Lesson:

Condition (If, If-else, and if else ladder & switch)

PW SKILLS

Topics Covered

1. Introduction to Conditional Statements
2. Conditional Statements in JavaScript
   a. if statement
   b. if-else statement
   c. If else ladder
3. Introduction to switch case.
4. Switch Syntax.
5. Switch with a return statement
6. What happens when we skip the break keyword?
7. Handling multiple cases having the same logiç
8. When to use switch statements or if/else statements?
9. Example: Weekday
10. When to use switch statements or if/else statements.

Introduction to Conditional Statements

Programming Languages are tools that allow us to write code that instructs the computer to do something. In every programming language, the code needs to make decisions and carry out actions accordingly depending on different inputs.

Human Beings make decisions all the time. For example, every morning, we make a decision between eating or not eating before starting our daily chores. Conditional statements allow us to represent such decision-making in JavaScript, from the choice that must be made.

JavaScript is a programming language that is commonly used to create interactive and dynamic elements on websites. One of the key features of JavaScript is the ability to use conditional statements to control the flow of a program.

Conditions work on boolean values, **true or false**. It is true if it meets the requirement, false otherwise. That is expressions (conditions) are evaluated to be either true or false.

Conditional Statements in JavaScript

There are three ways of writing conditionals in Javascript:

- If/else Statement
- Switch Statement
- Ternary Operator

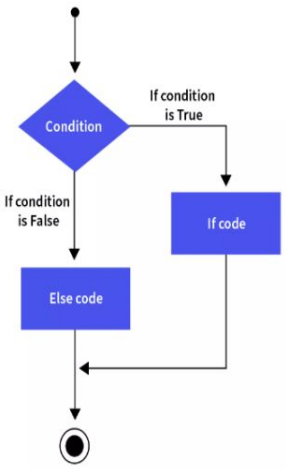
In this lecture let's look at if/else conditional and later on will look at other conditional statements.

1. if statement

The most basic form of a conditional statement is the **if statement**. The syntax for an **if statement** is as follows:

```
if (condition) {
   // statements
}
```

The condition is any expression that can be evaluated as true or false. For example, you can use a comparison operator (such as <, >, ==) to compare two values, or you can use a logical operator (such as &&, ||) to combine multiple conditions.



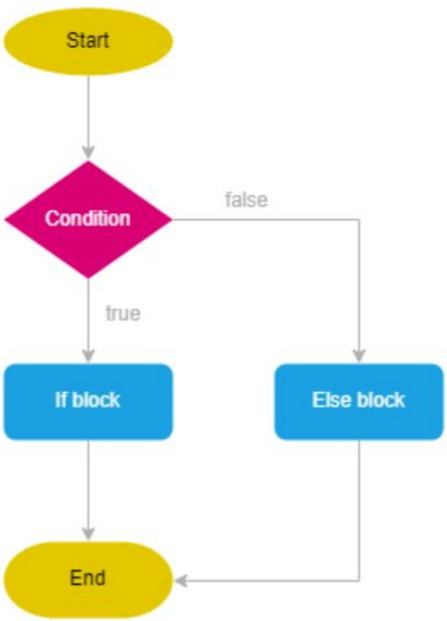
Example: Divide only, when the divisor is != 0

```
let dividend = 10;
let divisor = 5;

if(divisor =≠ 0){
  let result = dividend/divisor;
   console.log(result)
}
// Output
// 2
```

2. if-else statement

An **if statement** can also include an optional else statement, which will execute if the condition is false. The syntax for an **if-else statement** is as follows:




```
if (condition) {
   //if statements
} else {
   //else statements
}
```

Example 1: Only age above 18 are eligible for registration.

```
age = 15;.

if(age < 18){
   console.log("Sorry, you are not eligible")
}else{
  console.log("Registered Successfully")
}

// Output
// Sorry, you are not eligible
```

