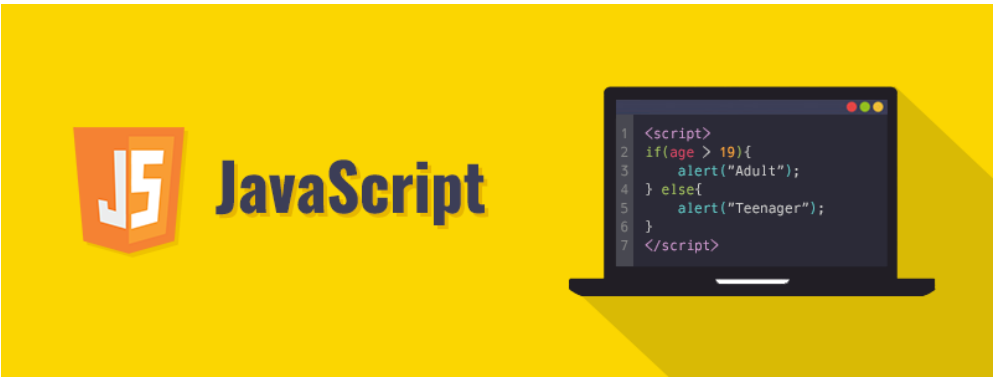
JavaScript Foundation – Training Day-2



Course content and duration:

Duration: 16 hours | Schedule: 8 days @ 2 hours/day

S. No	Day	Module	Topics
1	WED (10/12/2025)	Module 1: Introduction to JavaScript and Basics	JavaScript Overview, Syntax, and Variables
2	FRI (12/12/2025)	Module 2: Control Flow and Loops	Conditional Statements and Iteration
3	MON (15/12/2025)	Module 3: Functions and Scope	Function Declaration, Expressions, and Scope
4	TUE (16/12/2025)	Module 4: Arrays and Array Methods	Array Manipulation and Higher-Order Functions
5	WED (17/12/2025)	Module 5: Objects and Object-Oriented Programming	Objects, Properties, Methods, and Prototypes
6	THU (18/12/2025)	Module 6: DOM Manipulation and Events	Document Object Model and Event Handling
7	FRI (19/12/2025)	Module 7: Asynchronous JavaScript	Callbacks, Promises, and Async/Await
8	MON (22/12/2025)	Module 8: ES6+ Features and Best Practices	Modern JavaScript Features and Code Quality

Day2: Conditional Statements and Iteration

Contents
If-else statements and nested conditions
Switch statements
Ternary operator
For loops (standard, for...in, for...of)
While and do-while loops
Break and continue statements
Loop best practices and patterns
Common looping scenarios

What Are Conditional Statements?

- Allow code to make decisions
- Execute different blocks based on conditions (true/false)
- Foundation of program logic

Conditional statements allow your program to behave differently depending on user input, variable values, and real-time conditions.

They are like traffic lights—your program decides what to do next.

Summary

- If–else (if, else, if-else, else-if) → decision-making
- Switch → clean multi-case decision
- Ternary → shorthand decision
- Loops → repeat actions
- for…in → keys
- for…of → values
- while & do–while → unknown iteration counts
- break & continue → control loop flow

Today we'll explore how JavaScript makes decisions (conditions) and repeats tasks (loops). These are core skills for problem solving.

The if Statement: Used to execute code based on conditions.

The if statement is evaluated first. If the condition is true, only that block runs. Conditions must return a boolean value (true or false).

```
if (condition) {
    // code runs when condition is true
}
```

Example:

```
let age = 20;

if (age >= 18) {
    console.log("You are an adult.");
}
```

if-else Statement:

When the condition is false, the else part provides an alternate path.

```
if (condition) {
  // true block
} else {
  // false block
}
```

Example:

```
if (score >= 50) {
    console.log("Pass");
} else {
    console.log("Fail");
}
```

else-if Statement:

Use when more than two decisions are needed.

