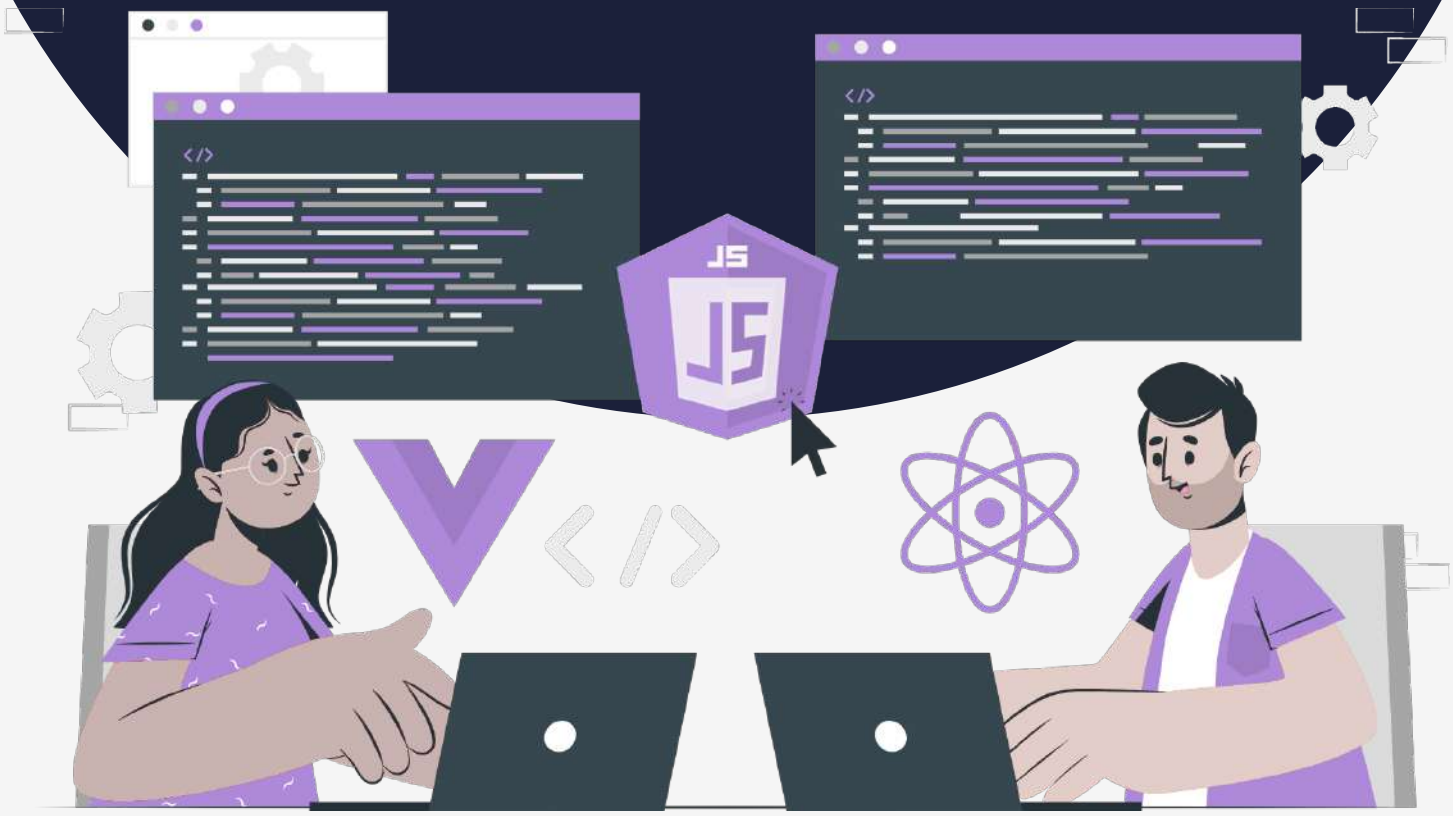
Lesson:

How to use JS in HTML File



Topics Covered :

1. Introduction to script in HTML
2. Ways to include javascript in HTML

In HTML, a script is a block of code that is used to add interactivity or other dynamic functionality to a webpage. The script is typically written in a programming language such as JavaScript, and it is executed by the browser when the webpage is loaded. As one of the core technologies of the web alongside HTML and CSS, JavaScript is used to make webpages interactive and to build web apps.

When working with files for the web, JavaScript needs to be loaded and run alongside HTML. This can be done either by writing javascript within an HTML document or in a separate file that the browser will load alongside the HTML document.

One can add javascript into HTML in the following ways:

1. Embedding the JavaScript code between a pair of <script> and </script> tags.
2. Creating an external JavaScript file with the .js extension and then loading it within the page through the src attribute of the <script> tag.
3. Placing the JavaScript code directly inside an HTML tag using special tag attributes such as onclick, onmouseover, onkeypress, onload, etc.

We will be looking at the first two methods in this lecture. The third method would be demonstrated in further lectures.

Embedding the JavaScript Code:
You can add JavaScript code in an HTML document by using the dedicated HTML tag <script> that wraps around JavaScript code.

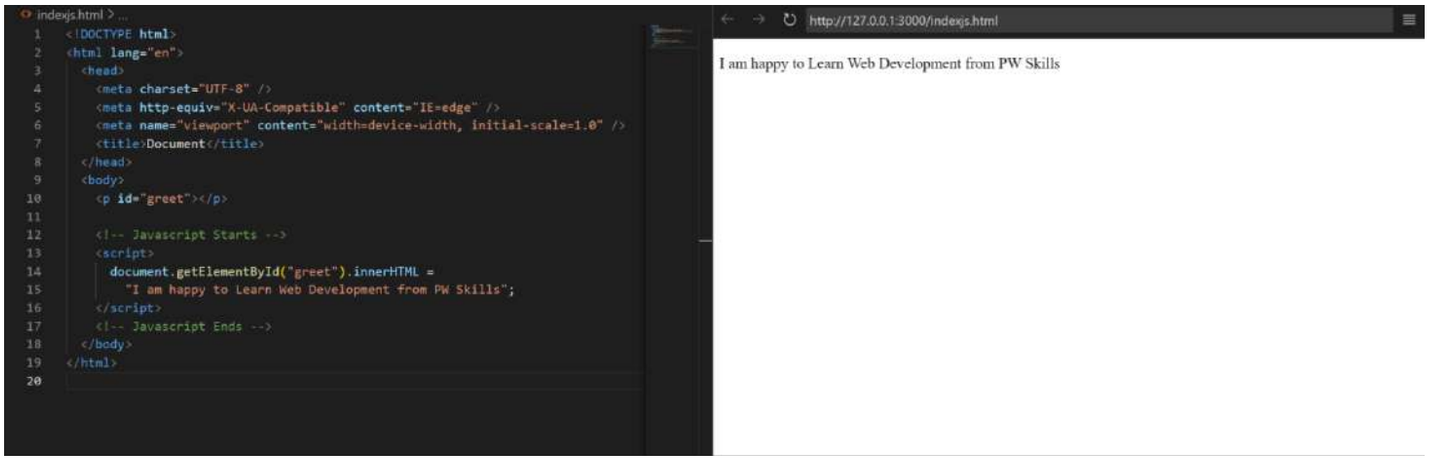
The <script> tag can be placed in the <head> section of your HTML or in the <body> section, depending on when you want the JavaScript to load.

Generally, JavaScript code can go inside the document <head> section in order to load the script before your HTML document.

However, if your script needs to run at a certain point within a page's layout you should put it at the point where it should be called, usually within the <body> section.

Usually, the script tag is included just before the body closing tag.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <p id="greet"></p>

    <!— Javascript Starts —>
    <script>      document.getElementById("greet").innerHTML =
        "I am happy to Learn Web Development from PW Skills";
    </script>    <!— Javascript Ends —>
  </body>
</html>
```



This is an HTML document that uses JavaScript to update the text of a paragraph element.

The script tag contains JavaScript code that is executed by the browser when the webpage is loaded. The script uses the document.getElementById method to access the paragraph element with the id "greet". The innerHTML property of this element is then set to a new string of text, "I am happy to Learn Web Development from PW Skills".

When the browser loads the page and runs the JavaScript, the text of the paragraph element is updated to the new string, and the user will see "I am happy to Learn Web Development from PW Skills" displayed on the page.

We will be looking at document, getElementById, and other commands in further lectures.

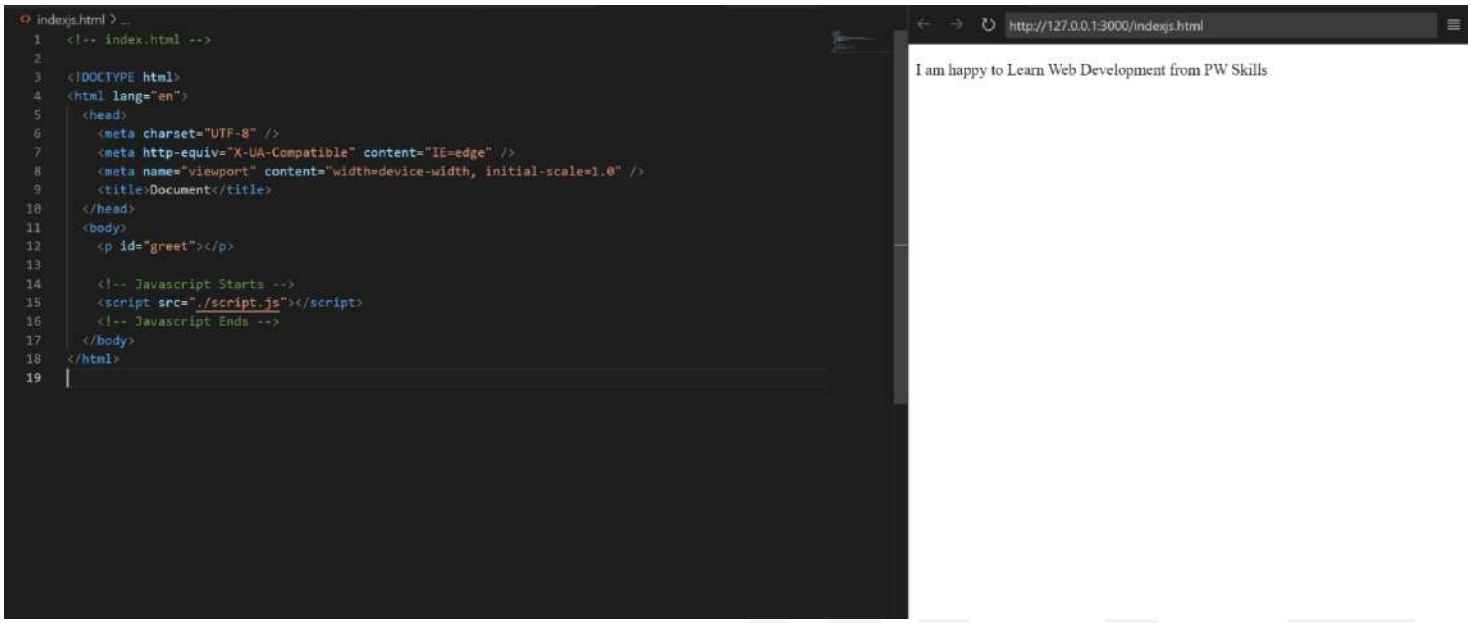
Working with a Separate JavaScript File
In order to accommodate larger scripts or scripts that will be reused across several pages, JavaScript code generally is written in one or more js files that are called within HTML documents, similar to how external files like CSS are called.

The benefits of using a separate JavaScript file include

1. Separating the HTML markup and JavaScript code isolates the script from the HTML document.
2. Separate files make maintenance easier
3. Usually, when an external JavaScript file is downloaded for the first time, it is stored in the browser's cache, so it won't need to be downloaded multiple times from the web server which makes the web pages load more quickly.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <p id="greet"></p>

    <!— Javascript Starts —>
    <script>      document.getElementById("greet").innerHTML =
        "I am happy to Learn Web Development from PW Skills";
    </script>    <!— Javascript Ends —>
  </body>
</html>
```

```
// script.js

document.getElementById("greet").innerHTML = "I am happy to Learn Web Development from
PW Skills";
```



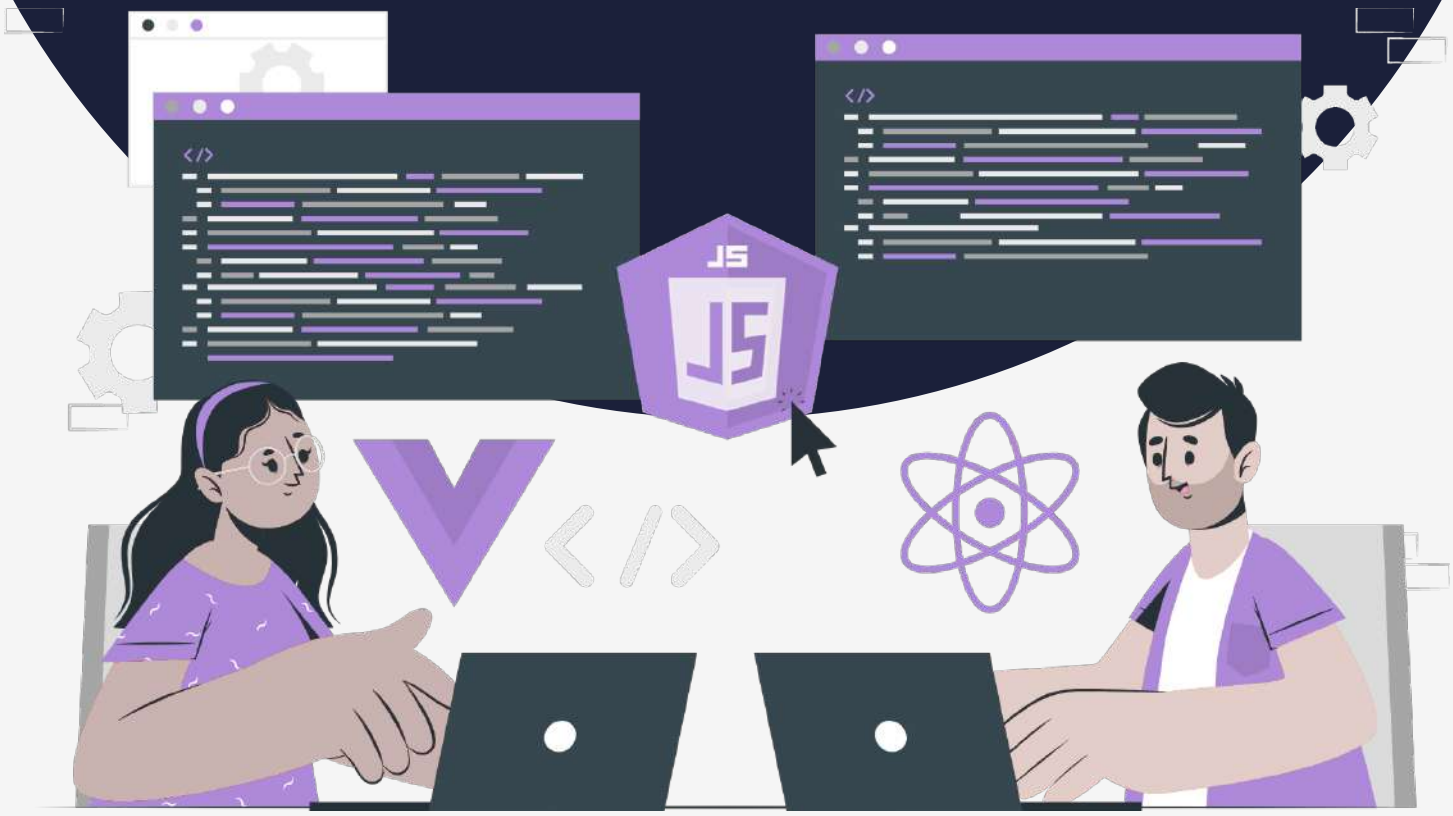
The JavaScript code is included in the HTML document through an external file called "script.js".