Example 2: Allow only admin to fetch user details

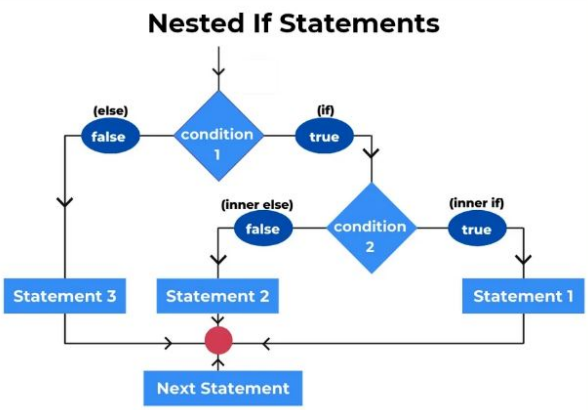
```
isAdmin = false;

if(isAdmin){
   console.log("fetch user details")
}else{
  console.log("This operation is only for Admins")
}

// Output
// This operation is only for Admins
```

3. Nested if else

We can nest if else statements inside another if else statements.



```
if (condition1) {
  if(condition2){
    // statement1
  }else{
    // statement2
  }
} else {
  if(condition3){
     // statement3
  }else{
     // statement4
  }
}
```

Example 2: Find the greatest number among 3 numbers.

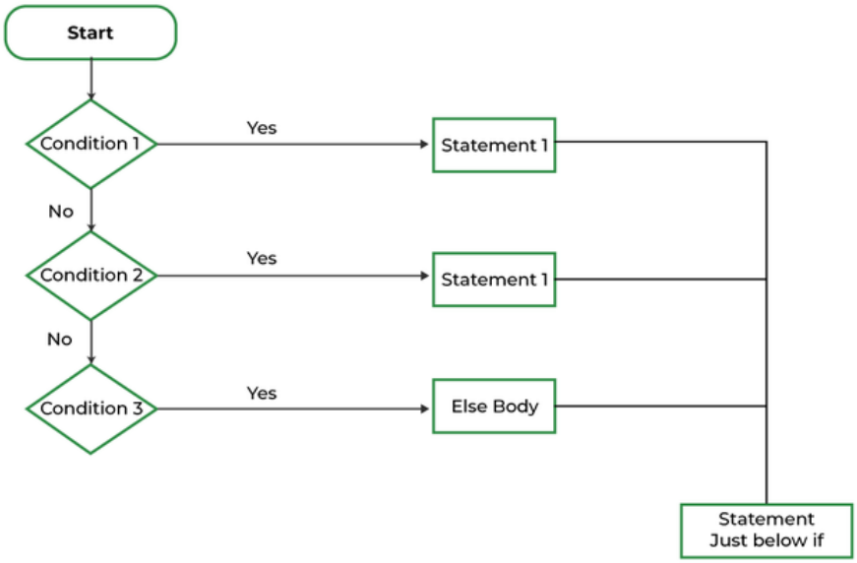
```
let a = 10;
let b = 14;
let c = 54;

if(a > b){
  if(a > c){
 console.log("a is greatest");
  }else{
     console.log("c is greatest");
  }
}else{
  if(b > c){
 console.log("b is greatest");
  }else{
     console.log("c is greatest");
  }
}

// Output - c is greatest
```

3. If else ladder

JavaScript also supports the use of **else if** statement, which allows you to chain multiple conditions together also called **if else ladde**r. The syntax for an **if-else ladder** is as follows:




```
if (condition1) {
  //statement1
} else if (condition2) {
  //statement2
} else {
  //statement3
}
```

Note 1: You can chain as many **else if** statements as you want.
Note 2: In the if-else ladder we can omit the last **else** block, it is not mandatory to put.

Example: The challenge here is to find out if the given number is even or odd.

Before solving any problem through programming it is important to first analyze what is the input that will be taken, what the conditionals involved, and what the output expected.

In this case, we will be taking integers as input. The output is expected to be a message telling if the number is even or odd.

The conditions to be considered to solve this problem are

1. Any number that is completely divisible [remainder must be 0] by 2 then it is an even number.
2. Any number that is not completely divisible [remainder must be 0] by 2 then it is an odd number.
3. Zero is neither an odd number nor an even number.

Before handling each condition, let declare our input variable,

```
// Input
var num;
```

In the above block of code, we have declared a variable named **num**. The variable num will be our input.

Implement 1st condition

Now let's handle the **1st condition:** that is if any number that is completely divisible [remainder must be 0] by 2 then it is an even number.