Execution stops at the first true condition; the rest are ignored.

```
let temp = 32;

if (temp > 40) {
    console.log("Very Hot");
} else if (temp > 30) {
    console.log("Hot");
} else if (temp > 20) {
    console.log("Warm");
} else {
    console.log("Cold");
}
```

Nested Conditions

When a condition is inside another condition.

Nesting is powerful but can get messy. Use it only when logic requires multiple checks within one category.

```
let score = 85;

if (score >= 90) {
    console.log("Grade A");
} else {
    if (score >= 75) {
        console.log("Grade B");
    } else {
        console.log("Grade C");
    }
}
```

Nested ifs get messy quickly. Prefer else-if for readability.

```
if (score >= 90) console.log("A");
else if (score >= 75) console.log("B");
else console.log("C");
```

Avoid Too Much Nesting

Hard to read
Prefer else if ladder or small functions

Too many nested blocks create what developers call *'arrow-shaped code'* or *'pyramid of doom'.*

What is "Arrow-Shaped Code" or "Pyramid of Doom"?

When your code contains **too many nested blocks**—usually from multiple if statements, loops, callbacks, or try-catch blocks—it starts to shift right like an arrow or pyramid:

```
if (condition1) {
    if (condition2) {
        if (condition3) {
            if (condition4) {
                // actual logic here
            }
        }
    }
}
```

This shape makes code:

- Hard to **read**
- Hard to **debug**
- Hard to **extend**
- Easy to introduce **bugs**

Developers call this **"arrow-shaped code"**, **"pyramid of doom"**, or **"callback hell"** (in asynchronous code).

Example: Bad Code (Arrow-Shaped)

Suppose we validate user input before saving:

```
function saveUser(user) {
    if (user) {
        if (user.name) {
            if (user.email) {
                if (user.email.includes("@")) {
                    console.log("User saved!");
                } else {
                    console.log("Invalid email format.");
                }
            } else {
                console.log("Email is required.");
            }
        } else {
            console.log("Name is required.");
        }
    } else {
        console.log("User object missing.");
    }
}
```

Why it's bad:

- Too many levels of indentation
- Hard to see the **main logic**
- You must mentally follow each nested branch

This is what we want to **avoid early**.

Refactored Version (Flattening the Pyramid)

Strategy: Use early returns to exit quickly when something is invalid.

```
function saveUser(user) {
    if (!user) {
        console.log("User object missing.");
        return;
    }

    if (!user.name) {
        console.log("Name is required.");
        return;
    }

    if (!user.email) {
        console.log("Email is required.");
        return;
    }

    if (!user.email.includes("@")) {
        console.log("Invalid email format.");
        return;
    }

    // ✓ Clear main logic
    console.log("User saved!");
}
```

Why this is better:

- Only **one level deep**
- Each check is isolated → easier to understand
- Main logic is at the **bottom and clearly visible**
- No mental gymnastics

“Deep nesting is a warning sign. If you see code drifting right, refactor.”

“Humans read code top-to-bottom. Don’t hide the main logic 10 levels deep.”

Logical Operators in Conditions

Logical operators help reduce nesting by combining related conditions.

&& (and) ||(or) ! (not)

Example:

```
let age = 25;
let hasID = true;

if (age >= 18 && hasID) {
    console.log("Entry allowed");
}
```

What Are Logical Operators?

Logical operators combine multiple conditions into a single expression.

Operators:

- **AND → &&**
- **OR → ||**
- **NOT → !**

Logical operators help you build decisions that depend on more than one condition. They return true or false and work seamlessly inside any conditional statement.

- && — all conditions must be true
- || — at least one must be true
- ! — reverses a boolean
- Supports short-circuit evaluation
- Reduces nesting and improves clarity

AND → &&

AND returns true only when **both** operands are true

If the first condition is false → the second is **not evaluated**

```
let hasLicense = true;

if (age >= 18 && hasLicense) {
    console.log("You can drive.");
}
```

OR → ||

OR returns true if **any** condition is true

If the first condition is true → the second is **skipped**

```
let isWeekend = false;
let isHoliday = true;

if (isWeekend || isHoliday) {
    console.log("You can relax!");
}
```

NOT → !