The script tag with the src attribute linking to the script.js file is included in the HTML document and it is executed by the browser when the webpage is loaded. The script.js file contains the JavaScript code that uses the document.getElementById method to access the paragraph element with the id "greet". The innerHTML property of this element is then set to a new string of text, "I am happy to Learn Web Development from PW Skills".

When the browser loads the page and runs the JavaScript code, the text of the paragraph element is updated to the new string, and the user will see "I am happy to Learn Web Development from PW Skills" displayed on the page.

Lesson:

Why and What is JavaScript



Topics Covered :

1. Introduction
2. Advantages of learning Javascript
3. Overview of JavaScript
4. History of JavaScript
5. What is ECMAScript?
6. Versions of ECMAScript
7. Overview os TypeScript

Why JavaScript
JavaScript is a popular programming language that is widely used to build web applications. It is a client-side scripting language, which means that it is run by your web browser rather than on a server. This makes it a good choice for building web applications that need to be fast and responsive, as the code is run locally on the user's device rather than having to be sent back and forth between a server and a client. JavaScript is also used to build mobile apps, create interactive documents, and build server-side applications with the help of runtime environments such as Node.js.

There are several reasons why JavaScript has become so popular.

1. **Ease of use:** JavaScript is a high-level language, which means that it is easy to learn and use. It is also a dynamically-typed language, which means that you don't have to specify the type of a variable when you declare it, making it easy to write code quickly.
2. **Cross-platform compatibility:** JavaScript is supported by all modern web browsers, so you can use it to build web applications that will run on any device with a web browser.
3. **A large developer community:** There is a large and active community of developers who use and contribute to JavaScript, which means that there are many resources available for learning the language and getting help when you need it.
4. **Powerful capabilities: J**avaScript has a lot of powerful features that allow you to build complex and interactive applications. For example, you can use JavaScript to manipulate the HTML and CSS of a web page, make asynchronous network requests, and work with multimedia and other types of data.
5. **Growing demand:** The demand for JavaScript developers is high and continues to grow, making it a good language to learn if you want to pursue a career in software development.

Overview of JavaScript.
JavaScript is a dynamic programming language that's used for web development, web applications, game development, and lots more. It allows you to implement dynamic features on web pages that cannot be done with only HTML and CSS.

It is a client-side scripting language, which means that it is executed by the web browser on the user's device rather than on a server, allowing for fast and responsive user interactions without the need for a page refresh. It is also an object-oriented language, which means it is based on the concept of objects and their methods, allowing for encapsulation and reusability of code.

JavaScript can be added to a web page by including it directly within the HTML file using a script tag, linking to a separate .js file, or through various frameworks and libraries such as jQuery, React, Angular, and VueJS.

JavaScript also has a large and active developer community, and it is a language in high demand for web development, with many resources available for learning the language and getting help when you need it. It is constantly evolving with new updates, enhancements, and new tools. With the rise of web development and the proliferation of web-enabled devices and technology, it has become a must-have skill for any web developer or anyone looking to work in the tech industry.

History of JavaScript.

JavaScript was created in **1995** by Brendan Eich, a programmer at Netscape Communications Corporation. It was originally called **Mocha**, then changed to **LiveScript**, and finally, it was given the name **JavaScript** to leverage the popularity of Java, which was a popular programming language at the time. JavaScript was first introduced in Netscape Navigator 2.0, a popular web browser of the time.

In 1996, JavaScript was submitted to the European Computer Manufacturers Association (ECMA) and it was standardized as ECMAScript. This standardized version of JavaScript is still used today and is supported by all modern web browsers like Google Chrome, Mozilla Firefox, Apple Safari, Microsoft Edge, etc.

In the early days of JavaScript, it was primarily used for simple things like form validation and simple mouse interactions, but as browsers became more powerful and web standards evolved, JavaScript became more widely used for building more complex web applications. With the introduction of popular libraries and frameworks like jQuery, AngularJS, React, and Vue.js, it has become easier to build complex and powerful web applications using JavaScript.

The emergence of Node.js in 2009 made it possible to run JavaScript on the server side and it become more popular as a full-stack language, allowing for code reuse and sharing between the client side and the server side.

What is ECMAScript?

ECMAScript (often referred to as simply "JavaScript") is a programming language specification standardized by the European Computer Manufacturers Association (ECMA). It was first published in 1997 and is used to create scripts for the web and other environments. JavaScript is the most widely-used implementation of ECMAScript and is supported by all major web browsers.

ECMAScript defines the syntax, semantics, and features of the language, and specifies how it should be implemented. The specification includes things like:

- **Data types:** ECMAScript supports several data types such as numbers, strings, and objects.
- **Variables:** ECMAScript defines the way variables are declared and used.
- **Expressions and operators:** ECMAScript specifies the operators that can be used to manipulate data and create expressions.
- **Control flow:** ECMAScript defines the various control flow statements such as if-else, for, and while loops.
- **Functions:** ECMAScript defines how functions are created, called, and returned.
- **Objects:** ECMAScript specifies how to create and manipulate objects, including how to define properties and methods.

Versions of ECMAScript

ECMAScript has several versions, with the latest one being ECMAScript 2022 (ES13). Each new version of ECMAScript adds new features and updates the existing ones. The first version was introduced in 1997 as ECMAScript 1. In which they introduced basic JavaScript syntax and features. Major updates in JavaScript were introduced in ECMAScript 6 (2015) and they make JavaScript a more powerful, expressive, and efficient language, allowing developers to write more maintainable and scalable code.

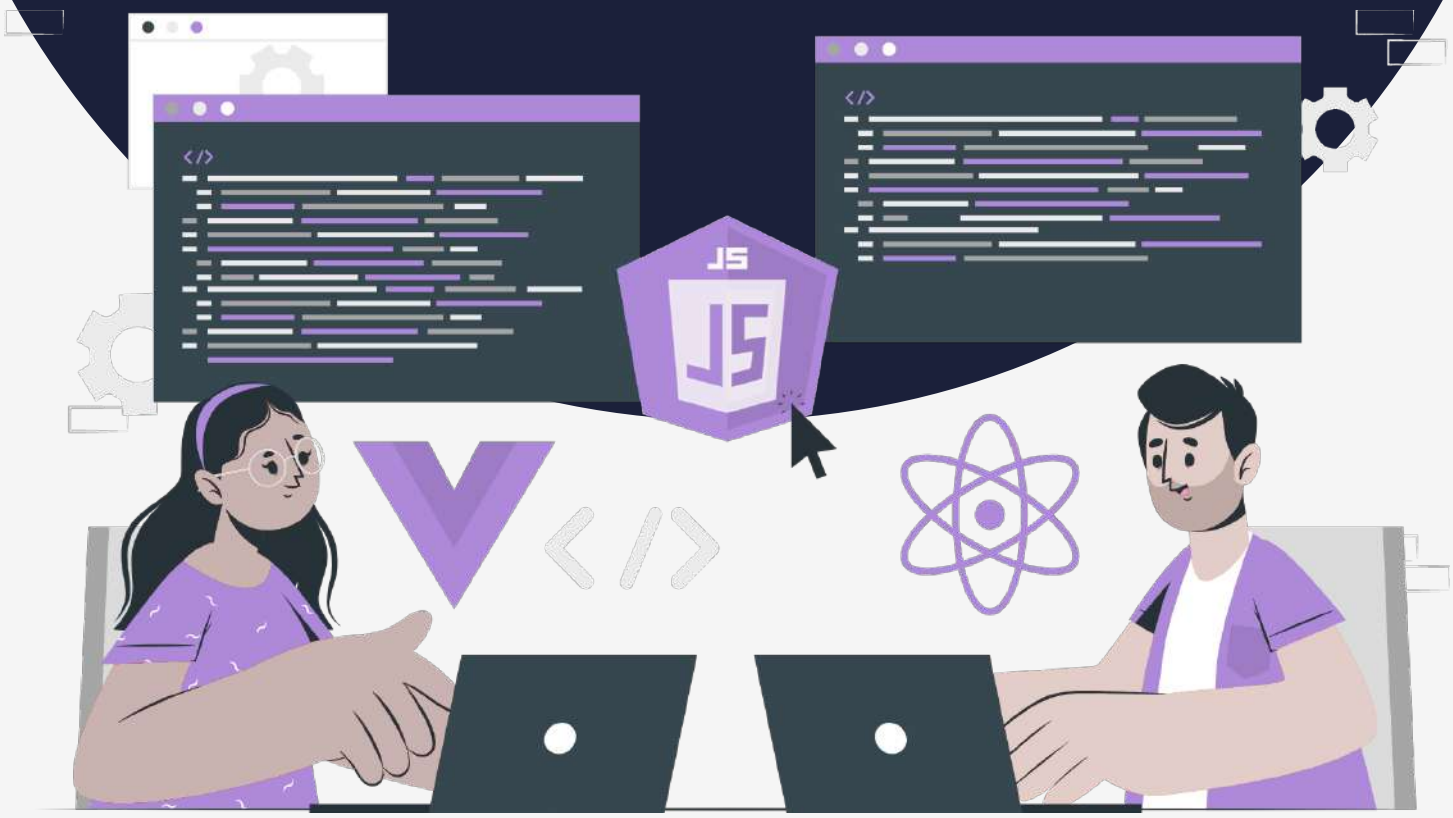
Overview of TypeScript

TypeScript is a typed superset of JavaScript that adds optional static types, class-based object-oriented programming, and other features to the dynamic and flexible nature of JavaScript. It is designed to make large-scale JavaScript development more manageable and scalable.

One of the main features of TypeScript is its support for static type checking. This means that the TypeScript compiler can analyze the code and check for type-related errors, such as passing a string where a number is expected before the code is run. This can help to catch and prevent errors early on in the development process, which can save time and effort in the long run.

Lesson:

Intro to Node JS (Runtime, V8, Libuv)



PW SKILLS

Topics Covered :

1. What is Node JS?
2. Features of Node JS.
3. Applications of Node JS
4. How to install Node JS?
5. What is Runtime?
6. Runtime in JavaScript.
7. How does JavaScript Runtime work?
8. What is V8?
9. Key features of V8.
10. What is Libuv?
11. Key features of Libuv

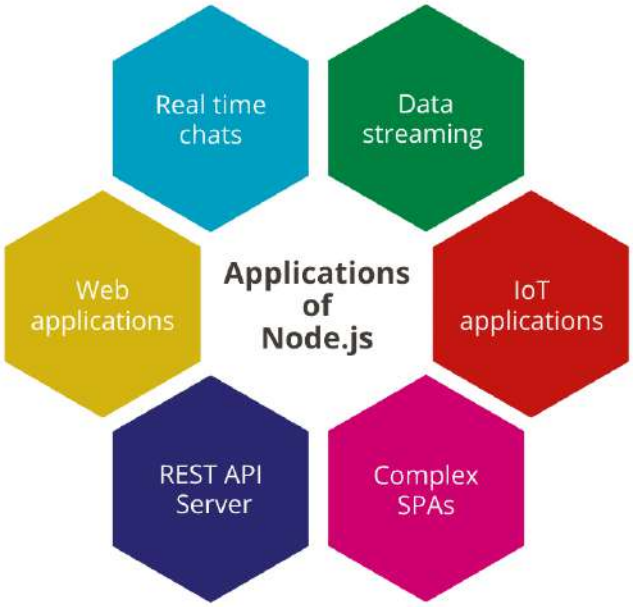
What is Node JS?

Node.js is an open-source, cross-platform JavaScript runtime environment that enables developers to run JavaScript code on the server side. It allows developers to create concurrent connections and real-time applications such as chat, gaming, and other types of applications needing real-time processing. It also has a large and active developer community that supports and maintains a vast ecosystem of open-source libraries and modules, making it easy to add functionality and integrate with other technologies.

Features of Node JS

1. **Easy:** Node.js is quite easy to start with. It's a go-to choice for web development beginners. With a lot of tutorials and a large community getting started is very easy.
2. **Scalable:** Node.js is a single-threaded program, which means that it can only handle one task at a time. However, it can handle a large number of connections at the same time without any delay. This is what makes Node.js very scalable. Because it can handle many connections simultaneously, it can handle a large number of requests and provide a fast response time, making it a great option for building scalable applications that can handle a lot of traffic.
3. **Speed:** Non-blocking thread execution means that Node.js can run multiple tasks at the same time without having to wait for one task to finish before starting another. This makes Node.js faster and more efficient because it can work on multiple tasks simultaneously, rather than having to wait for one task to finish before starting another. This allows Node.js to handle a large number of tasks with fewer resources and less delay.
4. **Packages:** A vast set of open-source Node.js packages is available that can simplify your work. There are more than one million packages in the NPM ecosystem today.
5. **Multi-platform:** Cross-platform support allows you to create SaaS websites, desktop apps, and even mobile apps, all using Node.js.
6. **Maintainable:** Node.js is an easy choice for developers since both the frontend and backend can be managed with JavaScript as a single language.

Applications of Node JS:



How to install Node JS

To install Node.js, you can follow these steps:

1. Go to the official Node.js website (https://nodejs.org/)
2. Download the installer for your operating system (Windows, macOS, or Linux)
3. Run the installer and follow the prompts to install Node.js on your system
4. Verify that Node.js is installed by opening a terminal or command prompt and typing "node -v". This should display the version of Node.js that you have installed.