To check if the number is completely divisible by 2 we will be making use of the modulo operator (%) which returns the remainder. If the result of the modulo operation is 0 then the number is even.

```
// Handling 1st Condition

var num = 10;

// Condition 01: Any number that is completely
divisible [ remainder must be 0 ] by 2 then it
is an even number.
if (num % 2 == 0) {
  console.log("The number given is an even
number");
}
```

If the condition is false, the code inside the if block will not be executed, and this code will not give any output.

Implement 2nd Condition:

Since we are not getting any output if the condition is false. It's time to handle the 2nd condition which is any number that is not completely divisible [remainder must be 0] by 2 then it is an odd number.

As we have already checked for the even number condition, now **if** the condition for even fails it is an odd number. We can check this through the **else** statement.

```
// Condition 01: Any number that is completely
divisible [ remainder must be 0 ] by 2 then it
is an even number
if (num % 2 == 0) {
  console.log("The number given is an even
number");
}
// Condition 02: Any number that is not
completely divisible [ remainder must be 0 ] by
2 then it is an odd number.
else {
  console.log("The number given is an odd
number");
}
```

Now the code is capable of checking if the given number is odd or even.

Implementing 3rd Condition:

We also have our third condition which is that zero is neither an odd number nor an even number. So the first condition we need to check is if the number is zero, then if the number is even, and at last, if both the conditions fail it is an odd.

Final Solution

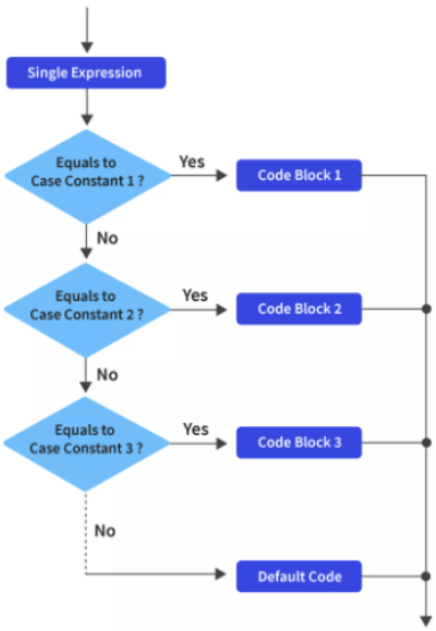
```
// Condition 03: Zero is neither an odd number
nor an even number.
if (num == 0) {
  console.log("Zero is neither an odd number nor
an even number");
}
// Condition 01: Any number that is completely
divisible [ remainder must be 0 ] by 2 then it
is an even number.
else if (num % 2 == 0) {

  console.log("The number given is an even
number");
}
// Condition 02: Any number that is not
completely divisible [ remainder must be 0 ] by
2 then it is an odd number.
else {
  console.log("The number given is an odd
number");
}
```

Switch Case

Introduction to switch case.

Since we have been talking about conditionals from past lectures. Let's look at a real-life scenario, assume it's lunchtime and you walk to your favorite restaurant. The attendant offers you the menu. On the menu are different delicious items made for special people like you. You go through the menu and choose one or more meals from the menu and have yourself a good lunch. That is what **switch** statements help us do in JavaScript.



Switch Syntax.

Let's look at the syntax of the switch statement.

```
switch (expression) {
  case value1:
    // code to execute;
    break;
  case value2:
    // code to execute;
    break;
  case value3:
    // code to execute;
    break;
  default:
  // default code;
}
```

switch keyword defines a switch block.

case keyword defines different case blocks, which will be executed only when case value matches with expression value. Case values can be a number, string or boolean etc.

break keyword is used at the end of every case block to terminate switch case evaluation further.

default keyword defines a default case block. When no case value is matched then the default block will be executed. It is not mandatory to include it.

Let's compare the switch syntax with the if statement

```
switch (expression) {
   case value1: // if (expression === value1 then
execute this block)
   // code to execute;
   break;

   case value2: // if (expression === value2 then
execute this block)
   // code to execute;
   break;

  case value3: // if (expression === value3 then execute
this block)
  // code to execute;
  break;

  default: // if (expression === none of the previous
values matched execute this block)
 // default code;
}
```

Example: Weekday

Let's now implement a switch statement considering an example.