NOT is useful for checking the opposite of a condition

Common pattern: if (!user) { ... }

```
let isLoggedIn = false;

if (!isLoggedIn) {
    console.log("Please log in.");
}
```

Next topic: Switch Statements

Switch Statements:

What is a Switch Statement?

A switch statement is used to replace multiple if-else conditions when you are comparing the same value against different possible matches.

Syntax:

```
switch (expression) {
  case value1:
    // code
    break;
  case value2:
    // code
    break;
  default:
    // code
}
```

It checks the value once and jumps directly to the matching case.

Always mention why **break** is needed — JavaScript will continue executing the next cases if break is missing.

Example:

```
let day = 3;

switch (day) {
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    console.log("Wednesday");
    break;
  default:
    console.log("Invalid day");
}
```

What happen if we forget break, which makes switch behave unexpectedly and without break, the next cases would also run.

```
let fruit = "apple";

switch (fruit) {
  case "apple":
    console.log("It's red");
  case "banana":
    console.log("It's yellow");
  default:
    console.log("Unknown fruit");
}
```

Multiple cases sharing the same logic is **cleaner than repeating code**.

```
let grade = "A";

switch (grade) {
  case "A":
    console.log("Pass");
    break;
  case "B":
    console.log("Pass");
    break;
  case "C":
    console.log("Pass");
    break;
  case "D":
    console.log("Fail");
    break;
  default:
    console.log("Invalid grade");
}
```

Switch vs If-Else

Use switch when:

- You compare **the same variable** against **many fixed values**.
- Values are simple (strings, numbers).
- You want cleaner, more readable branching.

Use if-else when:

- You are evaluating **ranges** (e.g., age > 18).
- You need **multiple logical conditions** (&&, ||).
- Expressions are dynamic.

Example of when NOT to use switch:

Switch cannot handle complex expressions like this.

```
if (temperature > 30 && humidity > 70) {
  console.log("Hot and humid");
}
```

Best Practices

✓ Always use break unless deliberate fall-through
✓ Group cases that share logic
✓ Add comments for intentional fall-through
✓ Use switch for value matching, not complex logic
✓ Keep switch statements short and readable

How switch jumps directly to a case:

```
1 → case 1
2 → case 2
3 → case 3
```

When you run a switch with value 3, JavaScript **does not check case 1, then case 2**. It **jumps directly to case 3**, like jumping to a labeled section in a book.

How This Proves the Direct Jump

But **you only see the statement inside case 3**, because:

- switch (day) checks the value 3
- Jumps straight to the matching case 3
- Runs the code inside it
- Stops at break

But if-else will execute line-by-line.

Example:

```
if (color === "red") {
  console.log("Stop");
} else if (color === "yellow") {
  console.log("Slow Down");
} else if (color === "green") {
  console.log("Go");
} else {
  console.log("Invalid Color");
}
```

Next topic: Ternary operator

Ternary Operator (?:) in JavaScript

Short form of if–else.

What is the ternary operator?

The **ternary operator** is a **short, concise** way of writing simple *if-else* conditions in a single line.
It is also called the **conditional operator**.

Styntax:

```
condition ? expressionIfTrue : expressionIfFalse;
```

How it reads:

- "If the **condition** is true, return **expressionIfTrue**
- otherwise return **expressionIfFalse**."

