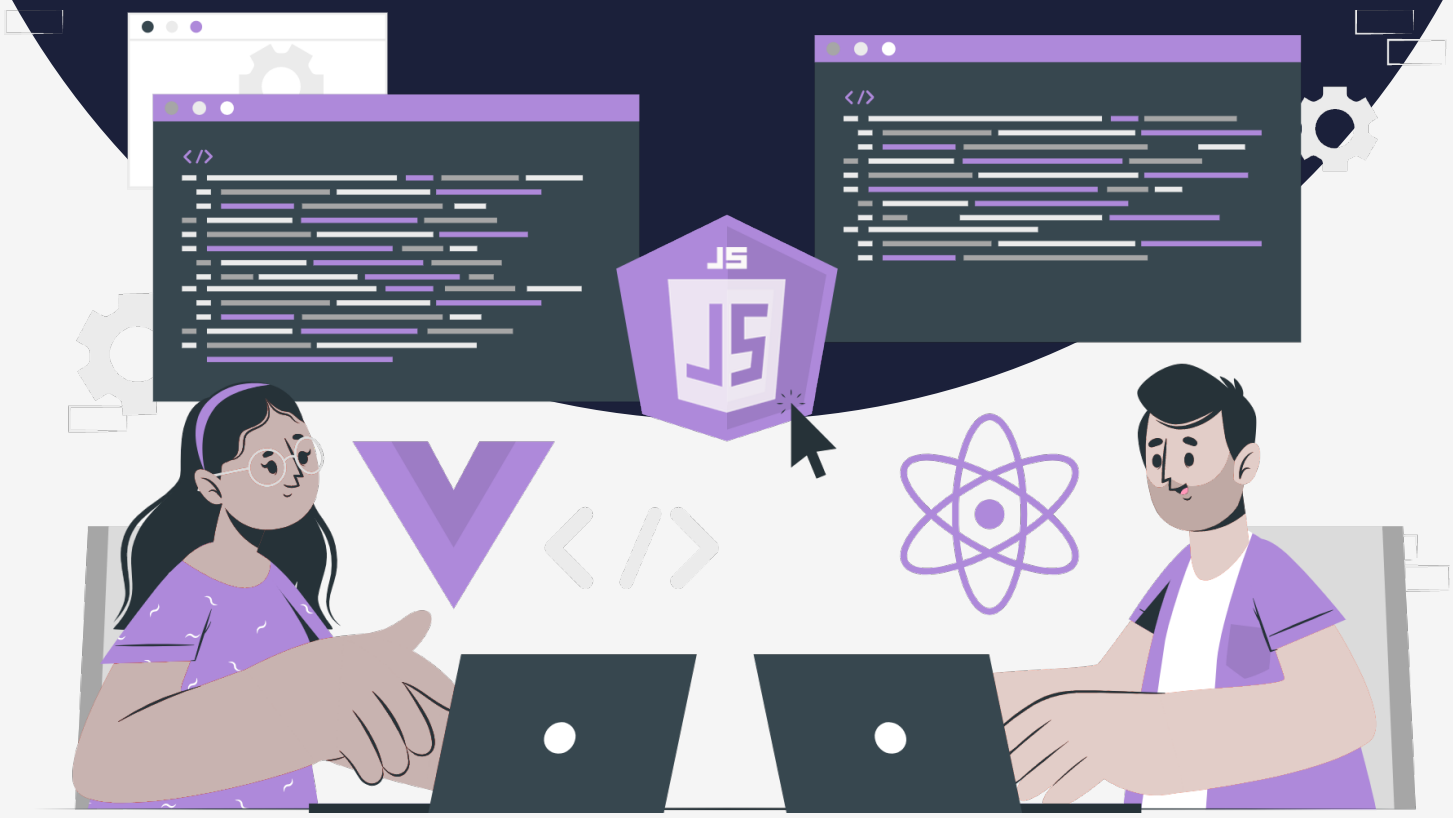
Lesson:

Working of function



Topics Covered:

1. Introduction.
2. Function Hoisting in Javascript.
3. Function Expressions.
4. Arrow functions.

Functions are really important in JavaScript programming. They let developers write code that can be used over and over again. Functions in JavaScript can be put in variables, passed as arguments to other functions, and received back as values from other functions. They make it easy to write complicated code by putting everything together and making it easy to understand and fix later.

