Day 2 — Control Flow + Arrays & Objects

---

1. Control Flow

1.1 if / else if / else

What it does: Used to execute different blocks of code depending on whether a condition is true or false.

```
const score = 85;
if (score >= 90) {
  console.log('A');
} else if (score >= 80) {
  console.log('B');
} else {
  console.log('C or below');
}
```

When to use: When you need simple branching logic based on numeric ranges or comparisons.

1.2 switch

What it does: Used when you have one value and several possible cases to compare it against.

```
const day = 'Mon';
switch (day) {
  case 'Mon':
    console.log('Start of the week');
    break;
  case 'Fri':
    console.log('Weekend is near');
    break;
  default:
    console.log('Midweek');
}
```

When to use: When checking one variable against multiple discrete values.

1.3 for loop

What it does: Repeats a block of code a specific number of times.

```
for (let i = 0; i < 5; i++) {
  console.log('Number:', i);
}
```

When to use: When you know exactly how many iterations are needed or you need an index.

1.4 while loop

What it does: Runs a block of code as long as a condition remains true.

```
let count = 0;
while (count < 3) {
  console.log('Count is', count);
  count++;
}
```

When to use: When you don't know how many times the loop will run but depend on a condition.

1.5 for...of

What it does: Iterates over the values of an iterable object such as an array or string.

```
const fruits = ['Apple', 'Banana', 'Orange'];
for (const fruit of fruits) {
  console.log(fruit);
}
```

When to use: To go through each value in an array without needing the index.

1.6 for...in

What it does: Iterates over the keys (property names) of an object.

```
const user = { name: 'Ali', age: 25, city: 'Cairo' };
for (const key in user) {
```

```
  console.log(key, ':', user[key]);
}
```

When to use: When you need to loop through all properties of an object.

---

2. Arrays

2.1 map()

What it does: Creates a new array by applying a function to every element.

```
const nums = [1, 2, 3];
const doubled = nums.map(x => x * 2);
console.log(doubled); // [2, 4, 6]
```

When to use: When you want to transform all elements in an array.

2.2 filter()

What it does: Creates a new array with elements that pass a specific test.

```
const numbers = [10, 25, 30, 15];
const above20 = numbers.filter(n => n > 20);
console.log(above20); // [25, 30]
```

When to use: When you need to keep only certain elements from an array.

2.3 reduce()

What it does: Combines all array elements into a single value using an accumulator.

```
const values = [5, 10, 15];
const total = values.reduce((sum, val) => sum + val, 0);
console.log(total); // 30
```

When to use: For totals, averages, or building objects from arrays.

2.4 some()

What it does: Checks if at least one element in the array passes the test.

```
const ages = [15, 22, 13];
const hasAdult = ages.some(a => a >= 18);
console.log(hasAdult); // true
```

When to use: To see if any element meets a condition.

2.5 every()

What it does: Checks if all elements in the array satisfy a condition.

```
const allAdults = ages.every(a => a >= 18);
console.log(allAdults); // false
```

When to use: To confirm all elements pass a rule.

2.6 find()

What it does: Finds and returns the first element that matches a condition.

```
const students = [
  { name: 'Ali', grade: 90 },
  { name: 'Sara', grade: 70 }
];
const topStudent = students.find(s => s.grade > 80);
console.log(topStudent); // { name: 'Ali', grade: 90 }
```

When to use: When you need a single match instead of a list.

2.7 flat()

What it does: Flattens nested arrays into a single array.

```
const nested = [1, [2, 3], [4, [5]]];
console.log(nested.flat(2)); // [1, 2, 3, 4, 5]
```

When to use: When you have nested arrays and need one flat list.

---

3. Objects

3.1 Creating and accessing

What it does: Defines and retrieves key-value pairs.

```
const person = { name: 'Ali', age: 25 };
console.log(person.name); // Ali
console.log(person['age']); // 25
```

When to use: For structured data with named properties.

3.2 Spread operator

What it does: Copies or merges objects without changing the original.

```
const user1 = { name: 'Ali', age: 25 };
const user2 = { ...user1, city: 'Cairo' };
console.log(user2); // { name: 'Ali', age: 25, city: 'Cairo' }
```

When to use: To duplicate or extend objects immutably.

3.3 Destructuring

What it does: Extracts specific values from objects or arrays into variables.

```
const employee = { id: 1, name: 'Sara', position: 'Developer' };
const { name, position } = employee;
console.log(name, position); // Sara Developer
```

When to use: To simplify access to object properties or array elements.

3.4 Optional chaining (?.) and nullish coalescing (??)

What it does: Accesses deeply nested properties safely and provides default values.

```
const userInfo = { name: 'Ali', address: { city: 'Cairo' } };
console.log(userInfo.address?.city); // Cairo
console.log(userInfo.contact?.phone ?? 'No phone'); // No phone
```

When to use: To avoid runtime errors when working with uncertain or optional data.

---

4. Safe Object Merge

What it does: Combines two objects deeply, ensuring nested properties are merged correctly.

```
const a = { x: 1, details: { color: 'red' } };
const b = { details: { size: 'M' } };
function deepMerge(obj1, obj2) {
  const result = { ...obj1 };
  for (const [key, value] of Object.entries(obj2)) {
    if (value && typeof value === 'object' && !Array.isArray(value)) {
      result[key] = deepMerge(obj1[key] || {}, value);
    } else {
      result[key] = value;
    }
  }
  return result;
}

console.log(deepMerge(a, b));
// { x:1, details:{ color:'red', size:'M' } }
```

When to use: When merging complex configurations or nested data structures safely.