Example — If/Else vs Ternary:

If/Else form

```
let age = 20;
let message;

if (age >= 18) {
    message = "You are an adult";
} else {
    message = "You are a minor";
}

console.log(message);
```

Ternary form (same logic, shorter)

```
let age = 20;
let message = age >= 18 ? "You are an adult" : "You are a minor";
console.log(message);
```

When to Use the Ternary Operator

✓ Good Use Cases

- Simple decision-making
- Assigning values based on condition
- Returning quick results
- Inline UI rendering (React/JS frameworks)

```
let isLoggedIn = true;
console.log(isLoggedIn ? "Welcome User!" : "Please Login");
```

✗ Avoid When

- Logic is long or has many steps
- Condition is hard to understand
- The readability becomes worse than an if/else

Nested Ternary Operators

You *can* nest ternaries, but avoid making them too complex.

```
let marks = 85;
let grade =
    marks >= 90 ? "A" :
    marks >= 75 ? "B" :
    marks >= 50 ? "C";

console.log(grade);
```

Too many nested ternaries can make code *hard to read*. Use them only when the logic stays simple.

Next chapter: Loops

Introduction to Loops

Loops repeat actions until a condition is met.

Why Do We Use Loops?

Imagine printing numbers from 1 to 100.

Without loops:

```
console.log(1);
console.log(2);
console.log(3);
// ... until 100
```

With loops:

```
for (let i = 1; i <= 100; i++) {
  console.log(i);
}
```

Key Benefits

- Reduce code repetition
- Make programs scalable
- Improve maintainability
- Automate repeated tasks

Types of Loops in JavaScript

JavaScript supports several loops:

Loop Type	When to Use
for	Know the number of iterations in advance
while	Continue until a condition becomes false
do...while	Execute at least once, then repeat
for...of	Iterate through arrays or iterable objects
for...in	Iterate through object properties

The Basic Loop Structure

Every loop contains three parts:

1. **Initialization** – Start value
2. **Condition** – When to stop
3. **Increment/Decrement** – How to move each step

Example:

```
for (let i = 0; i < 5; i++) {
  console.log("Loop count:", i);
}
```

Flow of a Loop:

```
Start
  ↓
Initialization → Condition true? → Yes → Run code → Update → Back to condition
                                    ↓
                                   No
                                    ↓
                                  End
```

For Loop (Standard)

Best for when you know how many times to loop. This is the most common loop structure.

Example:

```
for (let i = 1; i <= 5; i++) {
    console.log("Number:", i);
}
```

for…in Loop

Used to iterate **keys** of an object.

```
let user = {name: "Alex", age: 25, city: "Delhi"};

for (let key in user) {
    console.log(key, user[key]);
}
```

Do NOT use for…in for arrays—order is not guaranteed.

for…of Loop

Used to iterate **values** of iterable objects (arrays, strings, maps, sets).

Much cleaner than classic for loops for arrays.

```
let nums = [10, 20, 30];

for (let value of nums) {
    console.log(value);
}
```

```
for (let char of "forof") {
  console.log(char);
}
```

While Loop

Runs while condition is true. Use when the number of iterations is unknown.

```
let i = 1;
while (i <= 5) {
    console.log(i);
    i++;
}
```

Do–While Loop

Executes at least once.

```
let count = 1;
do {
    console.log("Run:", count);
    count++;
} while (count <= 3);
```

✓ Code executes once

✓ Then condition is checked

✓ Condition is false → loop stops

When to Use Do–While

Use it when:

- You need the code to run **at least once**
- You want to show a message once before validation
- You want to collect user input and then check it

```
let password;

do {
  password = prompt("Enter your password:");
} while (password !== "admin123");
```

The prompt **must appear at least once**, so *do–while* is the right choice.

Do–while:
You *always* taste food first (**do**) → *then* decide if you want more (**while condition is true**).

While:
You check if you're hungry (**condition**) → if yes, you start eating.

Next topic: Break & Continue

Break & Continue in JavaScript:

break Statement

The break statement **immediately stops** the loop (or switch) and **jumps out of it completely**.

👉 When do we use it?

- When a **condition is met** and there's no need to continue looping
- To **stop searching** once the target is found
- To **avoid unnecessary iterations**

✓ Example: Stop at the first multiple of 7

```
for (let i = 1; i <= 20; i++) {
  if (i % 7 === 0) {
    console.log("First multiple of 7 is:", i);
    break; // Exit the loop immediately
  }
}
```

```
const names = ["John", "Sita", "Aman", "Priya", "Kiran"];

for (let name of names) {
  if (name === "Priya") {
    console.log("Found Priya!");
    break;
  }
}
```

continue Statement

The continue statement skips the current iteration and moves to the next iteration.