Alternatively, you can use a package manager like apt or brew to install Node.js on Linux or macOS. You can refer to the Node.js official website for more information.

What is Runtime?
Runtime is a piece of code that implements portions of a programming language's execution model. Doing this, allows the program to interact with the computing resources it needs to work. Runtimes are often integral parts of the programming language and don't need to be installed separately.

Runtime is also when a program is running. That is, when you start a program running on a computer, it is a runtime for that program. In some programming languages, certain reusable programs or "routines" are built and packaged as a "runtime library." These routines can be linked to and used by any program when it is running.

Runtime in JavaScript.
In JavaScript, runtime refers to the period during which a JavaScript program is executed by a JavaScript engine. When a JavaScript program is loaded into a web page or runs using a command-line tool, the JavaScript engine starts to execute the code. The JavaScript engine follows a set of rules, known as the JavaScript specification, to interpret and execute the code.

During runtime, the JavaScript engine creates an execution context, a data structure containing the variables, functions, and other resources needed to execute the code. The engine then starts to execute the code, line by line, in the order it appears in the program.

JavaScript is a single-threaded language, which means that only one task can be executed at a time. However, JavaScript provides mechanisms such as callbacks and promises, to handle asynchronous operations and avoid blocking the execution of the code.

It's worth noting that JavaScript's runtime is different from other languages' runtime as when it is used for developing web applications, it runs on the browser on client side and for server side programming , it is run with Node.js. So the environment and the runtime are different.

How does JavaScript Runtime work?

JavaScript runtime works by interpreting and executing JavaScript code. When a JavaScript program is loaded into a web page or runs using a command-line tool, the JavaScript engine starts to execute the code. The engine follows a set of rules, known as the JavaScript specification, to interpret and execute the code.

The JavaScript engine creates an execution context for each function call and a global execution context for the entire program. The execution context contains the variables, functions, and other resources needed to execute the code. The engine then starts to execute the code, line by line, in the order it appears in the program.

JavaScript is a single-threaded language, which means that only one task can be executed at a time. However, JavaScript provides mechanisms such as callbacks and promises, to handle asynchronous operations and avoid blocking the execution of the code.

The JavaScript engine also includes a garbage collector, which constantly monitors the heap and frees up memory that is no longer being used. This is done in a way that does not interrupt the program execution.

JavaScript runtime in a browser is executed in a sandboxed environment, which means that it is limited in the resources it can access and what it can do. This is to protect the user's computer from malicious code. For example, JavaScript code running in a browser cannot access the user's file system or make arbitrary network connections.

JavaScript runtime on the server side, with Node.js, has more freedom and can access the file system, network resources, and more. The runtime is also different as it uses the V8 JavaScript engine developed by Google instead of the browser's JavaScript engine.

In summary, JavaScript runtime works by interpreting and executing JavaScript code in the order it appears. The JavaScript engine creates execution contexts, manages the execution flow, and handles memory management through garbage collection. The runtime also has some limitations in a browser environment to protect the user's computer.

What is V8?

V8 is an open-source JavaScript engine developed by Google which is used in the Google Chrome browser and also used in the runtime environment for Node JS. It's responsible for executing JavaScript code within the browser, parsing and compiling JavaScript code into machine code for faster execution.

V8 is known for its high performance, scalability, and compatibility with modern JavaScript features and APIs. The engine is implemented in C++ and is designed to handle complex, large-scale JavaScript applications, making it ideal for use in high-performance web browsers and server-side JavaScript environments.

The source code for V8 is freely available, allowing developers to contribute to its development and use it in their own projects. The V8 project has been actively maintained and developed by Google since its creation, and it continues to be a critical component of the Google Chrome browser and other web-based applications.

Key features of V8

- **Written in C++:** V8 is implemented in C++, making it fast and efficient. The C++ code is compiled into machine code, which can be executed directly by the computer's hardware, resulting in the fast and efficient execution of JavaScript code.
- **Dynamic Optimization:** V8 uses dynamic optimization techniques to continuously optimize the execution of JavaScript code, making it faster over time. The engine uses a Just-In-Time (JIT) compiler that generates optimized machine code for the code that is executed most frequently.
- **Heap Allocation:** V8 uses a compact, high-performance heap to allocate memory for JavaScript objects and data. The heap is designed to minimize fragmentation and maximize available memory, allowing V8 to handle large and complex JavaScript applications with ease.
- **Garbage Collection:** V8 uses a garbage collector to automatically reclaim memory that is no longer in use. This helps prevent memory leaks and improves the stability of JavaScript applications.
- **Modern JavaScript Features:** V8 supports modern JavaScript features, including ECMAScript 6 and later, as well as various APIs, such as Web Assembly, the Document Object Model (DOM), and more.
- **Open-Source:** V8 is an open-source project, with the source code available for anyone to use, modify, and contribute to. This makes it an attractive option for developers building web-based applications, as it provides access to a powerful and well-maintained JavaScript engine.

What is Libuv?

libuv is a cross-platform library for asynchronous I/O, developed for use in Node.js. It provides a uniform API for working with asynchronous I/O operations, such as file system operations, network communication, and timers.

Libuv abstracts away the differences between various operating systems and provides a consistent, high-level API for performing asynchronous I/O operations. This allows developers to write asynchronous code that works seamlessly across different platforms, without having to worry about the underlying operating system details.

In addition to its core I/O functionality, libuv also provides other features, such as thread pooling, process management, and signaling. These features make it a versatile and powerful library for building scalable and efficient applications.

Libuv is an open-source project and is used by a number of popular projects, including Node.js, and several other applications and frameworks. The library is actively maintained and has a large and active community of contributors and users, making it a well-supported and widely used tool for asynchronous I/O.

Key features of Libuv

- **Cross-Platform:** libuv abstracts away the differences between various operating systems and provides a consistent, high-level API for performing asynchronous I/O operations. This allows developers to write asynchronous code that works seamlessly across different platforms, without having to worry about the underlying operating system details.

- **Asynchronous I/O:** The core of libuv is its support for asynchronous I/O. The library provides APIs for working with file system operations, network communication, and other I/O operations that can take time to complete. These APIs allow developers to perform I/O operations in an asynchronous manner, which can improve the performance and scalability of their applications.

- **Event Loop:** libuv provides an event loop that allows developers to write asynchronous code in a simple and straightforward manner. The event loop listens for events, such as the completion of an I/O operation, and dispatches a callback function to handle the event. This allows developers to write code that is responsive and efficient, without having to deal with complex multi-threading or blocking I/O operations.

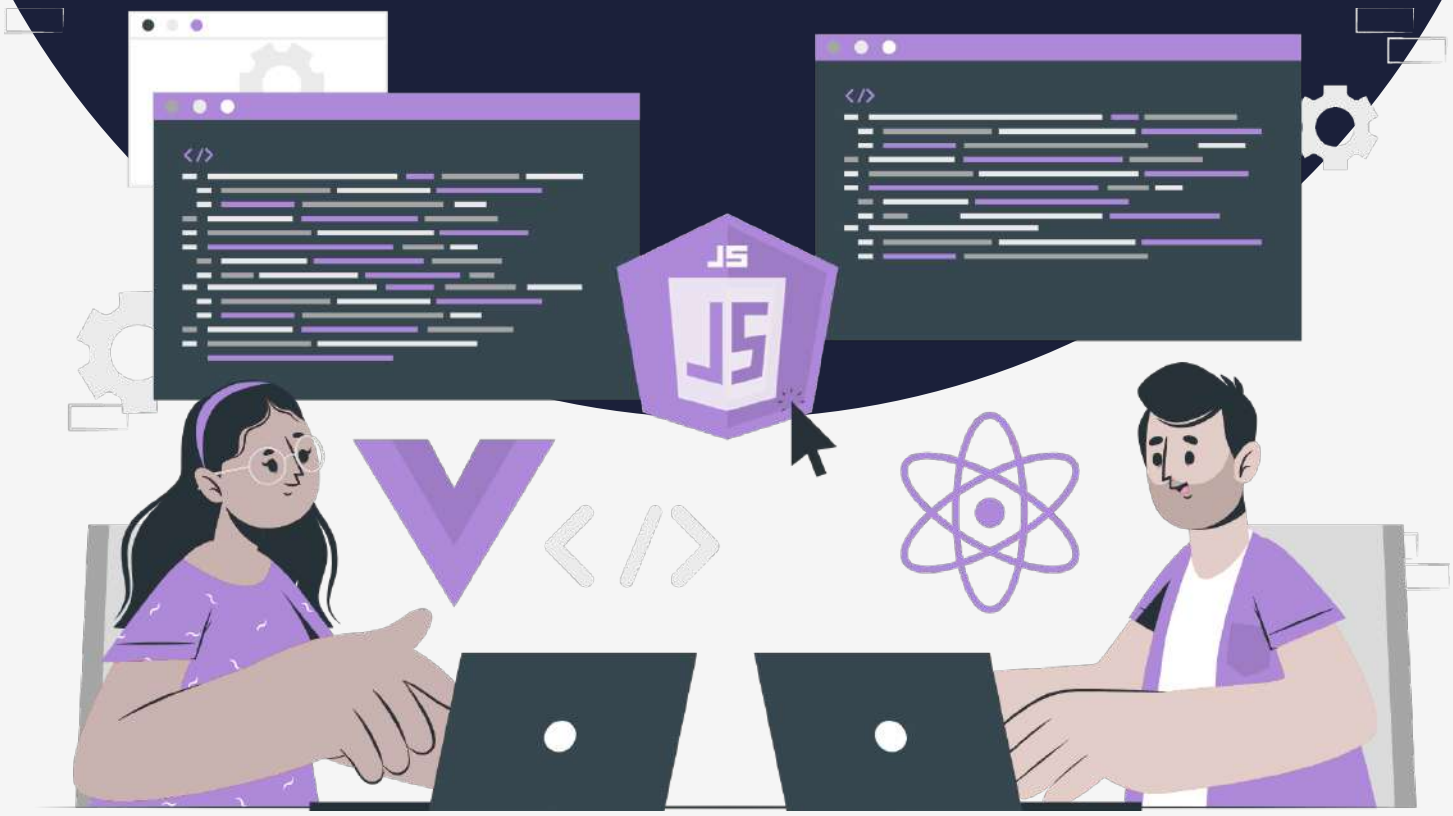
- **Thread Pool:** libuv provides a thread pool that allows developers to perform blocking I/O operations in a separate thread, freeing up the main thread to continue processing other events. This can improve the performance and scalability of applications, particularly when dealing with large amounts of data.

- **Process Management:** libuv provides APIs for working with processes, including process creation, process control, and process signals. This makes it easy to build multi-process applications, such as servers, that can take advantage of multiple cores or processors.

- **Open-Source:** libuv is an open-source project and is freely available for use. The source code is available on GitHub, and the library is actively maintained by a large and active community of contributors and users.

Lesson:

Values and Datatypes



PW SKILLS

Topics Covered :

1. Introduction to datatypes.
2. Datatypes in Javascript.

Data types are used to define the way the data is stored in memory. Storing data is an essential part of programming as it enables the manipulation, processing, and sharing of information within a program.

The data type is a classification of data according to the type of value that we want to operate on.

JavaScript is a dynamically typed language, which means the data type is defined by the engine itself during execution, the programmer need not explicitly declare the data type during defining. JavaScript engine is powerful enough to determine the type of data that we declare.

Following are the data types of JavaScript:

1. String
2. Number
3. Bigint
4. Boolean
5. Undefined
6. Null
7. Symbol
8. Object

String:
Strings are a data type used for representing text. A string is a sequence of characters, enclosed in single or double quotes.

```
"I am learning JavaScript"
"I am happy to learn from PW Skills"
```

Number:
Numbers are a data type used for representing numeric values. Numbers can be an integer, whole numbers, or decimal values [floating point values].

```
90
102.5
```

Other possible number values are infinity and NaN.

Infinity is a special value that is greater than any number.

Number.POSITIVE_INFINITY is Infinity and
Number.NEGATIVE_INFINITY is -Infinity.

```
Number.POSITIVE_INFINITY;
Number.NEGATIVE_INFINITY;
```

NaN stands for "Not a Number" and is a special value that represents the result of an undefined or unrepresentable mathematical operation.

```
NaN
Infinity
```

Bigint
In JavaScript, there is a maximum safe number that can be represented by the Number data type, is approximately 2^53 - 1. This means that integers larger than this value may lose precision when represented as a JavaScript number.

Similarly, there is also a minimum safe number that can be represented by the Number data type, which is approximate -(2^53 - 1). This means that integers smaller than this value may lose precision when represented as a JavaScript number.

For numbers greater than the maximum safe number or lesser than the minimum safe number, the BigInt data type can be used.

The BigInt data type number can also be treated as a regular number by adding n to it at the end.

```
902345874n
```

Boolean
Boolean is a logical type that is either true or false.

```
true
false
```

Booleans are often used to represent the outcome of a logical comparison or the result of a logical operation.

We will look into operators in further lectures.

Undefined
undefined is a special value that indicates that a variable or property has been declared but has not been assigned a value.

We will look into variables in the next lecture.

Null
Null means nothing or empty value. It is often used to indicate that a variable or property has no value.

We will look into variables in the next lecture.

Symbol
A Symbol is a datatype that can be used as an object property key.

We will be exploring objects in depth in further lectures. One thing to note is that Symbols are unique and are often used as an object property keys.

Object

In javascript, numbers, strings, booleans, undefined, null are called as primitive data types.
Objects are non-primitive data because they can hold multiple primitive data types within them.

The object data type can contain:

1. Array
2. Object.

An array is a special type of object that is used to store a collection of elements. Each element can be of any data type and can be accessed by an index, which is a zero-based number starting from 0.