We represent our weekdays in both number notation and text notation like 1 for Sunday, 2 for Monday, and so on.

Let's take the number notation as input and provide text notation as output using the switch statement.

In this case, the only condition is that the number notation of the day must be between 1 to 7. If any other input is given then it will be considered invalid input.

Let's look at the code.

```
var day = 5;

switch (day) {
  case 1: // if (day === 1) then execute this block)
    console.log("Sunday");
    break;


  case 2: // if (day === 2) then execute this block)
    console.log("Monday");
    break;

  case 3: // if (day === 3) then execute this block)
    console.log("Tuesday");
    break;

  case 4: // if (day === 4) then execute this block)
    console.log("Wednesday");
    break;

  case 5: // if (day === 5) then execute this block)
    console.log("Thursday");
    break;

  case 6: // if (day === 6) then execute this block)
    console.log("Friday");
    break;

  case 7: // if (day === 7) then execute this block)
    console.log("Saturday");
    break;

  default: // if (expression === none of the previous
conditions then execute this block)
    console.log("Day doesn't exist");
}
// Output: Thursday
```

This code will output **Thursday** to the console because the value of the variable **day** is **5**.

As we have passed the day to the switch statement, the day matches case **5** in the switch statement. When this case is executed, it will run the code block associated with it, which is console.log **(Thursday)**.

The break statement at the end of the case ensures that the code execution exits the switch statement after the matching case has been executed, so the code in the other cases and the default block will not be executed.

In the above example, we have used numbers as case values, 1,2,3,4, but be careful if you use "1" as your case value it will be treated as a different case because "1" is a string and 1 is a number datatype and switch cases use strict comparison (===). The values must be of the same type to match. A strict comparison can only be true if the operands are of the same type.

```
console.log(1 === '1') // false
```

Switch with return statement

We can use the **return** keyword instead of **break** for every case. It will be helpful, if we are using switch statements inside the **function**, because it will serve two purposes, one to come out of the **switch** block (like a **break**) and at the same time it returns the result from the function.

Lets see one example using return statement,

```
let grade = 'B';

function getValue(grade){
   switch (grade) {
    case 'A':
       return "Excellent";
    case 'B':
       return "Average"
    case 'C':
       return "Below than average";
    default:
       return "No Grade";
   }
}

console.log(getValue(grade)); // Average
```

In the above example, the **getValue** function returns keywords "Excellent", "Average", "Below Average" based on grades A, B, C respectively. It uses switch statements and instead of the **break** keyword in every case we are returning value using the **return** keyword.

What happens when we skip the break keyword?

In JavaScript, when you skip the break keyword in a switch statement, the program will continue executing the statements in the following case(s) without any further evaluation of the case conditions. This behavior is known as "fall-through."

Example

```
let color = "red";

switch (color) {
  case "red":
    console.log("The color is red.");
  case "blue":
    console.log("The color is blue.");
    break;
  case "green":
    console.log("The color is green.");
     break;
  default:
    console.log("The color is unknown.");
}

//Output
/*
In this case, if the value of color is "red," but the
program will print:

The color is red.
The color is blue.
*/
```

Since the **break** keyword is not used after the first case, the program will continue executing the statements in the subsequent cases. This is because JavaScript does not automatically exit the switch statement after a matching case unless a **break** statement is encountered.

Handling multiple cases having same logic

Sometimes we have situations where, we have same logic for different cases, here is an example,

Suppose in an organization, salary is incremented annually based on their role,

- If the role is an employee, manager or HR increment will be 5%.
- If the role is CEO, CIO, CTO increment will be 10%.

Lets see how we can implement this using switch,

```
let role = "CEO";
let salary = 100000;

switch (role) {
  case "employee":
  case "hr":

  case "manager":
   salary += 0.05*salary;
     break;
  case "CEO":
  case "CIO":
  case "CTO":
     salary += 0.1*salary;
     break;
  default:
    console.log("Unknown Role");
}

console.log(salary);
//Output - 110000
```

When to use switch statements or if/else statements?