👉 **When do we use it?**

- **When you want to ignore certain values**
- **When filtering out unwanted data**
- **When skipping invalid input**

✓ **Example: Skip even numbers**

```
for (let i = 1; i <= 10; i++) {
  if (i % 2 === 0) {
    continue; // Skip even numbers
  }
  console.log("Odd number:", i);
}
```

```
const scores = [45, -1, 67, 80, -5, 90];

for (let score of scores) {
  if (score < 0) {
    continue; // Ignore invalid data
  }
  console.log("Valid score:", score);
}
```

- **break** → exit loop immediately
- **continue** → skip current iteration
- Useful for **searching, filtering, validation**
- Be careful to avoid infinite loops, especially with continue in while loops

Searching in an Array

✓ Using break

```
const names = ["John", "Sita", "Aman", "Priya", "Kiran"];

for (let name of names) {
  if (name === "Priya") {
    console.log("Found Priya!");
    break;
  }
}
```

Search operations **should stop early** once the item is found → improves efficiency.

Skipping Bad Data

```
const scores = [45, -1, 67, 80, -5, 90];

for (let score of scores) {
  if (score < 0) {
    continue; // Ignore invalid data
  }
  console.log("Valid score:", score);
}
```

data cleaning → skip invalid values without stopping the full process

Infinite Loop with Continue in While

```
let i = 0;
while (i < 5) {
  if (i === 2) continue; // i never increments → infinite loop
  i++;
}
```

Always ensure your loop **still progresses** even when continue runs.

```
let i = 0;
while (i < 5) {
  i++;
  if (i === 2) continue;
}
```

Next chapter: Loop Best Practices

Loop Best Practices in JavaScript:

Choose the Right Loop for the Job

Best Practice

Use the loop that best matches the data structure and purpose.

✓ Guidelines

Task	Recommended Loop	Why
Iterate fixed number of times	for	Clear start/end/step
Iterate over array values	for...of	Clean and intuitive
Iterate over object keys	for...in	Reads object properties
Iterate until a condition	while / do-while	Useful for unknown iteration counts
Transform/filter arrays	.map(), .filter(), .reduce()	More readable & functional style

Example:

```
// Good: using for...of for arrays
const scores = [90, 85, 70];
for (const score of scores) {
  console.log(score);
}

// Good: using for...in for objects
const user = { name: "John", age: 25 };
for (const key in user) {
  console.log(key, user[key]);
}
```

Choosing the correct loop reduces complexity and makes code self-explanatory.

Avoid Infinite Loops

Best Practice

Always ensure the loop's **condition changes** and will eventually become false.

```
let i = 0;
while (i < 5) {
  console.log(i);
  // Forgot i++
}
```

bugs with loops come from missing updates to the loop variable.

```
let i = 0;
while (i < 5) {
  console.log(i);
  i++; // condition changes
}
```

Minimize Work Inside the Loop

Loops run many times — keep them lean.

✓ Do this:

- Cache array lengths
- Avoid unnecessary calculations
- Avoid repeated DOM access
- Avoid complex function calls inside the loop

Example: Cache length

```
const arr = [1,2,3,4,5];
for (let i = 0, len = arr.length; i < len; i++) {
  console.log(arr[i]);
}
```

Avoid Deeply Nested Loops

Deep nesting increases complexity.

Pyramid of Doom

```
for (let i = 0; i < a.length; i++) {
  for (let j = 0; j < b.length; j++) {
    for (let k = 0; k < c.length; k++) {
      console.log(a[i], b[j], c[k]);
    }
  }
}
```

When you see a loop inside a loop inside a loop — it's time to refactor.