Arrays are created by using the [ ] brackets. We will be looking at Arrays in depth in further lectures.

```
[1, 2, "Javascript", true, null];
```

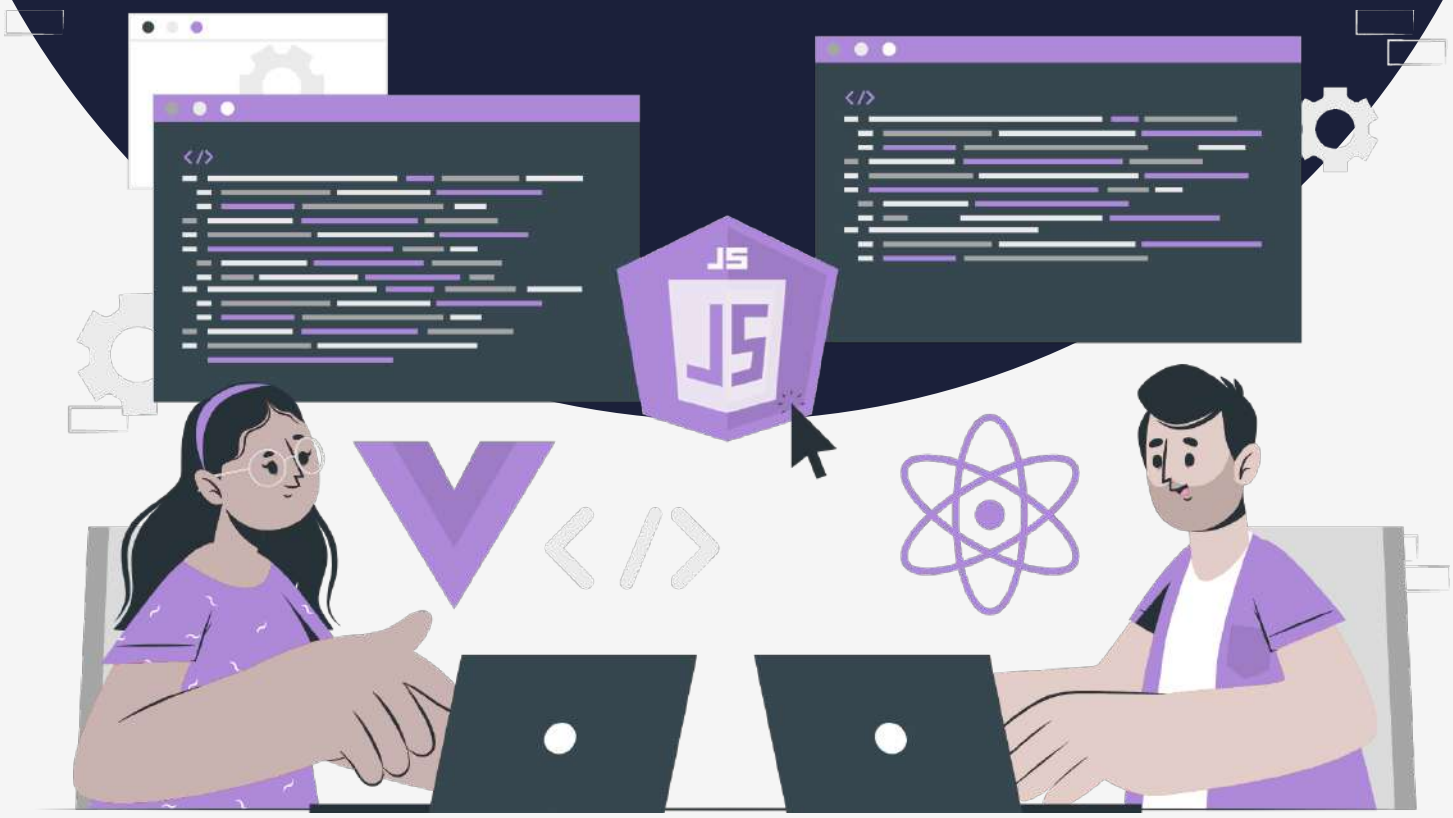
Object is a collection of properties, where each property has a name and a value. Objects are usually created by curly brackets { }. Each property of an object can be accessed using the dot notation or the bracket notation.

We will be looking at objects in depth in further lectures.

```
{
      name: "Mithun",
      company: "PW Skills"
}
```

Lesson:

Variables and typeof



PW SKILLS

Topics Covered :

1. Introduction to variables.
2. Creation of variables.
3. Naming variables in Javascript.
4. Assigning values to variables.
5. Introduction to typeof.
6. Benefits of typeof.

Variables are like containers, they are used to hold the information we'll need when programming. Variables store data of any datatype that can be used throughout a program.

Variable means anything that can vary. Variables hold the data value and it can be changed anytime we want.

Creating a variable is also called declaring a variable. There are four ways to create a variable in JavaScript.

1. var keyword.
2. let keyword.
3. const keyword.
4. No definition.

```
var name = "PW Skills";
let name = "PW Skills";
const name = "PW Skills";
name = "PW SKills";
```

Naming variables in JavaScript
When naming the variables, we must consider making the names descriptive and easily understandable. This will make our program easy to read and understand in the future when we have to refactor it.

Here are some rules one should look out for when naming variables:

Variable names should begin with either a letter or an underscore or a dollar sign.

```
var name = "PW SKills";
var Name = "PW SKills";
var _name = "PW SKills";
var $name = "PW SKills";
```

Variable names should not begin with numbers or special characters except the underscore and dollar signs.

Keywords are reserved words that have a specific meaning and cannot be used as variables. Keywords like if, else, for should not be used as variable names.

Variable names are case-sensitive. That means name and Name are different variable names.

To ensure consistency in naming variables adopt one of the following naming conventions in naming variables.

```
var companyName = "PW Skills"; // Camel Case
var CompanyName = "PW Skills"; // Pascal Case
var company_name = "PW SKills"; // Snake Case
```

Assigning values to a variable.
Storing data in a variable is also called assigning a value to a variable. To store data in a variable(assign value to a variable), use the = symbol. Place the variable name on the left side of the = symbol and place the value to store in the variable goes on the right side of the = symbol.

The = symbol is called the assignment operator. We will look into operators in depth in further lectures.

Variables can be created before assigning values to them.

```
var name;
name = "PW Skills";
```

Whenever we create a variable without assigning a value to it, by default javascript stores undefined [absence of the value].

Values can also be assigned to variables at the moment of creating them. Creating variables and assigning values to them at the same time is known as initializing a variable.

```
var name = "PW Skills";
var students = 12345678;
var enrolledToFSWD = true;
```

```
var name = "PW Skills", students = 12345678,
enrolledToFSWD = true;
```

typeof
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 operator is very useful for determining the data type of a given variable. In addition, it can also be used to check for null values.

```
var name = "PW Skills";
var students = 12345678;
var enrolledToFSWD = true;
var mentorDetails = {
   name: "Anurag",
   yearsOfExperience: 4,
};
var techStack = ["HTML", "CSS", "Javascript", "Node", "React", "Express"];
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 mentorDetails); // object
console.log(typeof techStack); // object
console.log(typeof couponCode); // object
console.log(typeof endDate); // undefined
console.log(typeof studentsEnrolled); // number
```

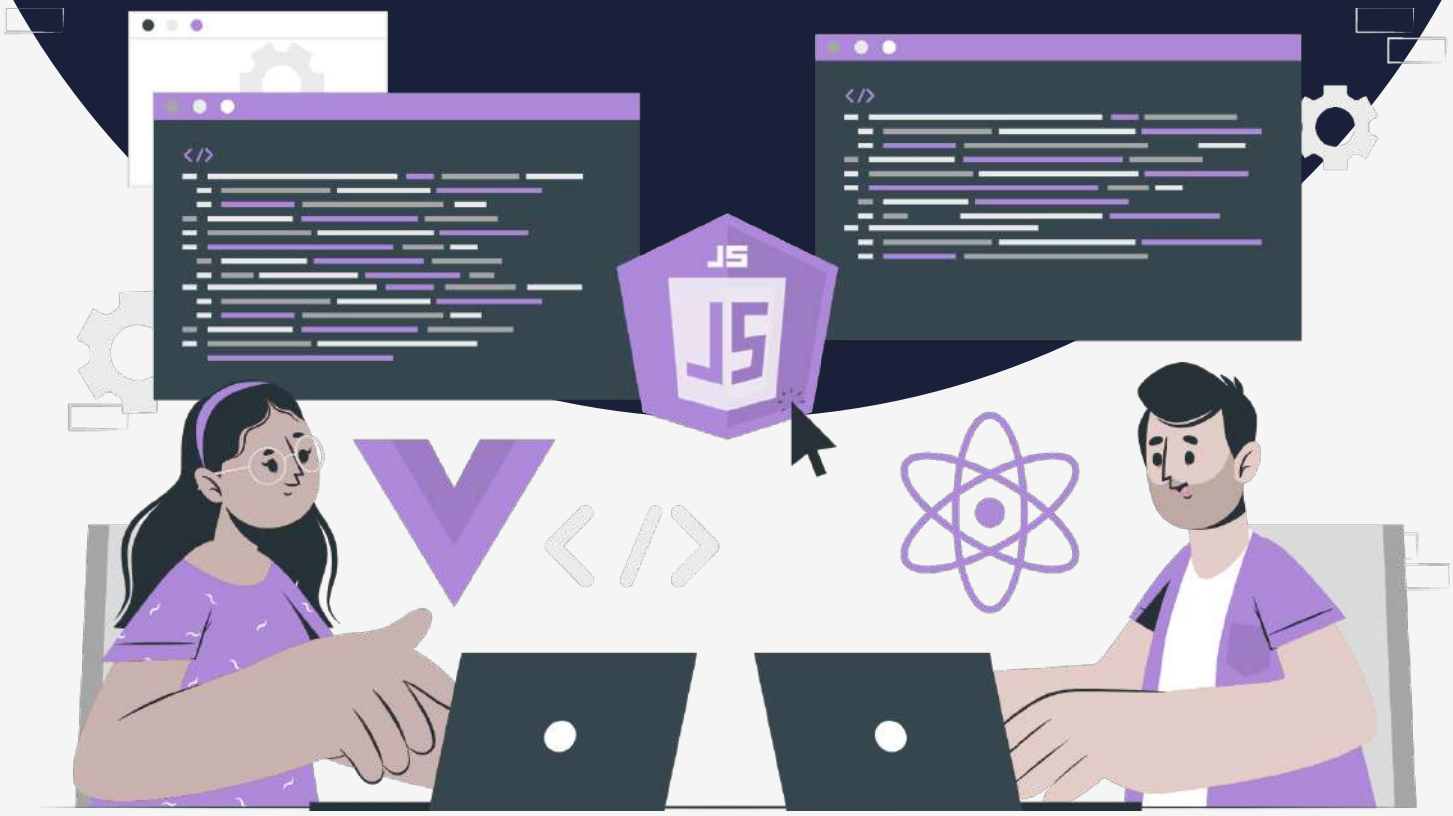
There are a few benefits to using the "typeof" operator in JavaScript. First, it is a convenient way to check if a variable is of a certain data type without having to check for conditions. Second, it can be used as a debugging tool to help check why a particular piece of code is not working as expected if any datatype issues. Finally, it can help prevent errors in code by giving a clear understanding of the data types that are being used.

How can the "typeof" operator be used to detect errors?

The "typeof" operator can be used to detect errors in the code. If we try to access a variable that has not been declared, we will get an error message. This is because "typeof" returns "undefined" for undeclared variables.

Lesson:

Operators



Topics Covered :

1. Introduction to operators.
2. Operators & Operands.
3. Operators in JavaScript.

Basically, programming is done for a variety of reasons and one of the important reasons why programming is done is to operate on the values in the variables with high accuracy and faster speed. To perform these operations we need operators.

What are operators and operands?

```
5 + 3 = 8
```

Here, we have to add 5 and 3 to get the final value. So we are using + to add these two values. And the final value is 8.

In the above example, the + sign performs an arithmetic operation to add these two values. So this + sign is the operator here. In modern programming languages, we use this + sign to add two values.

An operand is a data value that the operator will carry out the actions. It is the values on which we operate. So, in the above example, 4 and 3 are operands.

Operators are used in programming to perform operations on variables and values.

Operators in JavaScript.

1. Assignment Operators
2. Arithmetic Operators
3. Comparison (Relational) Operators
4. Logical Operators
5. Bitwise Operators

Assignment Operators:
Assignment operators are used to assign values to variables.

Example:

```
var name = "PW Skills";

// Assignment Operators//

1. Assign
var name = "PW Skills"; // name = "PW Skills"//

2. Add & Assign
var students = 100; // students = 100
students += 50; // students = 150
```

```
// 3. Subtract & Assign
var students = 100; // students = 100
students -= 50; // students = 50

// 4. Multiply & Assign
var students = 100; // students = 100
students *= 50; // students = 5000

// 5. Divide & Assign
var students = 100; // students = 100
students /= 50; // students = 2

// 6. Modulus & Assign
var students = 100; // students = 100
students %= 50; // students = 0

// 7. Exponential & Assign
var students = 100; // students = 100
students **= 2; // students = 10000
```

The commonly used assignment operator is =. We will understand other assignment operators such as +=, -=, *=, etc. once we learn arithmetic operators.

Arithmetic Operators
We use arithmetic operators to do mathematical operations like addition, subtraction, multiplication, division, etc. It simply takes numerical values as operands, performs an arithmetic operation, and returns a numerical value.

- Addition (+): Adds two values together.
- Subtraction (-): Subtracts one value from another.
- Multiplication (*): Multiplies two values together.
- Division (/): Divides one value by another.
- Modulus(%): Returns the remainder of a division operation.
- Exponentiation(**): raises to the power of.
- Increment Operator(++): Increases the value by 1.
- Decrement Operator(--): Decreases the value by 1.

In JavaScript, the "++" operator is used for both pre-increment and post-increment operations. Pre increment operator increments the value and returns the incremented value immediately. Post increment operator increments the value but returns the original value itself. The same follows with Pre decrement and Post decrement operators.

```
// Arithmetic Operators

var num1 = 100;
var num2 = 2;

// 1. Addition
var result = num1 + num2; // 102

// 2. Subtraction
var result = num1 - num2; // 98

// 3. Multiplication
var result = num1 * num2; // 200

// 4. Division
var result = num1 / num2; // 50

// 5. Modulus
var result = num1 % num2; // 0

// 6. Exponential
var result = num1 ** num2; // 10000

// 7. Pre Increment
var num = 10;
var result = ++num; // num = 11 ; result = 11

// 8. Post Increment
var num = 10;
var result = num++; // result = 10 ; num = 11

// 9. Pre Decrement
var num = 10;
var result = --num; // num = 9 ; result = 9

// 10. Post Decrement
var num = 10;
var result = num--; // result = 10 ; num = 9
```

Adding two numbers, will return the sum, but adding a number and a string will return a string:

```
// Adding Two Numbers

num1 = 10;
num2 = 5;
result = num1 + num2; // 15
result_type = typeof result; // number

// Adding A Number and a string
num1 = 100;
str1 = "Welcome ";
result = str1 + num1; // Welcome 100
result_type = typeof result; // string

// Adding A Number and a string
num1 = 100;
str1 = "10";
result = str1 + num1; // 10100
result_type = typeof result; // string
```

Comparison (Relational) Operators
Comparison operators compare two values and return a boolean value, either true or false.

