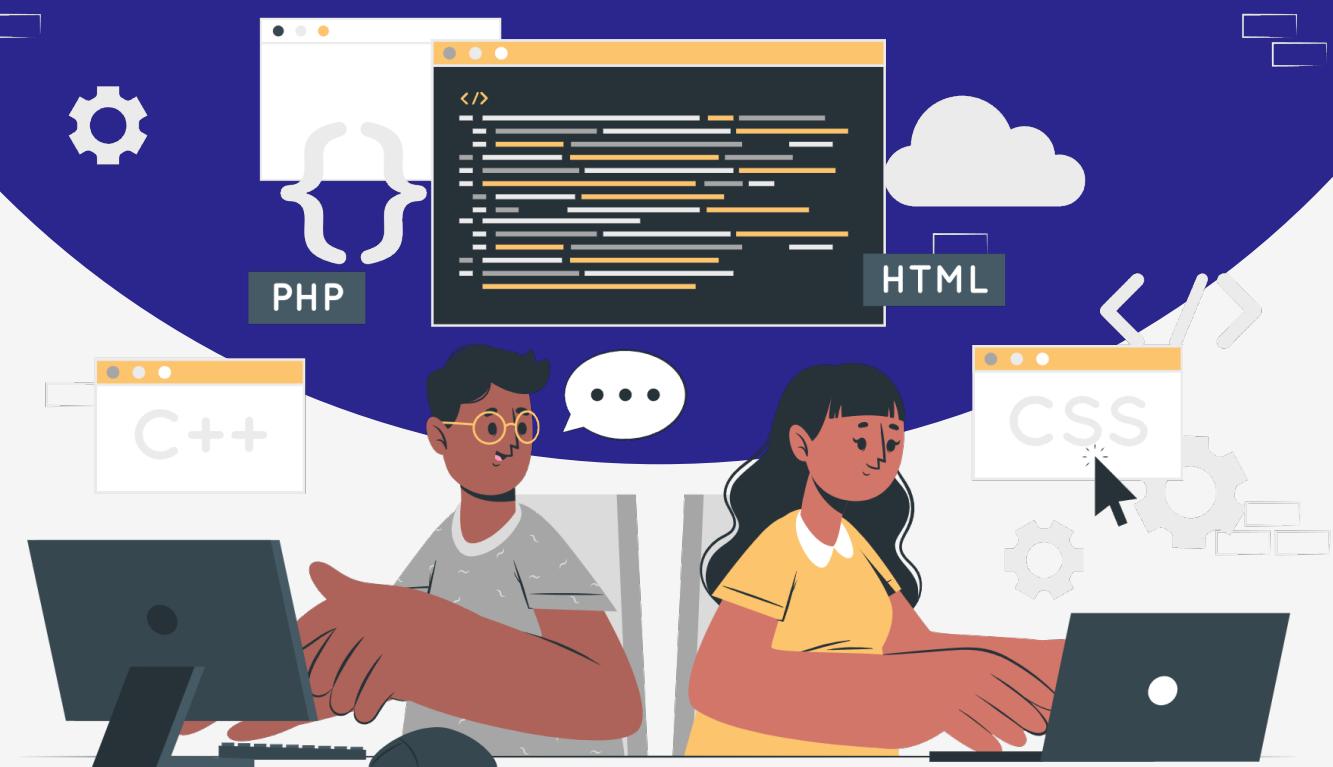


# Lesson:

## TypeOf and Ternary Conditions



# Topics Covered

1. Introduction to typeof Operator
2. typeOf Operator and type safety
3. Introduction to ternary operator
4. Chaining using the Ternary operator

## Introduction to typeof Operator

The **typeof** operator is a JavaScript operator that allows checking the data type of a given variable. It can be used with any data type, including objects, arrays, and even null values.

