Conditional Logic in TypeScript

1. Problem Statement

A school portal needs a **Student Performance Evaluator** module. It must:

- Check exam **eligibility** based on attendance.
- Determine **pass/fail** status by score.
- Assign **letter grades** (A–F) using score ranges.
- Provide **feedback messages** for each grade.

Manual spreadsheet rules have become error-prone and hard to update. You need clear, maintainable code that handles each decision point correctly.

2. Learning Objectives

By the end of this lesson, you will be able to:

- Write `if` statements to run code when a condition is true.
- Use `if…else` for two-way branching.
- Chain `else if` for multi-way decisions.
- Implement `switch` statements for discrete value handling.
- Organize decision logic into reusable functions.

---

3. Concept Introduction with Analogy

Analogy: The School Principal's Decision Book
Just as a principal uses a well-organized "Decision Book" of rules to manage students, your TypeScript code uses conditional statements to make decisions in a clear, reliable way. Let's unpack each rule in the book and see how it maps to TypeScript constructs:

1. **Eligibility Rule → `if` Statement**
    - Book Entry:

      > "If a student's attendance is 75% or higher, they may sit the exam."

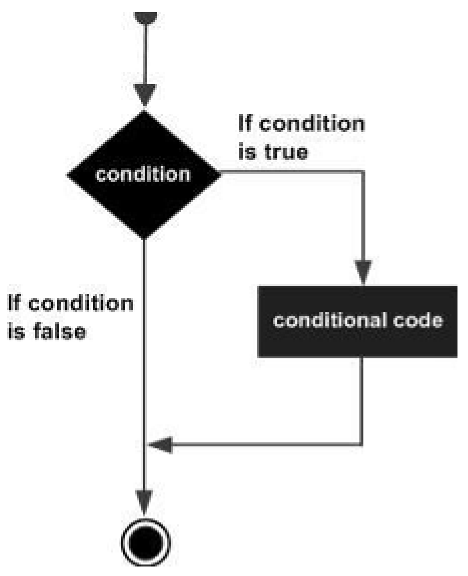
    - Code Equivalent:

      ```
      if (attendance >= 75) {
          // allow exam
      }
      ```

    - Explanation: The principal flips to the "attendance" page, checks the percentage, and if the condition is met, allows the student in. In code, the `if` evaluates a single boolean expression and runs its block only when true.

    ## Flowchart

The following flow chart shows how the if statement works.



2. **Pass/Fail Rule → `if...else` Statement**

   - Book Entry:

     > "If the student's score is 40 or above, mark 'Pass'; otherwise, mark 'Fail.'"

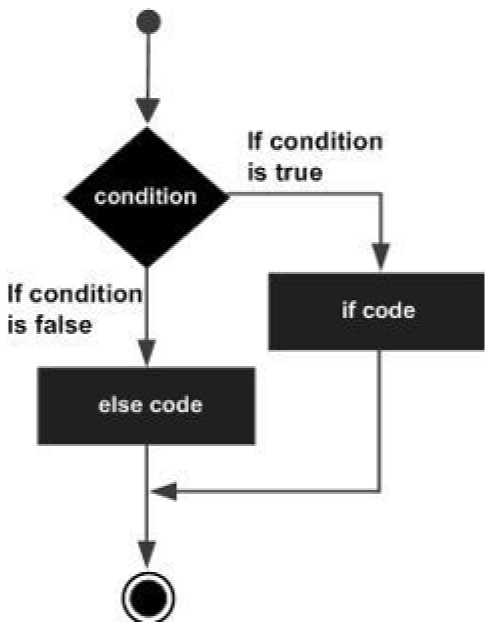
   - Code Equivalent:

```
if (score >= 40) {
 // Pass logic
} else {
 // Fail logic
}
```

   - Explanation: The principal reads the score, and if it meets the threshold, writes "Pass." Otherwise, they write "Fail." The `if…else` gives exactly two paths-one when true, one when false-matching the principal's binary decision.

Flowchart

The following flow chart shows how the if...else statement works.



3. **Grading Rule → `if…else if…else` Ladder**

   - Book Entry:

     > "90–100 → A; 80–89 → B; 70–79 → C; 60–69 → D; below 60 → F."

   - Code Equivalent:

```
if (score >= 90) {
  grade = "A";
} else if (score >= 80) {
  grade = "B";
} else if (score >= 70) {
  grade = "C";
} else if (score >= 60) {
  grade = "D";
} else {
  grade = "F";
}
```

- Explanation: The principal works down the list of ranges, stopping as soon as a match is found. The `else if` ladder mirrors this sequential evaluation-each condition is tested in turn until one is true.

4. **Feedback Rule → `switch` Statement**

- Book Entry:

> "For grade A, comment 'Excellent'; for B, 'Good job'; for C, 'Keep improving'; etc."