Better: Use helper functions:

```
function processCombination(a, b, c) {
  for (const x of a)
    for (const y of b)
      for (const z of c)
        console.log(x, y, z);
}
```

Prefer Functional Array Methods When Appropriate

Functional methods are often **clearer** and **less error-prone**. Readable code helps future you — and your team.

Example

```
const nums = [1,2,3];

nums.forEach(n => console.log(n));        // simple iteration
const doubled = nums.map(n => n * 2);     // transformation
const evens = nums.filter(n => n % 2 === 0); // filtering
```

Use Meaningful Loop Variables

```
for (let x = 0; x < arr.length; x++) { ... }
```

```
for (let index = 0; index < arr.length; index++) { ... }
```

Avoid Mutating Arrays While Looping Over Them

Modifying an array while iterating can cause unexpected index shifts.

```
const arr = [1,2,3,4];
for (let i = 0; i < arr.length; i++) {
  arr.splice(i, 1); // modifies array structure
}
```

```
for (const item of [...arr]) {
  // safe operations
}
```

Loop Best Practices Checklist

✓ Choose the right loop type
✓ Avoid infinite loops
✓ Keep loop bodies minimal
✓ Use break/continue sparingly
✓ Avoid deep nesting
✓ Prefer functional array methods
✓ Use meaningful variable names
✓ Do not mutate arrays while iterating
✓ Make exit conditions predictable

Next chapter: Common Looping Patterns

Common Looping Patterns:

Loops are not just repetition tools — they help us **process collections, transform data, search values, and build efficient logic**.
These patterns appear in **real-world JS tasks**, especially when working with arrays, objects, or API data.

Counting Loop (Index Loop)

✓ When to use

- Running a loop **specific number of times**
- Needing access to the **index**

Example: This is the most basic loop — we know how many iterations we need, so we control the counter manually.

```
for (let i = 0; i < 10; i++) {
  console.log("Step:", i);
}
```

Iterating Over Arrays

A. For Loop (classic)

Gives full control: index, step size, direction.

```
const nums = [10, 20, 30];
for (let i = 0; i < nums.length; i++) {
  console.log(nums[i]);
}
```

B. for…of Loop (simple)

Best for reading values only.

```
for (const value of nums) {
  console.log(value);
}
```

For arrays, for…of is clean and avoids index mistakes.

Iterating Over Objects

When to use:

To access **keys**, **values**, or **entries** of objects.

```
const user = { name: "Srikanth", age: 25, role: "Developer" };

for (const key in user) {
  console.log(key, "→", user[key]);
}
```

Better alternatives (modern JS):

```
Object.entries(user).forEach(([key, value]) => {
  console.log(key, value);
});
```

Search Pattern (Finding a Value)

When to use

Finding **first matching item** in an array.

Example:

```
const nums = [3, 8, 12, 5, 9];
let found = null;

for (const n of nums) {
  if (n > 10) {
    found = n;
    break;
  }
}
console.log("First number > 10:", found);
```

Modern method:

```
nums.find(n => n > 10);
```

Filter Pattern (Selecting Items)

When to use

Build a new array based on a condition.

```
const nums = [1, 5, 8, 12, 3];
const filtered = [];

for (const n of nums) {
  if (n > 5) filtered.push(n);
}

console.log(filtered); // [8, 12]
```

Modern method:

```
nums.filter(n => n > 5);
```

Aggregation Pattern (Summing / Combining)

When to use

Compute totals, averages, counts.

```
const prices = [100, 200, 50];
let total = 0;

for (const p of prices) {
  total += p;
}

console.log("Total:", total);
```

Equivalent using reduce:

```
prices.reduce((sum, p) => sum + p, 0);
```

Transformation Pattern (Mapping)

✓ When to use

Convert one array into another.

```
const nums = [1, 2, 3];
const doubled = [];

for (const n of nums) {
  doubled.push(n * 2);
}

console.log(doubled); // [2, 4, 6]
```

Modern method:

```
nums.map(n => n * 2);
```