers = [10, 2, 30, 4];
numbers.sort();
console.log(numbers); // Output: [10, 2, 30, 4] // Incorrec
t order

numbers.sort((a, b) => a - b);
console.log(numbers); // Output: [2, 4, 10, 30]
```

8. `reverse()`

Reverses the array in place.

```
const numbers = [1, 2, 3];
numbers.reverse();
```

```
console.log(numbers); // Output: [3, 2, 1]
```

9. `join()`

Joins all elements into a string.

```
const words = ['Hello', 'world'];
const sentence = words.join(' ');
console.log(sentence); // Output: 'Hello world'
```

Edge Cases and Best Practices

Modifying Arrays While Iterating

Be careful when changing an array inside a loop.

Example:

```
const numbers = [1, 2, 3, 4, 5];
numbers.forEach((num, index) => {
  if (num % 2 === 0) {
    numbers.splice(index, 1); // Removes even numbers
  }
});
console.log(numbers); // Output may be unexpected due to in
dex shifts
```

Solution: Iterate Backwards or Use `filter()`:

```
for (let i = numbers.length - 1; i >= 0; i--) {
  if (numbers[i] % 2 === 0) {
    numbers.splice(i, 1);
  }
}
console.log(numbers); // Output: [1, 3, 5]
```

Using `filter()`:

```
const oddNumbers = numbers.filter(num => num % 2 !== 0);
console.log(oddNumbers); // Output: [1, 3, 5]
```

Copying Arrays

Assignment copies the reference, not the array.

```
const original = [1, 2, 3];
const copy = original;
copy.push(4);
console.log(original); // Output: [1, 2, 3, 4]
```

Creating a Shallow Copy:

```
const shallowCopy = original.slice();
shallowCopy.push(5);
console.log(original);     // Output: [1, 2, 3, 4]
console.log(shallowCopy);  // Output: [1, 2, 3, 4, 5]
```

Using Spread Operator:

```
const spreadCopy = [...original];
```

Deep Copying Arrays

For arrays containing objects or other arrays.

```
const nestedArray = [1, [2, 3]];
const deepCopy = JSON.parse(JSON.stringify(nestedArray));
deepCopy[1].push(4);
console.log(nestedArray); // Output: [1, [2, 3]]
console.log(deepCopy);    // Output: [1, [2, 3, 4]]
```

Edge Case: Functions and Special Objects

`JSON.parse(JSON.stringify())` doesn't work with functions or special object types.

Working with Array-Like Objects

Some objects (like `arguments` or NodeLists) are array-like but lack array methods.

Converting to Arrays

Using `Array.from()`:

```
function example() {
  const argsArray = Array.from(arguments);
  console.log(argsArray);
}
example(1, 2, 3); // Output: [1, 2, 3]
```

Using Spread Operator (if iterable):

```
const nodeList = document.querySelectorAll('div');
const nodesArray = [...nodeList];
```

Handling Edge Cases

Sparse Arrays

Arrays with missing indices.

```
const sparseArray = [1, , 3]; // Missing index 1
console.log(sparseArray.length); // Output: 3
console.log(sparseArray[1]);     // Output: undefined
```

Iterating Over Sparse Arrays:

```
sparseArray.forEach((value, index) => {
  console.log(`${index}: ${value}`);
});
// Output:
// '0: 1'
// '2: 3'
```

Edge Case: Methods Skipping Missing Elements

Methods like `forEach()`, `map()`, `filter()`, and `reduce()` skip missing elements but not elements with `undefined` values.

Using `hasOwnProperty()` to Check for Existence

```
for (let i = 0; i < sparseArray.length; i++) {
  if (sparseArray.hasOwnProperty(i)) {
    console.log(`${i}: ${sparseArray[i]}`);
  }
}
```