# typeof

JAVASCRIPT

How to check the  
Type in JavaScript

### Example

```
var name = "PW Skills";
var students = 12345678;
var enrolledToFSWD = true;
var couponCode = null;
var endDate;
var studentsEnrolled = NaN;
```

```

console.log(typeof name); // string
console.log(typeof students); // number
console.log(typeof enrolledToFSWD); // boolean
console.log(typeof couponCode); // object
console.log(typeof endDate); // undefined
console.log(typeof studentsEnrolled); // number

var mentorDetails = {
  name: "Anurag",
  yearsOfExperience: 4,
};

var techStack = ["HTML", "CSS", "Javascript", "Node",
"React", "Express"];

console.log(typeof mentorDetails); // object
console.log(typeof techStack); // object

```

There are a few benefits to using the **typeof** operator in JavaScript.

It is a convenient way to check if a variable is of a certain data type without having to check for conditions.

**Note:** typeof array is “object”, because array derived from object data type, and typeof returns all values other than primitive data type as “object”.

### **typeof Operator and type safety**

Type safety means to refrain from doing the wrong operation on the wrong datatype.



We can implement some level of type safety, using the type operator.

**Example:** Let's say we have a username variable called **name**.

```
let name = "Subham";
```

Later on, we transformed the name to uppercase.

```
name.toUpperCase() // "SUBHAM"
```

But suppose, if we mistakenly assigned a name as a number. It will throw an error, `Uncaught TypeError`

```
let name = 23;
name.toUpperCase() // VM40:2 Uncaught TypeError:
name.toUpperCase is not a function
```

So, to tackle this problem, we can use the `typeof` operator in if condition, before performing the `toUpperCase` operation to check whether the name is a string or not. We will study more about conditional statements in later lessons.

```
let name = 23;

if(typeof name === "string"){
  name.toUpperCase()
} else{
  console.log("Please provide a String")
}
// Output - Please provide a String
```

Type safety is a vast concept. We have a popular language called typescript (JavaScript with Types), which handles type safety very wisely.

# Introduction to Ternary Operator

So far we have seen operators are used to assign, compare, and evaluate one, two operands. Ternary operator is one of the special operators that have 3 operands and are often used as a shorthand for an if-else statement.

**The majority of developers use the ternary operator because:**

1. The ternary operator allows you to write simple if-else statements in a single line of code, **making your code more compact and readable**.
2. The ternary operator can make your code more expressive by allowing you to clearly express the intent of the code in a **single line**.
3. The ternary operator is simple to use and understand, making it a good choice for short and simple conditions.
4. The ternary operator is widely used in several libraries and frameworks such as React.js where it's **widely used**.

## Syntax of Ternary Operator:

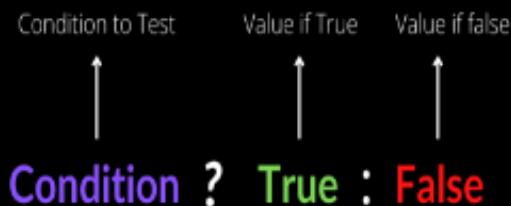
Let's look at the syntax compared with the if-else statement.

```
// if-else statement

if (condition) {
    expressionIfTrue;
} else {
    expressionIfFalse;
}
```

The if-else statement takes a condition that will be evaluated to be either true or false. The if-else statement above can be rewritten with ternary operators.

```
// Ternary Operator
condition ? <expressionIfTrue> : <expressionIfFalse>
```



## Example

Let's look at an example. Assume that you need to check if the person is logged in or not and provide the access to PW Skills lab.

```
var isTheUserLoggedIn = true;

// Using ternary operator

isTheUserLoggedIn
? console.log("PW Skills lab Access Granted !!")
: console.log("PW Skills lab Access Denied !!");

OR

console.log(isTheUserLoggedIn
? "PW Skills lab Access Granted !!"
: "PW Skills lab Access Denied !!");
```

## Chaining Ternary operator

Like we have seen nested if-else statements in last lesson, we can do the same with ternary operator by chaining. **Chaining** refers to the practice of using multiple ternary operators in succession to create more complex conditional expressions.

