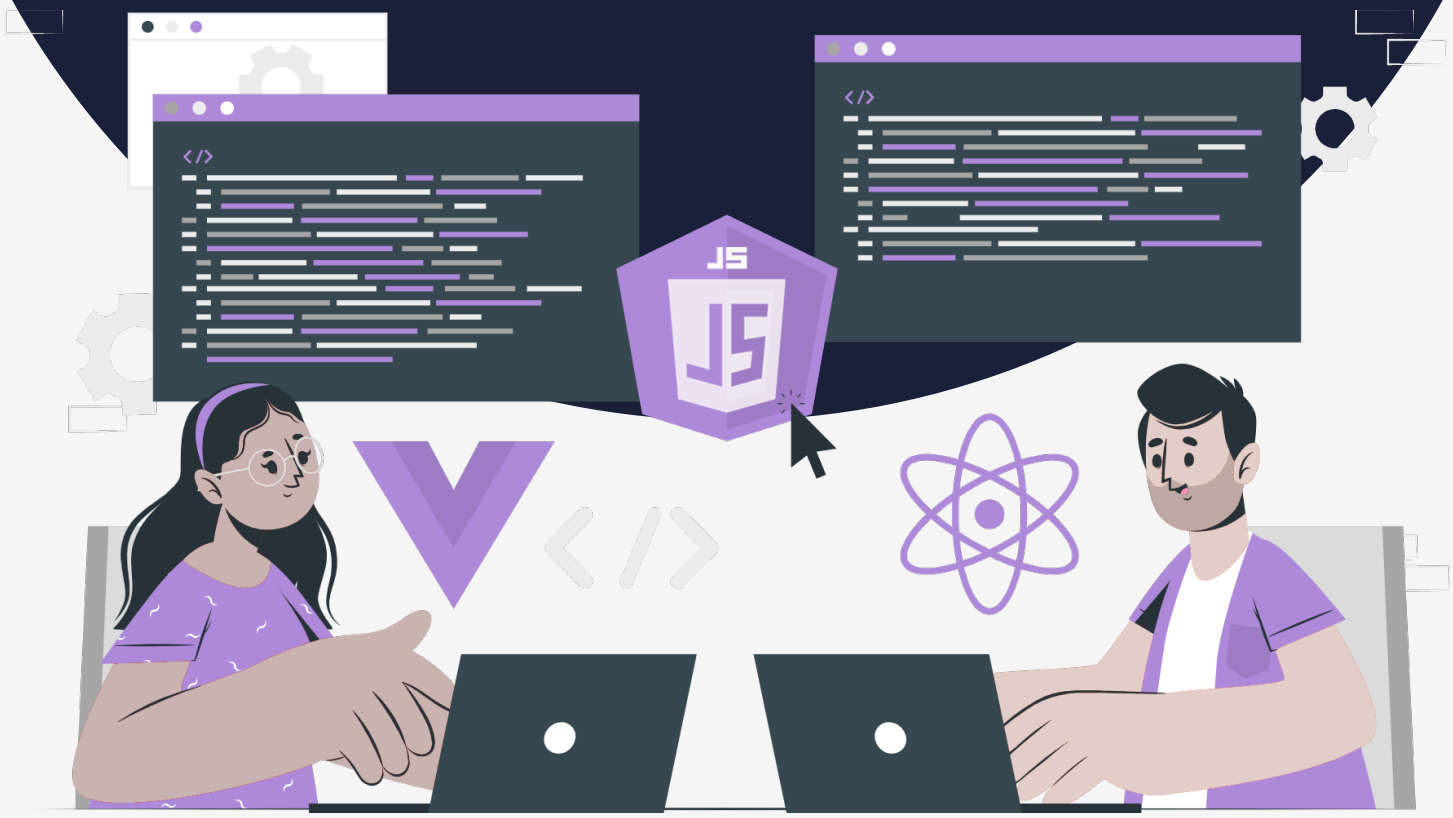
Lesson:

Let, var, and Const and Temporal Dead Zone



PW SKILLS

Topics Covered:

1. Introduction.
2. Var keyword.
3. Let Keyword.
4. Const Keyword.
5. Temporal Dead Zone.

In JavaScript, variables are used to store data values that can be accessed and manipulated throughout the program. There are three different ways to declare variables in JavaScript: var, let, and const. While var has been used traditionally to declare variables, the introduction of let and const in ES6 has added more flexibility and control over variable scope and behavior. However, when using let and const, there is a concept called the temporal dead zone.