As we have looked at in earlier lectures, functions can be created in various ways, such as using function declarations, function expressions, and arrow functions. Function declarations define a named function that can be called anywhere in the code, while function expressions create an anonymous function that is assigned to a variable. Arrow functions are a shorthand syntax for creating anonymous functions.

Functions can also take parameters, which are values passed to the function that can be used inside the function to perform specific tasks. By using functions in JavaScript, developers can create powerful, flexible code that can be easily reused and modified.

In this lecture let's look at the working of these functions.

Function Hoisting in Javascript.

In the previous lecture, we looked into variable hoisting. Function hoisting is similar to variable hoisting in JavaScript, as both involve moving the declaration of a variable or function to the top of its scope during the code's creation phase. It gives us the advantage that we can use a function before it is declared.

```
printMessage();

function printMessage() {
  console.log("Welcome to PW Skills");
}

// OUTPUT: Welcome to PW Skills
```

In the above code, we are calling the printMessage() function before it is actually defined in the code. However, since function declarations are hoisted in JavaScript, the function is moved to the top of its scope during the creation phase, allowing us to call it before it is defined.

So, when the code is executed, it will output "Welcome to PW Skills" to the console, since the printMessage() function has already been defined and is ready to be called.

This example demonstrates the power of function hoisting in JavaScript and how it can be used to write clean and organized code by allowing you to declare your functions in any order and still be able to call them from anywhere in the same scope.

How does JS Engine look at it?

When the JavaScript engine is given with the code, it first goes through a process called hoisting. During hoisting, the engine moves all function declarations to the top of their respective scopes, allowing you to call them before they are actually defined in the code.

In the code above, the printMessage() function is declared using the function keyword. So, during hoisting, the engine moves the entire printMessage() function declaration to the top of the code's scope. This means that by the time the code is executed, the function is already defined and can be called from anywhere in the same scope.

So, when the printMessage() function is called before its actual declaration, the JavaScript engine already knows that the function exists and can execute it without any errors. The engine simply looks for the function declaration at the top of its scope and executes the code inside it.

Function Expressions.

We all know that function expressions offer us a flexible and powerful way to define functions in JavaScript, allowing us to assign them to variables or pass them as arguments to other functions. But, function expressions, unlike function declarations, are not hoisted to the top of their respective scopes. This means that you cannot call a function expression before it is declared because it is not yet defined in the code.

```
console.log(printMessage());

var printMessage = function () {
  console.log("Welcome to PW Skills");
};

// OUTPUT: TypeError: printMessage is not a function
```

In the above code, we are attempting to call the function printMessage() before it is defined. This will result in a TypeError because printMessage is still undefined when we try to call it.

This is because the printMessage variable is being defined as a function expression, rather than a function declaration. In the creation phase, printMessage will be considered as a variable and given the value undefined When the execution phase begins, and the first line of the code is encountered, the JS engine sees that the variable printMessage is being called as a function, hence it throws an error saying add is not a function.

This shows that function expressions are not hoisted, so the variable printMessage will be undefined until the line of code that defines it is executed.

Arrow functions.

From the previous lectures, we know that Arrow functions are a concise way to write functions in JavaScript using the "=>" syntax. They are also a compact way of writing function expressions.

So do you think they can be hoisted? No, arrow functions are not hoisted in JavaScript.

```
console.log(printMessage());

var printMessage = () ⇒ {
  console.log("Welcome to PW Skills");
};

// OUTPUT: TypeError: printMessage is not a function
```

In the above code, we are attempting to call the arrow function printMessage() before it is defined. This will result in a TypeError because printMessage is still undefined when we try to call it.

Arrow functions, like other function expressions, are not hoisted in JavaScript. This means that the variable printMessage is still undefined when we try to call it with console.log().