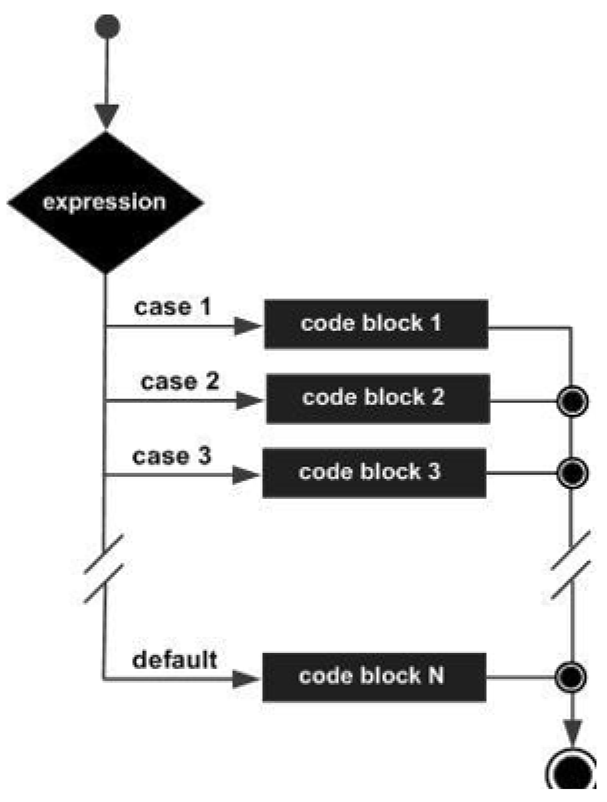
- Code Equivalent:

```
switch (grade) {
  case "A":
    // Excellent
    break;
  case "B":
    // Good job
    break;
  // … and so on
  default:
    // Fallback comment
}
```

- Explanation: The principal turns directly to the grade's section in the Book and reads the prepared comment. A `switch` lets code jump straight to the matching `case` block for a discrete set of values, then "break" to stop.

Flowchart

The following flow chart explains how a switch-case statement works.



Why This Analogy Works

- **Clarity & Organization**: Just as the principal's Book keeps rules neatly organized, conditional statements structure your logic in clear, maintainable blocks.

- **Step-by-Step Evaluation**: The principal follows each rule in a predictable order; your code does the same by evaluating conditions linearly.
- **Single Source of Truth**: The Book holds definitive policies; your code holds business rules in one place, minimizing ambiguity.
- **Easy Updates**: If the principal changes a policy, they edit the Book. If requirements change, you update or extend your `if` / `switch` logic-keeping code and intent in sync.

---

4. Technical Deep Dive

4.1 `if` Statement

Syntax:

```
if (condition) {
  // executes when condition is true
}
```

Flow: Evaluate `condition`; if true, run block; otherwise skip it.

4.2 `if…else` Statement

Syntax:

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

Flow: One of two paths.

4.3 Nested `if…else if…else`

Syntax:

```
if (cond1) {
  // block1
} else if (cond2) {
  // block2
} else {
  // block3
}
```

Flow: Checks in order, runs first matching block.

4.4 `switch` Statement

Syntax:

```
switch (value) {
  case const1:
    // block1
    break;
  case const2:
    // block2
    break;
  default:
    // fallback
    break;
}
```

Flow: Matches `value` to a `case`; runs that block; `break` prevents fall-through.

---

5. Step-by-Step Code Walkthrough

Below is a complete implementation of our four rules:

```
// 1. Check eligibility: attendance ≥ 75%
function checkEligibility(attendance: number): boolean {
  if (attendance >= 75) {
    console.log("Eligible: attendance is sufficient.");
    return true;
  } else {
    console.log("Not eligible: attendance below 75%.");
    return false;
  }
}

// 2. Determine pass/fail: score ≥ 40
function passOrFail(score: number): boolean {
  if (score >= 40) {
    console.log("Result: Pass");
    return true;
  } else {
    console.log("Result: Fail");
    return false;
  }
}

// 3. Assign letter grade
function assignGrade(score: number): string {
  if (score >= 90) {
    return "A";
  } else if (score >= 80) {
    return "B";
  } else if (score >= 70) {
    return "C";
  } else if (score >= 60) {
    return "D";
  } else {
    return "F";
  }
}

// 4. Provide feedback via switch
function provideFeedback(grade: string): void {
  switch (grade) {
    case "A":
      console.log("Feedback: Excellent performance!");
      break;
    case "B":
      console.log("Feedback: Great job! Keep it up.");
      break;
    case "C":
      console.log("Feedback: Good effort; aim higher next time.");
      break;
    case "D":
      console.log("Feedback: Needs improvement; review your work.");
      break;
    default:
      console.log("Feedback: Unsatisfactory; please seek help.");
      break;
  }
}

// Main evaluator combining all steps
function evaluateStudent(attendance: number, score: number): void {
  if (!checkEligibility(attendance)) return;
  if (!passOrFail(score)) return;
  const grade = assignGrade(score);
  console.log(`Assigned Grade: ${grade}`);
  provideFeedback(grade);
}

// Example Run
evaluateStudent(80, 85);
```

---

6. Interactive Challenge / Mini-Project

Implement four small functions to practice each decision-making construct:

1. **`checkSign(num: number): void`**
   Use an **`if`** statement to log whether `num` is positive.

2. **`evenOrOdd(num: number): void`**
   Use an **`if…else`** to log whether `num` is even or odd.

3. **`getGrade(score: number): string`**
   Use an **`if…else if…else`** ladder to return a letter grade:

   - `score ≥90 → "A"`
   - `score ≥80 → "B"`
   - `score ≥70 → "C"`
   - `score ≥60 → "D"`
   - otherwise `"F"`

4. **`provideFeedback(grade: string): void`**
   Use a **`switch`** to log a feedback message for each grade (`"A"`… `"F"`), with a `default` for any unexpected value.

---

7. Common Pitfalls & Best Practices

- Always include **braces `{}`** even for single statements: avoids errors when adding lines later.
- Use **strict equality `===`** for comparisons to avoid type coercion bugs.
- In `switch`, always add a **`default`** case and **`break`** after each `case`.
- Order `else if` from **most to least restrictive** to ensure correct branch selection.
- Keep each decision block **focused** on a single rule for readability.

8. Quick Recap & Key Takeaways

- `if` for single checks.
- `if…else` for two-way branching.
- `else if` for multiple conditions.
- `switch` for selecting among discrete values.
- Organize decision logic into small, reusable functions for clarity and maintainability.