The following are the comparison operators in javascript:

- **Equal (==):** Compares two values for equality, returns true if they are equal and false if they are not.
- **Strict equal (===):** Compares two values for equality and type, returns true if they are equal and of the same type, and false if they are not.
- **Not equal (!=):** Compares two values for inequality, returns true if they are not equal and false if they are.
- **Strict not equal (!==):** Compares two values for inequality or type, returns true if they are not equal or not of the same type, and false if they are.
- **Greater than (>):** Compares two values, returns true if the left operand is greater than the right operand, and false otherwise.
- **Greater than or equal to (>=):** Compares two values, returns true if the left operand is greater than or equal to the right operand and false otherwise.
- **Less than (<):** Compares two values, returns true if the left operand is less than the right operand and false otherwise.
- **Less than or equal to (<=):** Compares two values, returns true if the left operand is less than or equal to the right operand and false otherwise.

```
// Comparison Operators

var num1 = 10;
var num2 = 20;
var num3 = 10;

var str1 = "10";
var str2 = "20";
```

```
// 1. Equal to
var result = num1 == num2; // false
var result = num1 == num3; // true

// 2. Strict Equal
var result = num1 === num3; // true
var result = num1 === str1; // false

// 3. Not equal
var result = num1 != num2; // true
var result = num1 != num3; // false

// 4. Strict Not Equal
var result = num1 !== num3; // false
var result = num1 !== str1; // true

// 5. Greater than
var result = num1 > num3; // false
var result = num2 > num3; // true

// 6. Greater than or Equal to
var result = num1 >= num3; // true
var result = num2 >= num3; // true

// 7. Lesser than
var result = num1 < num3; // false
var result = num2 < num3; // false

// 8. Lesser than or Equal to
var result = num1 <= num3; // true
var result = num2 <= num3; // false
```

Logical Operators:
Logical operators perform logical operations and return a boolean value, either true or false.

Logical operators in javascript:

- **Logical AND (&&):** This operator returns true if both operands are true, and false otherwise. It is a logical first operator, meaning that if the first operand is false, the second operand is not evaluated.
- **Logical OR (||):** This operator returns true if at least one of the operands is true, and false otherwise. Like && operator, it is also a short-circuit operator.
- **Logical NOT (!):** This operator inverts the Boolean value of the operand. If the operand is true, the operator returns false, and if the operand is false, the operator returns true

These logical operators can be used to perform a variety of tasks, such as testing multiple conditions or combining multiple conditions in a single expression. They are often used with other techniques and operations to perform more complex tasks, such as flow control and data validation. We will be looking at their applications in further lectures.

```
// Logical Operators

var num1 = 10;
var num2 = 20;
var num3 = 10;

// 1. Logical AND
var result = num1 ⩾ num3 && num1 == num3; // true
var result = num1 ⩾ num2 && num1 == num3; // false

// 2. Logical OR
var result = num1 ⩾ num3 || num1 == num3; // true
var result = num1 ⩾ num2 || num1 == num3; // true
var result = num1 ⩾ num2 || num1 > num3; // false

// 3. Logical NOT
var result = num1 == num3; // true
var result = !(num1 == num3); // false
```

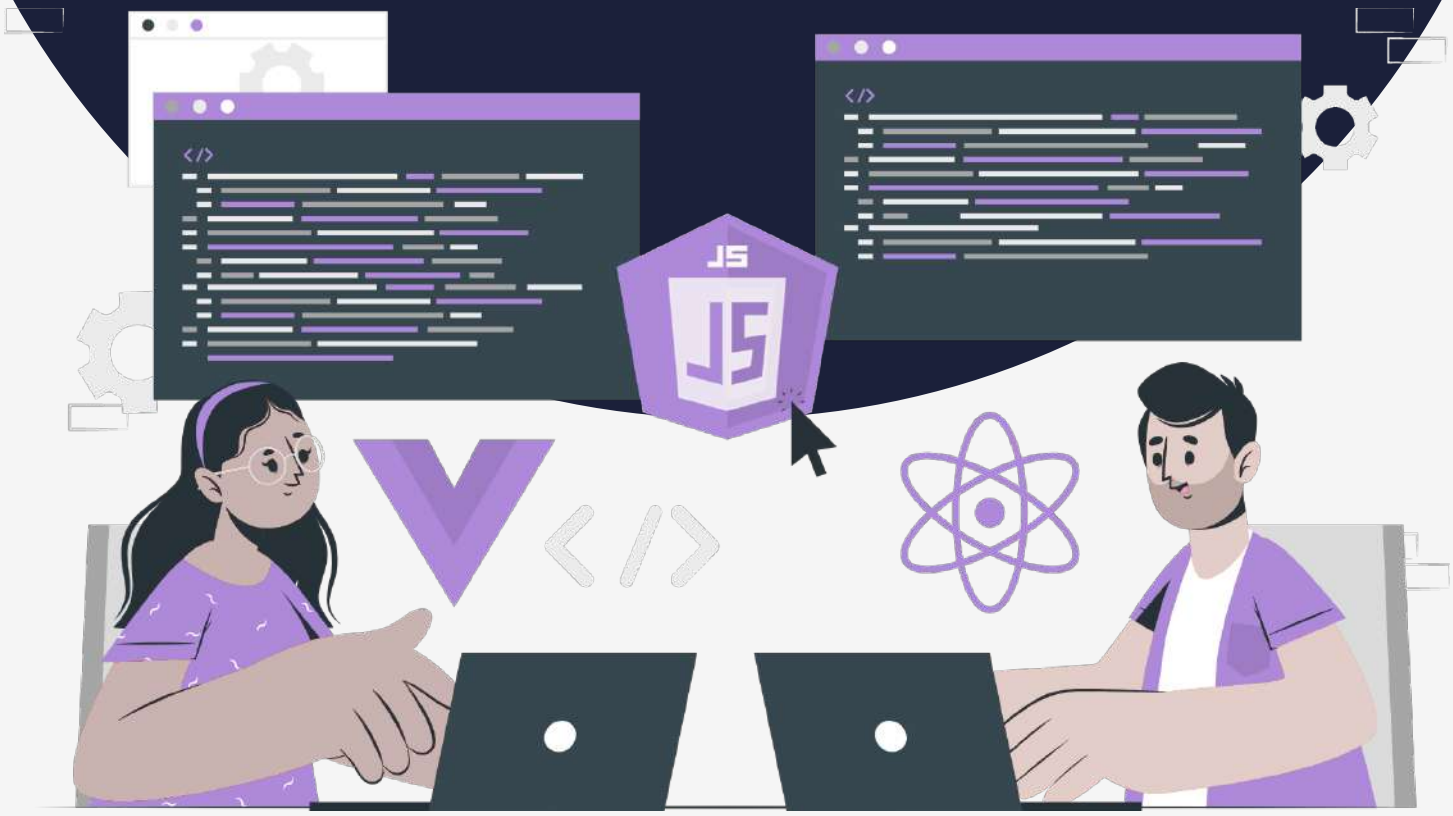
Bitwise Operators:
Bitwise operators perform operations on binary representations of numbers. Bitwise operators are particularly useful in low-level programming [programming that deals with the underlying hardware of a computer] and optimization, but they can also be used in a wide range of other areas, including data compression, encryption, and game development.

Bitwise operators in JavaScript are:

- **Bitwise AND (&):** This operator compares each bit of the first operand to the corresponding bit of the second operand, and if both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.
- **Bitwise OR (|):** This operator compares each bit of the first operand to the corresponding bit of the second operand, and if at least one of the bits is 1, the corresponding result bit is set to 1.
- **Bitwise XOR (^):** This operator compares each bit of the first operand to the corresponding bit of the second operand, and if the bits are different, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.
- **Bitwise NOT (~):** This operator inverts all the bits of the operand, effectively swapping 1s for 0s and 0s for 1s.
- **Left Shift (<<):** This operator shifts the bits of the operand to the left by the specified number of places, adding zeroes to the right.
- **Right Shift (>>):** This operator shifts the bits of the operand to the right by the specified number of places, discarding the bits shifted out.

Lesson:

What are Conditions, If, If-else, if-else-if



Topics Covered :

1. Conditionals in Programming.
2. Conditionals in JavaScript.
3. Ways of writing conditionals in Javascript.
4. Introduction to if, if-else, and if-else if.
5. Check if the given number is even or odd.
6. Order of evaluation for if, else if, and else 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.

There are three ways of writing conditionals in Javascript:

- If/else Statement
- Switch Statement
- Ternary Operator

In this lecture let's look at the basic syntax of these conditionals and later will look at the code along with examples.

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

```
if (condition) {
    // code to be executed if the condition is true
}
```

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.

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) {
  // code to be executed if the condition is true
} else {
  // code to be executed if the condition is false
}
```

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

```
if (condition1) {
  // code to be executed if condition1 is true
} else if (condition2) {
  // code to be executed if condition1 is false and condition2 is true
} else {
  // code to be executed if condition1 and condition2 are both false
}
```

You can chain as many else if as you want

JavaScript also has a ternary operator, which is a shorthand way to write a simple if-else statement. The syntax for a ternary operator is as follows:

```
Condition ? true-expression: false-expression;
```

It is a shorthand way to write an if-else statement.

JavaScript also has a switch statement. A switch statement allows you to check for multiple conditions and execute different codes depending on the value of a particular expression. The syntax for a switch statement is as follows:

```
switch (expression) {
  case value1:
    // code to be executed if expression == value1
    break;
  case value2:
    // code to be executed if expression == value2
    break;
  default:
  // code to be executed if the expression is not equal to any of the values
}
```

The switch statement is useful when you have a large number of conditions to check and the conditions are based on the value of a single expression.

Conditional statements in JavaScript are an important tool for controlling the flow of a program. They allow you to execute different codes depending on the outcome of an evaluation, and they come in several forms: if-else, else-if, ternary operator, and switch statement. Understanding how and when to use these statements is essential for writing effective JavaScript code.

Let's understand the basic conditionals which are if, if-else, and if-else-if. To understand these concepts we will be looking into a simple example problem.

The if, else, and else if keywords are used in programming to control the flow of a program based on certain conditions. These conditions are typically evaluated as either true or false.

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 are 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.

Let's look at each condition one by one.

```
// Input

var num = 10;
```

In the above block of code, we have declared a variable named "num" and assigned it the value of 10. The variable num will be our input.

Now let's handle the condition number 01 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.

From the previous lecture, we know the syntax of the if statement, and we will write the code considering the same syntax.

```
// Input

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");
}
```

This code block checks if the variable "num" is an even number using an if statement. The if statement checks the condition num % 2 == 0.

The modulus operator % is used to find the remainder of dividing the variable "num" by 2.

If the remainder is 0, it means that the number is completely divisible by 2 and it is an even number.

If the condition is true, the code inside the if block will be executed, which is console.log("The number given is an even number");. This will output the message "The number given is an even number" to the console, confirming that the number is even.

```
// Input
var num = 10; // Output: The number given is an even number

// Condition 01: Any number that is completely divisible [ remainder must be 0 ] by 2
then 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.

```
// Input
var num = 11; // Output:

// 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");
}
```

Since we are not getting any output if the condition is false. It's time to handle the second 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.

```
// Input

var num = 11; // Output: The number given is an odd number

// 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");
} else {
  console.log("The number given is an odd number");
}

// Input
var num = 10; // Output: The number given is an even number

// 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");
} 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.

We also have our third condition which is 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 condition fails it is an odd number.

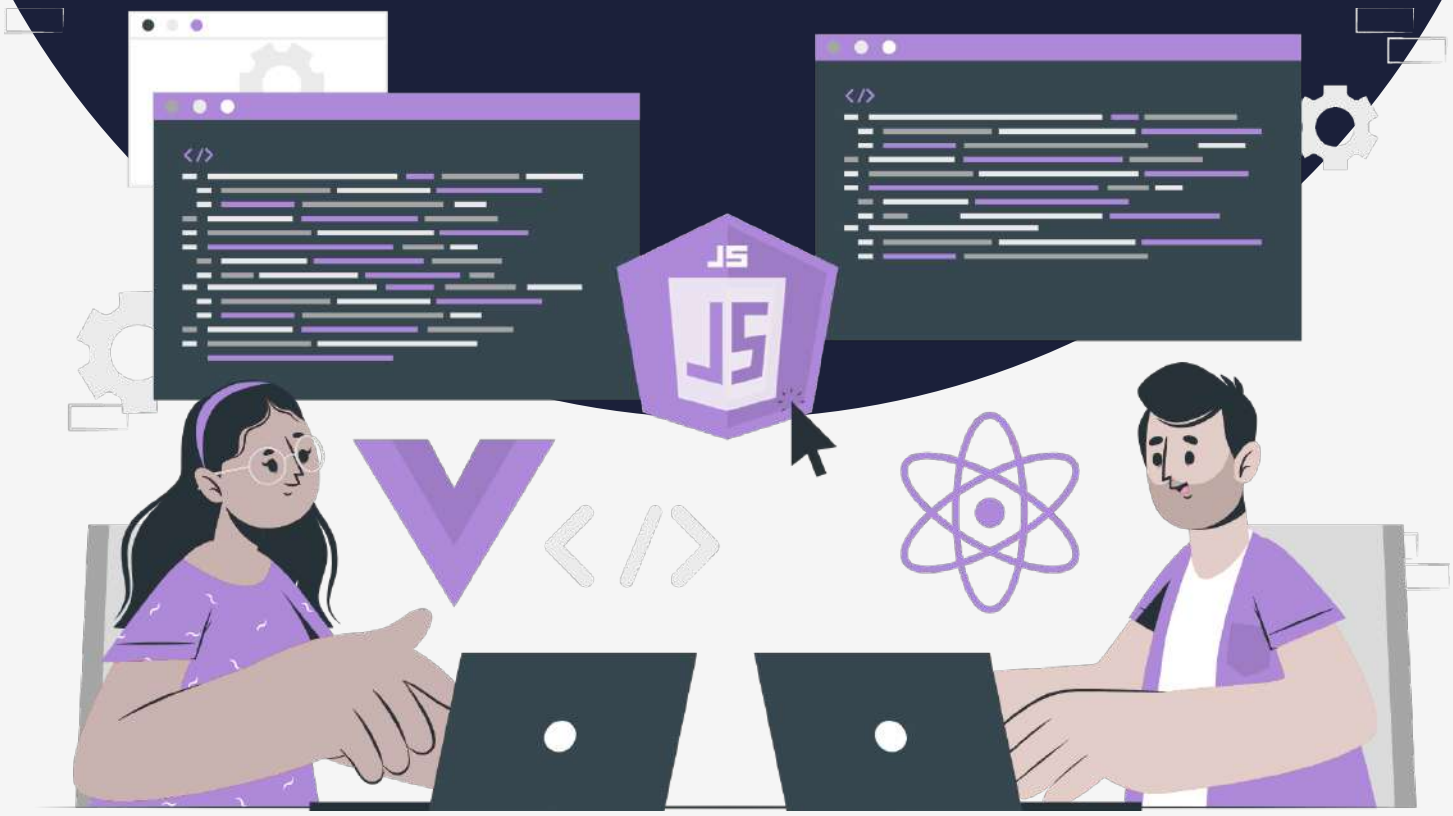
```
// Input

var num = 10; // Output: The number given is an even number

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


// Input
var num = 11; // Output: The number given is an odd number

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


// Input
var num = 0; // Output: Zero is neither an odd number nor an even number

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

In JavaScript, the order of evaluation for if, else if, and else statements are as follows:

1. The if statement is evaluated first. If the condition in the if statement is true, the code inside the if block will be executed and the program will skip over any subsequent else if or else statements.
2. If the condition in the if statement is false, the program will move on to the next else if statement and evaluate its condition. If the else if condition is true, the code inside its block will be executed and the program will skip over any remaining else if or else statements.
3. If all of the if and else if conditions are false, the program will move on to the else statement and execute the code inside its block.