- **if/else** conditional branches are great for variable conditions that result in a Boolean, whereas switch statements are great for fixed data values.
- In a situation where more than one choice is preferred, the **switch** is a better choice than an **if/else** statement.
- Considering the speed of execution it is advised if the number of cases is more than 5 use a **switch**, otherwise, you may use **if-else** statements.
- **Switch** is more readable and looks much cleaner when you have to combine cases.

Interview Point.

1. **Explain a scenario where using a switch statement might lead to unexpected behavior.**

Ans: One scenario where using a switch statement might lead to unexpected behavior is when not using the break statement within each case.

```
let day = "Monday";
let message = "";

switch (day) {
 case "Monday":
    message += "It's Monday. ";
  case "Tuesday":
    message += "It's Tuesday. ";
  case "Wednesday":
    message += "It's Wednesday. ";
  case "Thursday":
    message += "It's Thursday. ";
  case "Friday":
    message += "It's Friday. ";
  case "Saturday":
    message += "It's Saturday. ";
  case "Sunday":
    message += "It's Sunday. ";
}
console.log(message);
```

In this example, there are no break statements after each case. As a result, if day is "Monday," the code will fall through all subsequent cases, and the final console.log will output:

```
It's Monday. It's Tuesday. It's Wednesday. It's Thursday.
It's Friday. It's Saturday. It's Sunday.
```

This behavior is often unexpected because developers might assume that once a matching case is found, the switch statement will exit. Without break statements, the control flow continues to the next case, executing all the subsequent code until the end of the switch statement.

Always use break statements appropriately to control the flow of a switch statement and prevent unintended fall-through behavior.

2. Discuss the potential performance issues associated with deeply nested if-else statements.

Ans: Deeply nested if-else statements can introduce several potential performance issues and impact the readability and maintainability of the code. Here are some considerations:

1. **Readability and Maintainability:**

- Code Complexity: Deeply nested structures make the code more complex and harder to read. It becomes challenging to understand the flow and logic, especially for someone new to the codebase.

- Maintenance Difficulty: Making changes or fixing bugs in a deeply nested structure can be error-prone and time-consuming. Each level of nesting adds to the cognitive load for developers.

2. **Performance Overhead:**

- Execution Time: Every additional level of nesting introduces a slight performance overhead. While modern JavaScript engines are optimized to handle such structures efficiently, the difference becomes more noticeable in large, complex applications.

- Code Parsing: Deeply nested structures may take slightly longer to parse during the compilation phase, affecting the initial load time of the application.

3. **Potential for Bugs:**

- Increased Probability of Errors: Deep nesting increases the likelihood of introducing logical errors, especially when dealing with complex conditions and multiple levels of indentation. It's easier to overlook a condition or make a mistake in deeply nested code.

- Difficulty in Debugging: Debugging becomes more challenging as developers need to trace through multiple levels of nesting to identify the source of an issue.

3. **Can you identify a potential issue when using non-primitive values (objects, arrays) in a switch statement? Provide an example.**

Ans: One potential issue when using non-primitive values (objects, arrays) in a switch statement is that JavaScript's switch statement performs strict equality (===) comparisons. This can lead to unexpected behavior when comparing object references.

```
let fruit = { name: 'Apple' };

switch (fruit) {
  case { name: 'Apple' }:
    console.log('It\'s an apple!');
    break;
  default:
    console.log('Unknown fruit.');
}
// Output: 'Unknown fruit.'
```

The output will be 'Unknown fruit.' rather than 'It's an apple!'. This is because the switch statement internally uses strict equality (===) for comparisons, and objects are compared by reference, not by their content. In this case, the object literal used in the case statement creates a new reference, which is not strictly equal to the fruit object reference.

To correctly handle non-primitive values in a switch statement, you might consider using a series of if-else statements or a different approach, like comparing a specific property of the object:

```
let fruit = { name: 'Apple' };
if (fruit.name === 'Apple') {
  console.log('It\'s an apple!');
} else {
  console.log('Unknown fruit.');
}

// Output: ‘It's an apple!’
```

This ensures that the comparison is based on the value of the name property rather than the object reference, avoiding the unexpected behavior introduced by the switch statement's strict equality comparison.