# JavaScript Lecture: Objects & Arrays (Updated)

## Introduction: Objects and Arrays

Objects and arrays are fundamental data structures in JavaScript. Objects: Collections of key-value pairs (properties). Useful to model real-world entities. - Keys are strings (or symbols); values can be any type, including functions and other objects. Arrays: Ordered lists of values accessed by numeric indices. Arrays are specialized objects with array methods. Key points: - Arrays are zero-indexed and have a `length` property. - `typeof` an array returns 'object' — use `Array.isArray()` to check for arrays. - Both objects and arrays are mutable by default (you can change them in place), but immutability patterns are common in modern JS.

```
// Object example
const person = {
  name: "Alice",
  age: 25,
  languages: ["English", "French"],
  address: { city: "Paris", zip: "75000" }
};

console.log(person.name);            // Alice
console.log(person["age"]);          // 25
console.log(person.languages[0]);    // English
console.log(person.address.city);    // Paris

// Array example
const fruits = ["apple", "banana", "cherry"];
console.log(fruits[0]);              // apple
console.log(fruits.length);          // 3
console.log(typeof fruits);          // "object"
console.log(Array.isArray(fruits));  // true

// Objects and arrays together
const data = { users: [{ id: 1, name: "Alice" }, { id: 2, name: "Bob" }] };
console.log(data.users[0].name);     // Alice
```

## 1. Object Literals & Destructuring

Objects in JavaScript are collections of key-value pairs. Object literals are the most common way to create objects. Destructuring allows extracting values from objects into variables easily.

```
// Object literal
const person = { name: "Alice", age: 25, city: "Paris" };

// Destructuring (basic)
const { name, age } = person;
console.log(name); // Alice
console.log(age);  // 25

// Destructuring with default values and renaming
const { city, country = "Unknown", name: firstName } = person;
console.log(city);       // Paris
console.log(country);    // Unknown
console.log(firstName);  // Alice
```

## 2. Array Methods (map, filter, reduce, find, some, every)

Arrays provide powerful methods for iteration and transformation. Use them instead of manual loops to write cleaner code.

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

// map -> transform values
const doubled = numbers.map(n => n * 2); // [2,4,6,8,10]

// filter -> keep values matching condition
const evens = numbers.filter(n => n % 2 === 0); // [2,4]

// reduce -> accumulate values (sum)
const sum = numbers.reduce((acc, n) => acc + n, 0); // 15

// find -> first element matching condition
const firstEven = numbers.find(n => n % 2 === 0); // 2

// some -> check if any element matches
const hasEven = numbers.some(n => n % 2 === 0); // true

// every -> check if all elements match
const allPositive = numbers.every(n => n > 0); // true

console.log({ doubled, evens, sum, firstEven, hasEven, allPositive });
```

## 3. Spread & Rest Operators

Spread (...) expands elements (arrays or object properties). Rest (...) collects remaining elements into an array (used in function parameters or destructuring).

```
// Spread examples (arrays)
const arr1 = [1, 2, 3];
const arr2 = [...arr1, 4, 5];
console.log(arr2); // [1,2,3,4,5]

// Spread with objects (shallow copy / merging)
const obj = { a: 1, b: 2 };
const newObj = { ...obj, b: 42, c: 3 };
console.log(newObj); // { a:1, b:42, c:3 }

// Rest parameters in functions
function sum(...nums) {
  return nums.reduce((acc, n) => acc + n, 0);
}
console.log(sum(1, 2, 3, 4)); // 10

// Rest in destructuring (arrays)
const [first, ...rest] = [10, 20, 30, 40];
console.log(first); // 10
console.log(rest);  // [20,30,40]
```

## 4. Object/Array Immutability

Immutability means not modifying the original object/array; instead create a new one. This is useful in functional programming and React state updates.

```
const obj = { name: "Alice", age: 25 };

// Mutable update (changes original)
obj.age = 26;

// Immutable update (create a new object)
const updatedObj = { ...obj, age: 26 };
console.log(obj, updatedObj);
```

```
// Immutable array update
const arr = [1, 2, 3];
// Mutable: arr.push(4) modifies arr
const newArr = [...arr, 4]; // does not change original arr
console.log(arr, newArr);

// Deep immutability note: spread only shallow-copies; nested objects still reference same inner obje
const nested = { a: { x: 1 } };
const shallowCopy = { ...nested };
shallowCopy.a.x = 99;
console.log(nested.a.x); // 99 -> shows shallow copy shares nested reference

// For deep clones use structuredClone (modern) or JSON or libraries
// const deep = structuredClone(nested);
```