Lesson:

Switch Case



PW SKILLS

Topics Covered :

1. Introduction to switch case.
2. Break statement.
3. Syntax of switch case.
4. Implementation of switch case.
5. When to use switch statements or if/else statements.

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 statements allow you to execute different blocks of code based on different conditions (cases) being matched. It is a more efficient way to use multiple if/else checks.

We can have many case statements but unlike if statements, they are executed from the first matched value until a break is specified. It checks only for equality in the values. So, in order to come out of the switch statement we need to specify a break or the condition must match the default case.

A break statement is used to stop the execution of a loop or switch statement before it has completed all of its iterations or cases. We will look into the break statement in further lectures.

Switch statements use strict comparisons and for the code to be able to execute the values must be the same type.

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

```
switch (expression) {

  case condition1:
    // code to execute;
    break;

  case condition2:
    // code to execute;
    break;

  case condition3:
    // code to execute;
    break;

  default:
  // default code;
}
```

Let's compare the switch syntax with the if statement.

```
switch (expression) {
    case condition1: // if (expression === condition1 then execute this block)
    // code to execute;
    break;

    case condition2: // if (expression === condition2 then execute this block)
    // code to execute;
    break;

    case condition3: // if (expression === condition3 then execute this block)
    // code to execute;
    break;

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

Let's now implement a switch statement considering an example. We represent our weekday 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.

```
// Input
var day = 5; // Output: Thursday

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");
}
```

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").

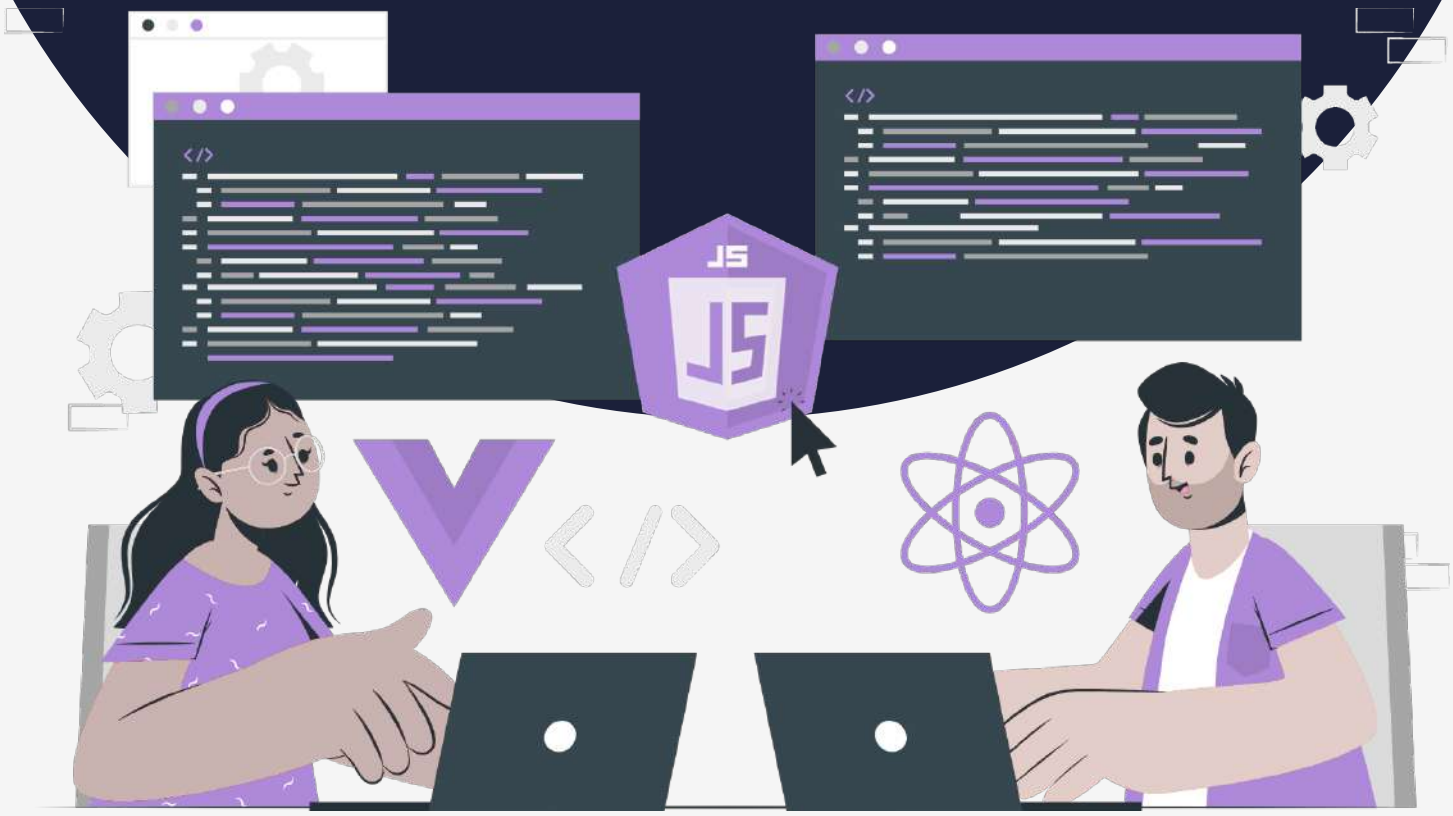
The break statement at the end of the case ensures that the code execution exits the switch statement after the matching case has been executed, so the code in the other cases and the default block will not be executed.

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.

Lesson:

Ternary Conditions



PW SKILLS

Topics Covered :

1. Introduction to ternary operator.
2. Why majority of developers use the ternary operator?
3. Syntax of Ternary operator.
4. Code Implementation.
5. Chaining using the Ternary operator.

Operators are used to assigning, compare, and evaluate one, two, or more operands. We have different types of operators in JavaScript. These include comparison, arithmetic, logic, ternary, etc. Ternary operators have 3 blocks 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>
```

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.

We can do this in either of the ways:

1. Using if-else statements.
2. Using the ternary operator.

```
var isTheUserLoggedIn = true;

// Using if-else statement

if (isTheUserLoggedIn) {
  console.log("PW Skills lab Access Granted !!");
} else {
  console.log("PW Skills lab Access Denied !!");
}
```

```
var isTheUserLoggedIn = true;

// Using ternary operator

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

While learning if-else statements we have seen how to chain if-else multiple statements.

In the same way, we can chain multiple ternary operators.

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 can we do this using the ternary operator.

```
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
```

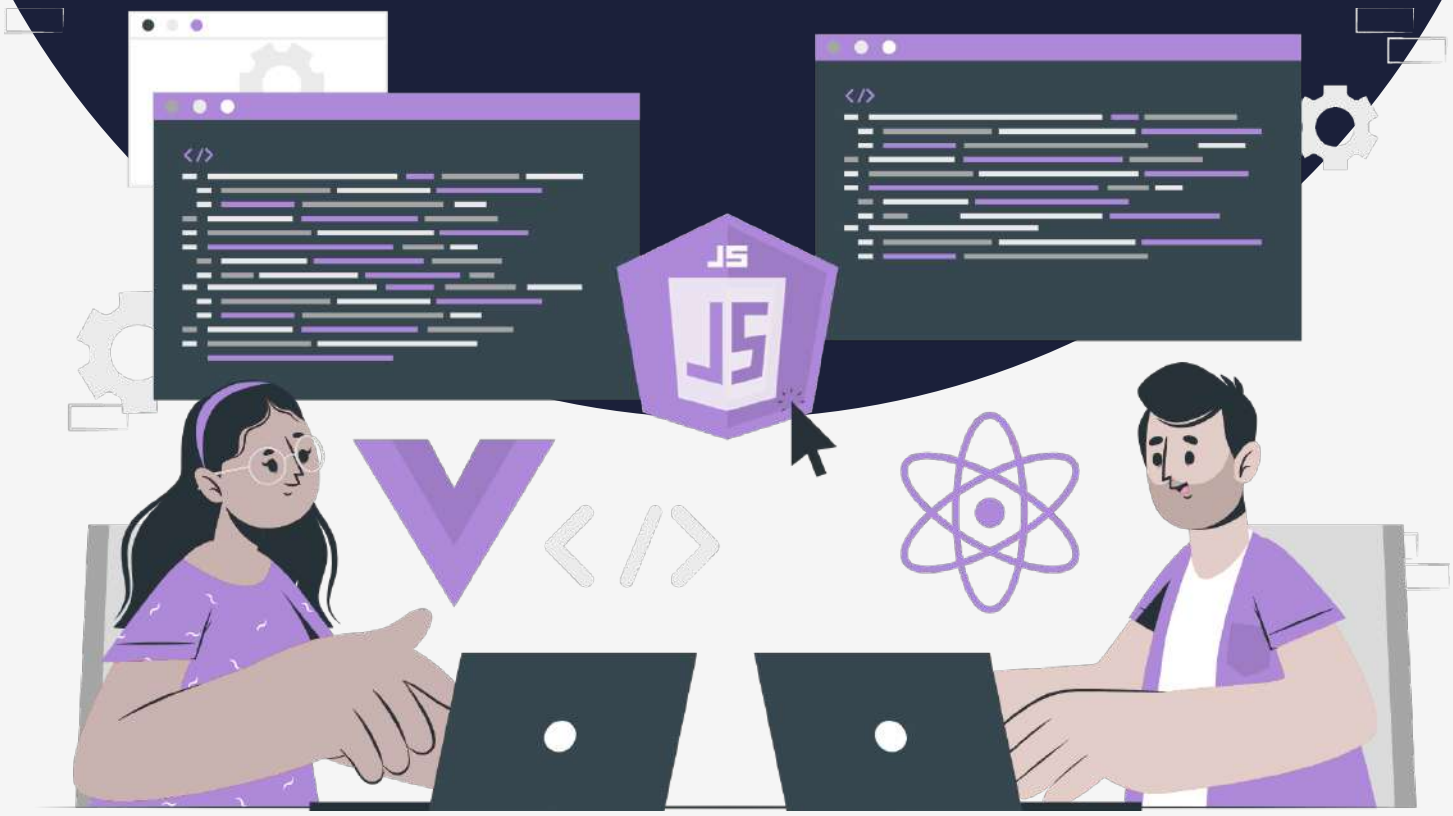
```
var isTheUserLoggedIn = true;

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
```

```
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 as they are not readable and debugging would be complex.

Lesson:

Introduction to loops



PW SKILLS

List of content:

What are loops?

As you may see in the diagram, a loop, is a structure, series, or process, the end of which is connected to the beginning.

In programming, a loop is a set of instructions that are repeatedly carried out until a specific condition is met in computer programming. Typically, a certain action is taken, such as receiving and modifying a piece of data, and then a condition is verified, such as determining whether a counter has reached a predetermined value. If not, the following instruction in the sequence directs the computer to go back to the first instruction in the series and repeat it. A loop is a fundamental concept in programming that is frequently applied when creating programs.

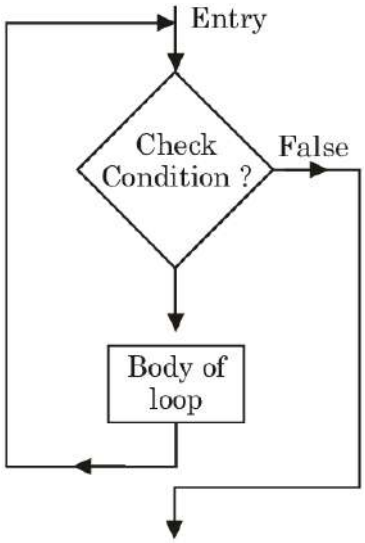
A loop that never ends is known as an infinite loop.
As a result, the loop keeps repeating until the operating system notices it and crashes the program, or until another event happens (such as having the program automatically terminate after a certain duration of time).

Why loops?
The control statement and the body of a loop can be conceptually separated into two components. The prerequisites for the execution of a loop's body are listed in the control statement of the loop. The conditions in the control statement must be true for each loop iteration. The block of code or series of logical assertions that will be executed repeatedly make up the body of a loop.

As a result, you will save time when using a loop in your program by eliminating the need to repeatedly write the body of the loop's code. As long as the conditions in the control statement are true, the code block will be run several times. The loop will end when the conditions in the control statement are no longer true. The loop will continue to run even if the conditions are not explicitly stated in the control statement. We refer to these loops as infinite loops. A loop automatically becomes infinite if there is no termination condition specified in the control statement.

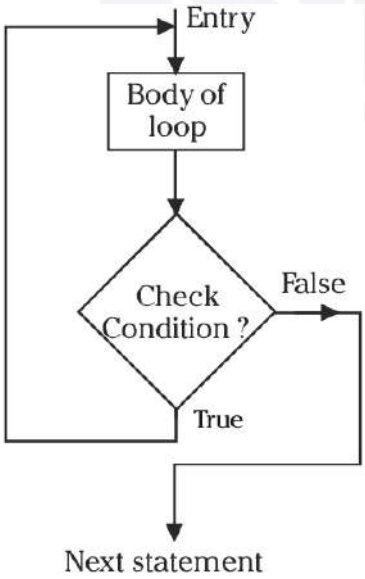
Types of loops:
When you have a basic understanding of the syntax and purpose of the different forms of loops, the concept of "what is Loop" will become evident. In most computer programming languages, loops can be divided into two categories: entry-controlled loops and exit-controlled loops.