Understanding the differences between var, let, and const, as well as the concept of the temporal dead zone, is essential to writing effective and efficient JavaScript code. In this lecture, we will be looking at var, let, and const along with temporal dead zone in depth.

var

In JavaScript, the var keyword is used to declare a variable. This keyword has been used since the early days of JavaScript and is still widely used today.

```
var name = "PW SKills";

console.log(name); // PW SKills
```

Variables declared with the var keyword have function scope, meaning that they are accessible only within the function in which they are declared or any nested functions within that function.

```
var name = "PW SKills";
console.log(name); // PW SKills
function printName() {
  console.log(name); // PW SKills
}

printName();

function youtube() {
  var channelName = "College Wallah";
}

console.log(channelName); // ReferenceError: channelName is not defined
```

In the above code, we first declare a variable name and assign it the value "PW SKills". We then log the value of the name to the console, which outputs "PW SKills".

We then define a function called printName(), which logs the value of the name to the console. When we call printName(), it also outputs "PW SKills", since the name is in the same scope.

Next, we define a function called youtube(), which declares a variable called channelName and assigns it the value "College Wallah". However, channelName is declared using the var keyword, which means that it is function-scoped and is only accessible within the youtube() function.

When we try to log the value of channelName to the console outside of the youtube() function, we get a ReferenceError because channelName is not defined in that scope.

Variables declared with the var keyword are subject to hoisting. This means that the variable declarations are moved to the top of their scope, regardless of where the actual declaration appears in the code. However, the variable assignments are not hoisted, only the declarations.

```
console.log(name); // undefined

var name = "PW SKills";
```

Declaring the same variable with var multiple times within the same scope is allowed and does not result in a syntax error. However, this behavior can lead to unexpected results and should generally be avoided.

```
var name = "PW SKills";
var name = "College Wallah";
console.log(name); // College Wallah
```

Function expressions declared with “var” are hoisted to the top of their scope but are not defined, which means that they are not accessible until they are assigned. Accessing them before their assignment throws an exception.

```
printName(); // TypeError: printName is not a function
var name = "PW SKills";
var printName = function () {
  console.log(name);
};
```

Arrow functions declared with var are hoisted but are not accessible until they are assigned.

Let.

Let is a keyword used to declare block-scoped variables. Block-scoped variables are only accessible within the block they are declared in, which is defined by curly braces {}.

```
let name = "PW SKills";
console.log(name); // PW SKills

{
  let name = "PW SKills";
}
console.log(name); // ReferenceError: name is not defined
```

Unlike var, which allows redeclaration of variables within the same scope, let does not allow redeclaration of variables within the same scope. Attempting to declare a let variable with the same name as an existing let variable within the same scope will result in a SyntaxError.

```
let name = "PW SKills";
let name = "College Wallah"; // SyntaxError: Identifier 'name' has already been declared.
```

Variables declared with let can be reassigned a new value after they have been initialized. This means that the value of the variable can change throughout the execution of the program.

```
let name = "College Wallah";
name = "PW Skills";

console.log(name); // PW Skills
```

Unlike var variables, which are hoisted to the top of their scope and initialized with a value of undefined, let variables are not hoisted to the top of their scope. Instead, they are only accessible after they have been declared.

```
console.log(name); // ReferenceError: name is not defined
let name = "PW SKills";
```

Like regular let variables, function expressions declared with let are also not hoisted to the top of their scope. This means that the function expression is only accessible after it has been declared.

```
let printName = function () {
  console.log("PW SKills");
};
printName(); // PW Skills

printName(); // ReferenceError: Cannot access 'printName' before initialization

let printName = function () {
  console.log("PW SKills");
};
```

Similar to regular let variables, arrow functions declared with let are also not hoisted to the top of their scope. This means that the arrow function is only accessible after it has been declared.

const.