## Example

Let's assume in order to access the "Full Stack Web Developer Course", the user must be both logged in and should have purchased the course. Let's see how we can do this using the ternary operator.

Course purchase module: User can only purchase a course if he/she loggedIn.

**Case 1:** User not loggedIn

```
var isTheUserLoggedIn = false;

var isTheCoursePurchased = false;

isTheUserLoggedIn
? isTheCoursePurchased
? console.log("Access Granted")
: console.log("Access Denied!! Please Buy The
Course")
: console.log("Access Denied!! Please Login");
// OUTPUT : Access Denied!! Please Login
```

**Case 2:** User loggedIn, but not purchased course

```
var isTheUserLoggedIn = true;

var isTheCoursePurchased = false;

isTheUserLoggedIn
? isTheCoursePurchased
? console.log("Access Granted")
: console.log("Access Denied!! Please Buy The
Course")
: console.log("Access Denied!! Please Login");

// OUTPUT : Access Denied!! Please Buy The Course
```

**Case 3:** User not loggedIn and purchased course

```
var isTheUserLoggedIn = true;

var isTheCoursePurchased = true;

isTheUserLoggedIn
? isTheCoursePurchased
? console.log("Access Granted !!")
: console.log("Access Denied!! Please Buy The
Course")
: console.log("Access Denied!! Please Login");

// OUTPUT : Access Granted !!
```

Using chaining in ternary operators is not advised because chaining multiple ternary operators can make the code more difficult to read and understand, especially if the expressions become complex. This can make it harder for other developers (including yourself in the future) to maintain and debug the code.

## Interview Point.

### 1. What is the difference between the ternary operator and a regular if-else statement?

Ans: The ternary operator and a regular if-else statement in JavaScript both serve the purpose of handling conditional expressions, but they differ in syntax and use cases. The ternary operator, with its concise syntax (condition ? expressionIfTrue: expressionIfFalse;), is often favored for simple conditional assignments or expressions where the logic is straightforward. It is suitable for compact code snippets and can be used within larger expressions.

On the other hand, the if-else statement has a more extensive and flexible syntax (if (condition) { /\* code if true \*/ } else { /\* code if false \*/ }). It is preferred for more complex conditional logic or scenarios where multiple statements need to be executed based on the condition. If-else statements are generally more readable for intricate conditions or when several statements are involved, offering greater flexibility in code structure.

### 2. What are the potential pitfalls of nesting too many ternary operators within a single expression?

Ans: Nesting too many ternary operators within a single expression can introduce several potential pitfalls:

#### **Decreased Readability:**

As the number of nested ternary operators increases, the code becomes harder to read and understand. It may lead to confusion among developers, especially those who are not familiar with the codebase.

#### **Difficulty in Debugging:**

Debugging nested ternary operators can be challenging, as it requires tracking multiple conditions and expressions within a single line. Identifying the source of an issue or understanding the flow becomes more complicated.

#### **Lack of Code Separation:**

Nesting multiple ternary operators within a single line eliminates the clear separation of different conditions and expressions. Code separation is crucial for readability and maintainability, and excessive nesting undermines this principle.

### 3. Is it possible to use the ternary operator to check for multiple conditions at once? Why or why not?

Ans: Yes, it is possible to use the ternary operator to check for multiple conditions at once using logical operators. You can combine conditions with logical operators (such as `&&` for "and" and `||` for "or") to create more complex conditional expressions.

```
let result = (x > 0 && y > 0) ? 'Both are positive' :  
(x > 0 || y > 0) ? 'At least one is positive' :  
'Neither is positive';
```

However, it's crucial to be mindful of the complexity and readability of the code. If the logic becomes too intricate, using an if-else statement might be more appropriate for clarity and maintainability. Striking a balance between conciseness and readability is essential when working with multiple conditions using the ternary operator.