1. Entry controlled loop



The control statement is written right at the start of an entry-controlled loop. Pre-checking Loop is another name for this kind of loop. The control statements' conditions are initially verified, and the body of the loop is only run if the conditions are met. The lines of code in the Loop's body won't be run if the condition turns out to be false. For loop is an entry-controlled loop, which means the control statements are written at the very beginning of the loop structure.

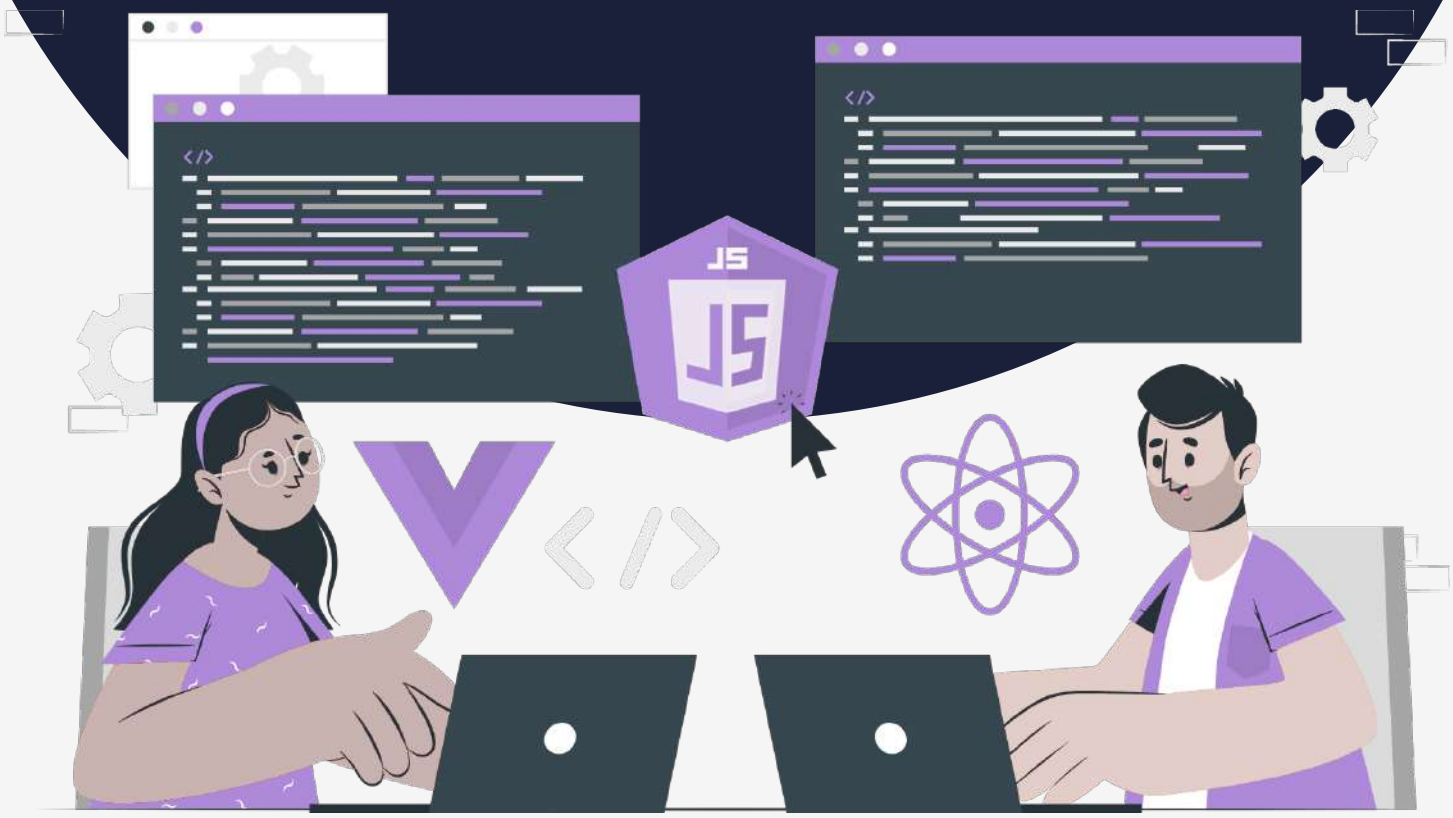
2. Exit controlled loop



Here, the control statements are written at the end of the loop structure which means that the loop will at least run once even if the condition of loop is false.
do-while loop is an exit-controlled loop.
Each of these loops would be discussed in the subsequent lectures.

Lesson:

While and Do-while loop



PW SKILLS

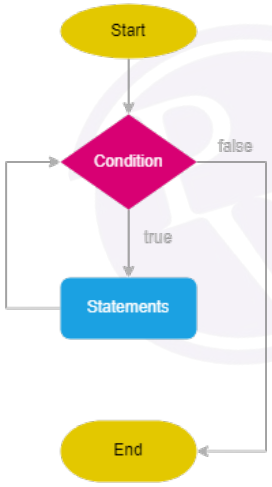
List of content:

Loops are used in programming to repeatedly run a block of code. A loop can be used, for instance, to repeat a message hundred times. It's only a simple illustration; loops may be used to accomplish much more.

While loop
Syntax :

```
while (condition) {
    // body of loop
}
```

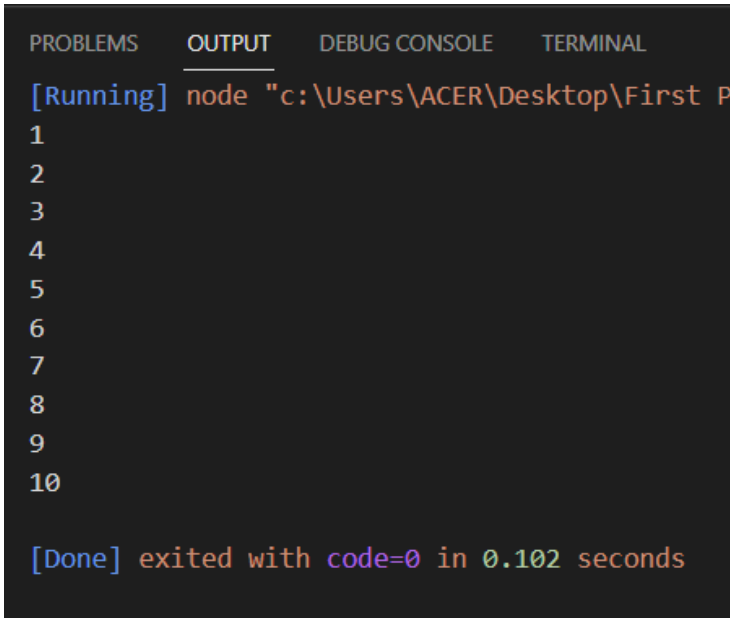
- The parentheses has the condition that is to be evaluated by a while loop ().
- The while loop's code is run if the condition evaluates to true.
- The condition is assessed again.
- until the condition is false, this process keeps going.
- The loop ends when the condition is evaluated as false.



Example 1: To print numbers from 1 to 10 using while loop

```
let i = 1, n = 10;
while (i ⩽ n)
{
    console.log(i);
    i=i+1;
}
```

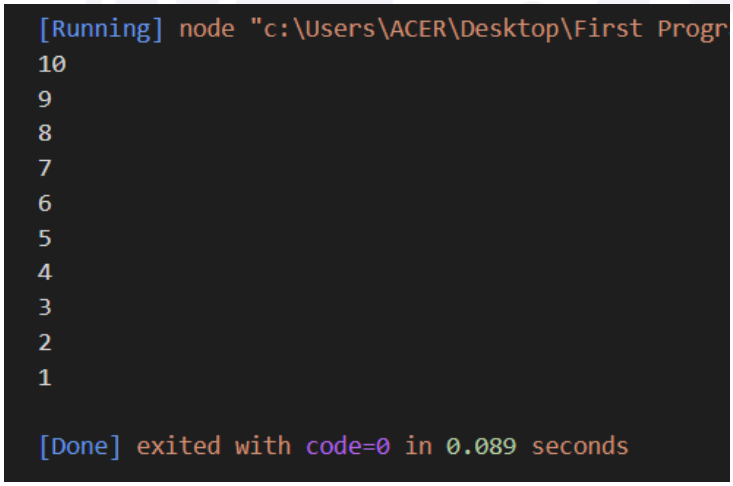
Output:



Example 2 : To print numbers from 10 to 1 in decreasing order.

```
let i = 1, n = 10;
while (n ⩾ i){
   console.log(n);
    n=n-1;
}
```

Output:



Do while loop

Syntax :

```
do {
    // body of loop
} while(condition)
```

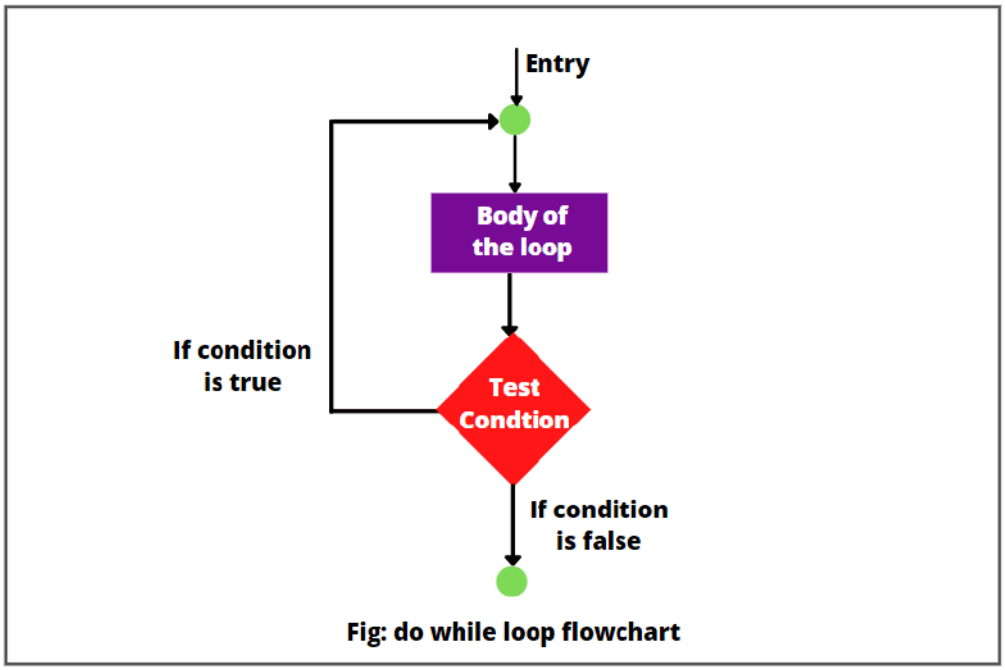


Fig: do while loop flowchart

- The loop's body is first executed. The condition is analyzed afterwards.
- The body of the loop within the do statement is repeated if the condition evaluates to true.
- A second assessment of the condition is made.
- The body of the loop within the do statement is repeated again if the condition evaluates to true.
- Up until the condition is evaluated as false, this process keeps going. The loop then ends.

Example 1 : Write a program to print numbers from 1 to 7 in a line using do-while loop

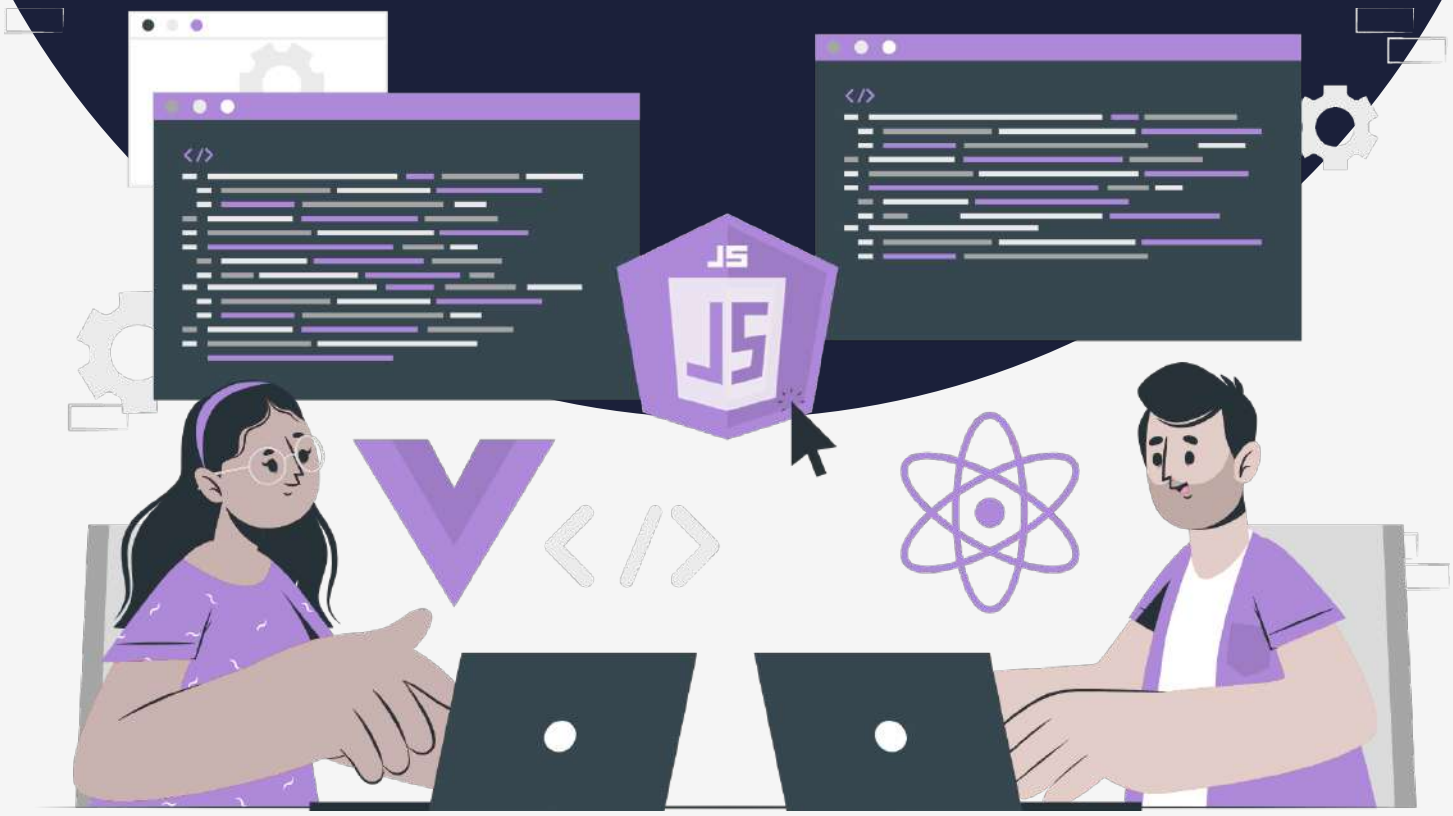
```
let result = '';
let i = 0;
do {
  i = i + 1;
  result = result + i;
} while (i < 7);
 console.log(result);
```