Const is another keyword used for declaring variables. Unlike let and var, variables declared with const cannot be reassigned to a new value after they have been initialized.

```
const name = "PW Skills";
console.log(name); // PW Skills
```

```
const name = "College Wallah";
name = "PW Skills"; // TypeError: Assignment to constant variable.
```

However, the value they point to can still be mutated if it is a mutable data type, such as an object or an array.

```
const techStack = ["HTML", "CSS"];
console.log(techStack); // [ 'HTML', 'CSS' ]
techStack.push("JavaScript");
console.log(techStack); // [ 'HTML', 'CSS', 'JavaScript' ]
```

let, const does not allow redeclaration of variables within the same scope. Attempting to declare a const variable with the same name as an existing const variable within the same scope will result in a SyntaxError.

```
const name = "PW SKills";
const name = "College Wallah"; // SyntaxError: Identifier 'name' has already been declared.
```

Like, let, const declarations also have block scope. This means that they are only accessible within the block in which they are defined.

```
{
  const name = "PW Skills";
}
console.log(name); // ReferenceError: name is not defined
```

In JavaScript, const variables are not hoisted to the top of their scope like var variables. This means that you cannot access a const variable before it has been declared.

```
console.log(name); // ReferenceError: Cannot access 'name' before initialization
const name = "PW Skills";
```

const using function expressions are also not hoisted like var variables. This means that you cannot access a const variable that is declared using a function expression before it has been declared.

```
printName(); // ReferenceError: Cannot access 'printName' before initialization

const printName = function () {
  console.log("PW Skills");
};
```

Variables declared with const using arrow functions are also not hoisted like var variables. This means that you cannot access a const variable that is declared using an arrow function before it has been declared.

Temporal Dead Zone.

Temporal Dead Zone (TDZ) is a term used to describe the behavior of JavaScript's let and const declarations. This term came into existence in ES6 with the introduction of let and const to help developers write better code and make debugging easy, by telling them that they are accessing some data before defining it.

The TDZ is a period of time between the start of a block scope and the point at which a variable is declared. During this period, the variable exists but cannot be accessed or used.

From the above sections, we now know that let and const declarations have block scope, which means they are only accessible within the block in which they are defined. However, unlike var declarations, let and const declarations are not hoisted to the top of their scope. Instead, they remain in the TDZ until they are declared. Accessing a variable that is still in the TDZ will result in a ReferenceError.

TDZ errors occur because let/const declarations do hoist their declarations to the top of their scopes.

```
console.log(name); // Variable name is in TDZ
let name = "PW Skills";

// OUTPUT: ReferenceError: Cannot access 'name' before initialization.
```

One of the main advantages of the TDZ is that it helps prevent errors caused by accessing variables before they have been initialized. In traditional JavaScript, variables declared with var are hoisted to the top of their scope, which can sometimes result in code that unintentionally uses variables before they are ready. By contrast, the TDZ enforces a clear order of operations for variable declaration and initialization, making it easier to reason about the behavior of JavaScript code.

However, the TDZ can also be a disadvantage because it can make code more difficult to understand, especially for developers who are new to JavaScript. Because the TDZ is a subtle behavior that is not immediately obvious to the reader, code that relies heavily on let and const variables can be harder to read and debug. Additionally, some developers find that the TDZ makes it harder to write concise, readable code.

```
var printName = () => {
  console.log(name); // ReferenceError: Cannot access 'name' before initialization
  let name = "PW SKills";
};
printName();
```

The above example demonstrates how the Temporal Dead Zone (TDZ) can make code harder to read and write. In particular, the use of let to declare the variable name in the arrow function creates a TDZ for a name that lasts until it is assigned a value. As a result, trying to access the name before it is initialized will cause a ReferenceError to be thrown.

This can make the code harder to read and understand, especially for developers who are not familiar with the concept of TDZ. To make the code more concise and readable, some developers might choose to use a different variable declaration keyword, such as var, which does not create a TDZ. Alternatively, they might choose to declare the variable name outside of the function and pass it in as a parameter, which would also avoid the issue of the TDZ.

Another disadvantage of TDZ is that it can cause performance issues in some cases. Because the TDZ requires the JavaScript engine to perform extra work to enforce the correct order of variable declaration and initialization, it can sometimes slow down code execution. However, these performance issues are typically minor and are not a significant concern for most JavaScript developers.