Output:

```
[Running] node "c:\Users\ACER\Desktop\First F
1234567

[Done] exited with code=0 in 0.105 seconds
```

Lesson:

For loop, Break and Continue



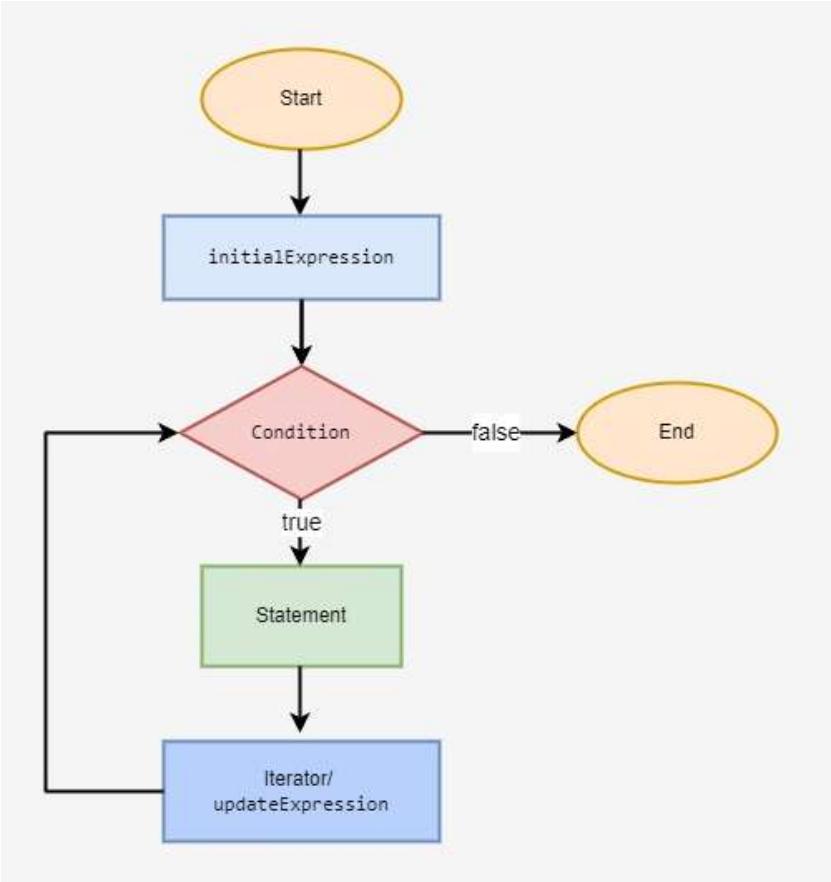
PW SKILLS

List of content:

For Loop

Syntax :

```
for (initial expression; condition; update expression) {
    // for loop body
}
```



- InitialExpression only ever executes once while initializing and declaring variables.
- The condition is assessed.
- The for loop is ended if the condition is false.
- The for loop's code block is performed if the condition is satisfied.
- When the condition is true, the update expression changes the initial expression's value.
- The condition is once more assessed. Up till the condition is false, this process keeps going.

Example 1: Using For loop print "PW Skills" 3 times.

```
for (let i = 0; i < 3; i++)
{
    let name = "PW Skills";
    console.log(name);
  }
```

Output:

```
[Running] node "c:\Users\ACER\Desktop\First Pr
PW Skills
PW Skills
PW Skills

[Done] exited with code=0 in 0.106 seconds
```

Example 2: Display a sequence of even numbers till 20

```
for (let i = 2; i ⩽ 20; i+=2) {
    console.log(i);
  }
```

```
[Running] node "c:\Users\ACER\Desktop\First Pr
2
4
6
8
10
12
14
16
18
20

[Done] exited with code=0 in 0.094 seconds
```

Nested for loop

An outer for loop and one or more inside for loops constitute a nested for loop. Control re-enters the inner for loop and initiates a fresh execution every time the outer for loop repeats.

In other words, every time the outer for loop repeats, the control will enter the inner for loop.

Nested for loop often looks like this:

```
// Outer for loop.
 for ( initialization; test-condition; increment/decrement )
 {
// Inner for loop.
   for ( initialization; test-condition; increment/decrement )
   {
     // statement of inner loop
   }
 // statement of outer loop
 }
```

The execution flow is something like this:

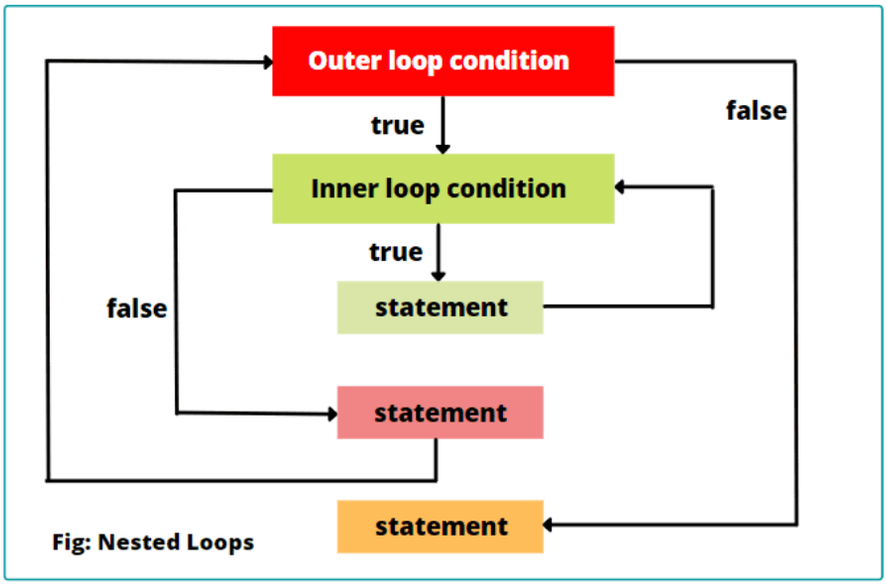


Fig: Nested Loops

Example 3 : Write a program to show the inner for loop values for each outer iteration in along with the outer "for" loop.

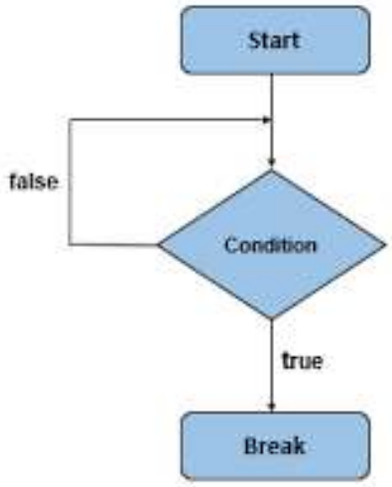
```
for(let i=1;i⩽3;i++){ //outer loop
    console.log("for i= " + i + " the innerloop values are")
    for(let j=1;j<3;j++){  //inner loop
        console.log("j= "+j)
    }
}
```

Output:

```
[Running] node "c:\Users\ACER\Desktop\Fi
for i= 1 the innerloop values are
j= 1
j= 2
for i= 2 the innerloop values are
j= 1
j= 2
for i= 3 the innerloop values are
j= 1
j= 2

[Done] exited with code=0 in 0.099 secor
```

Break Statement:
A loop such as a for, do...while, or while loop, a switch, or a label statement are all prematurely terminated by the break statement. The break statement's syntax is as follows:

break [label];



Example of using break with for loop:
Can you guess the output of the following code ?

```
for (let i = 0; i < 4; i++) {
  console.log(i);
  if (i == 2) {
    break;
  }
}
```

Output:

```
[Running] node "c:\Users\ACER\Desktop\First
0
1
2

[Done] exited with code=0 in 0.087 seconds
```

Here, we use an if statement inside the loop. If the current value of i becomes 2, the if statement will execute and the break statement will terminate the loop.
That is why we only see numbers till 2 in the output.

Example of using break with while loop

We will take the same example and see if it works in while loop

```
let i = 0;

while (i < 4) {
  console.log(i);
  i++;
  if (i == 3) {
    break;
  }
}
```

Output:

```
[Running] node "c:\Users\ACER\Desktop\First Pr
0
1
2

[Done] exited with code=0 in 0.092 seconds
```

Here again, we have used an if statement to check the condition for break, the moment i becomes 3, if statement is executed leading to loop termination.

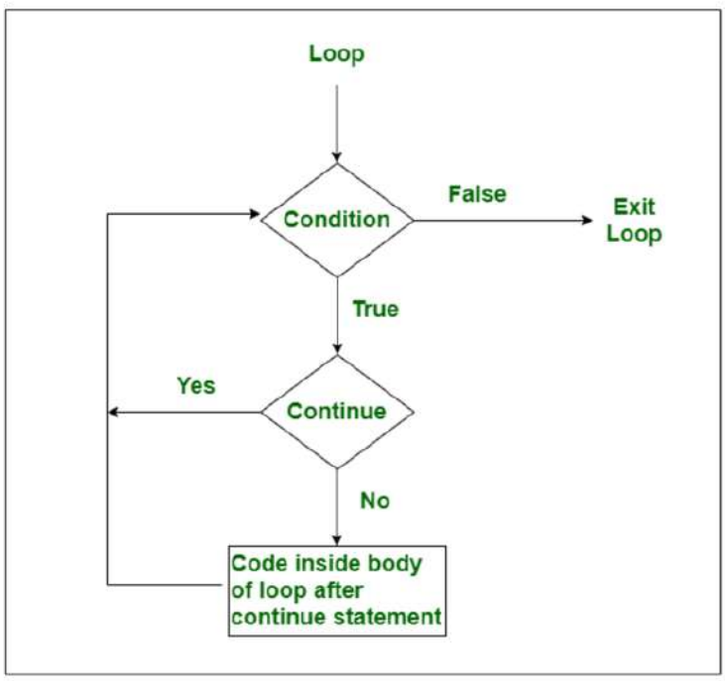
Notice and observe the difference between for and while loops for getting the same result.

Continue Statement:
The current iteration of the loop is skipped when using the continue statement, and the program moves on to the subsequent iteration.

The syntax of the continue statement is:

continue [label];

label is optional and rarely used.

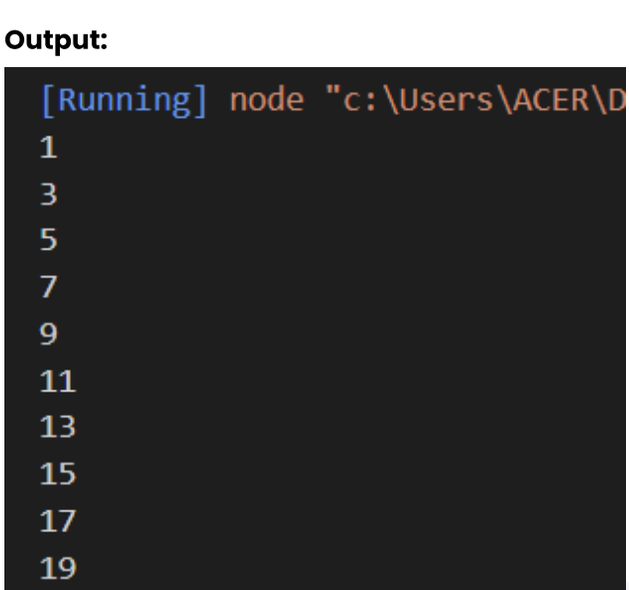


Example of using continue in for loop

Write a program to display only odd numbers till 20:

```
for (let i = 0; i < 20; i++) {
    if (i % 2 === 0) {
      continue;
    }
    console.log(i);
  }
```

Output:



Here, the for loop iterates through the values from 0 to 20.

The remainder of the division of the current value of i by 2 is returned by the i%2 expression.

The continue statement, which skips the current iteration of the loop and moves to the iterator expression i++, is executed if the remainder is zero. If not, the value of i is output to the console.

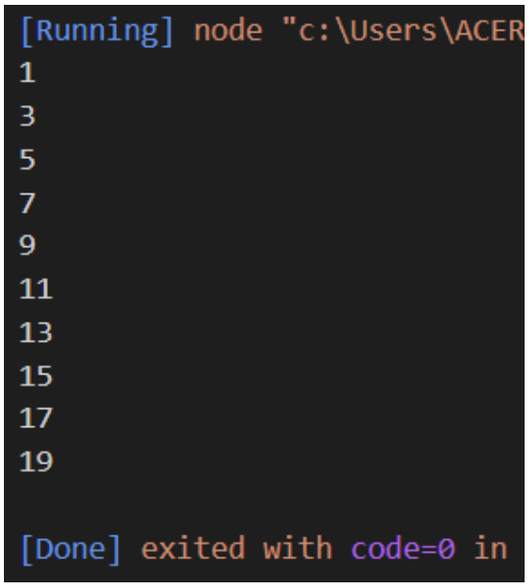
Example of using continue in while loop

We will use the same example here and implement it with while loop

Write a program to display only odd numbers till 20

```
let i = 0;
while (i < 20) {
  i++;
  if (i % 2 === 0) {
    continue;
  }
  console.log(i);
}
```

Output:



Here, the while loop iterates through the values in this example from 0 to 20.

The remainder of the division of the current value of i by 2 is returned by the i%2 expression.

The continue statement, which skips the current iteration of the loop and moves to the iterator expression i++, is executed if the remainder is zero. If not, the value of i is output to the console.

We can also use a label here, to print the same result. If you are wondering how, here it is :

```
labelex:
for (let i = 0; i < 20; i++) {
    if (i % 2 === 0) {
      continue labelex;
    }
    console.log(i);
  }
```

Here, the continue statement skips the execution and takes it to labelex.
You can use a label to identify a loop and the continue statement, here, to tell a program whether to skip the loop or keep running it. Note that JavaScript has no goto statement(as in C/C++); you can only use labels with break and continue.
Labeled loops are easier to track and understand with respect to program flow in case of continue.

Output: