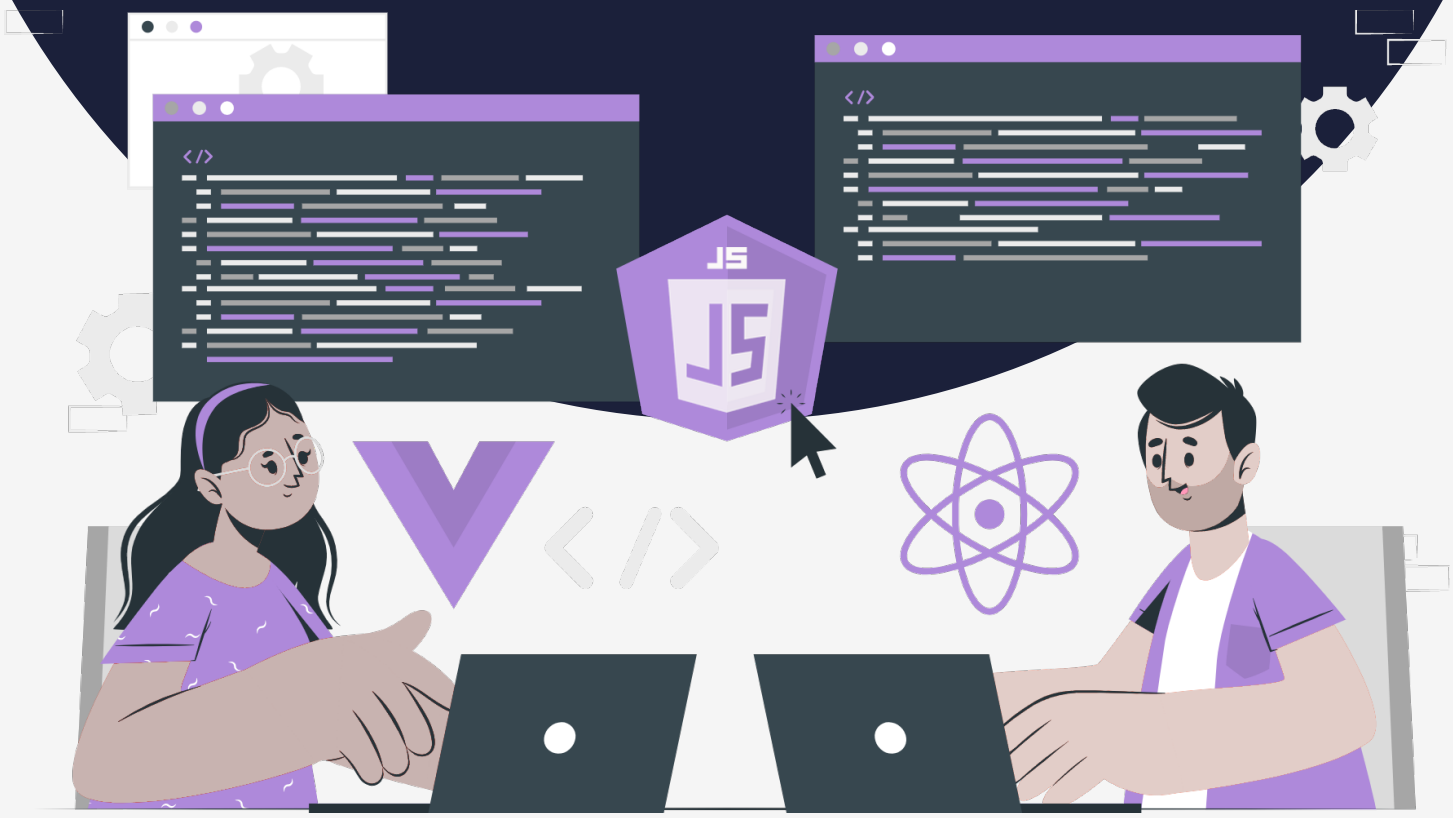
Lesson:

Hoisting



Topics Covered:

1. Introduction.
2. What is Hoisting?
3. How Hoisting Works?
4. Undefined vs ReferenceError.
5. Variable hoisting.

Have you ever written code in JavaScript and found that you can use a variable or function before it's actually declared? That's because of a nifty feature called hoisting! In simple terms, hoisting is when JavaScript moves all variable and function declarations to the top of their respective scopes before the code is executed.

This means that you can use variables and functions before they are actually declared in your code. It's a bit like the concept of "cheating" on a test - you can use the answer before you've actually written it down! But just like cheating on a test, relying too heavily on hoisting can lead to confusion and errors in your code.

So it's important to understand how hoisting works and use it effectively to make your code more readable and maintainable. In this lecture let's look at what hoisting is and master the art of hoisting and write better code in JavaScript.

What is Hoisting?

Before answering this question let's look at some code samples and their outputs.

```
var name = "PW Skills";

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

In the above code, the first line declares a variable called "name" and assigns it the value "PW Skills". The "var" keyword is used to indicate that we are declaring a variable.

The second line uses the "console.log" function to print the value of the "name" variable to the console. The value that is printed will be "PW Skills" since that's the value that was assigned to the variable.

```
console.log(name);

var name = "PW Skills";

// Output: undefined
```

In the above code, we are trying to access the name variable before it is declared, so it should give us an error, but why is it printing undefined? Let us try removing the declaration and see what happens.

```
console.log(name);

// Output: ReferenceError: name is not defined
```

The output to the console will be a "ReferenceError". This is because the variable "name" has not been declared or assigned a value, so it does not exist in the current scope.

When the code is executed, the "console.log" function is called and attempts to print the value of the "name" variable. However, since the variable has not been declared, a "ReferenceError" is thrown.

This behavior is different from the previous example, where the variable was declared but was called before the declaration. In this case, the variable has not been declared at all, so it cannot be used in the code. This is an important concept to understand in JavaScript, as it can help you avoid common errors and bugs in your code.

Now, there is no issue with the third example as it's clear that the variable is not declared if the error is raised. But, in the second example, the variable is declared but is called beforehand. This returned undefined. Let's look at this.

In simple terms, we can say that in the second example, the code actually looks like the following to the javascript engine.

```
var name;
console.log(name);
name = "PW Skills";
// Output: undefined
```

Here's what happens when the code is executed:

1. The variable "name" is declared using the "var" keyword, but it has not yet been assigned a value.
2. The "console.log" function is called, attempting to print the value of the "name" variable.
3. The value of the "name" variable is currently "undefined" since it has not been assigned a value yet.
4. The variable "name" is then assigned the value "PW Skills".
5. The code has finished executing, and the output to the console is "undefined".

So even though the "name" variable is declared before the "console.log" function, its value has not yet been assigned when the "console.log" function is called. This is why the output to the console is "undefined".

Now, let's define hoisting.

The javascript mechanism in which variables and function declarations are moved to the top of their scope before execution of the code is called Hoisting.

In simple words, Hoisting is a mechanism that makes it seem like variables and function declarations are moved to the top of the scope and lets us access variables and functions even before they are declared.

Understanding hoisting in JavaScript is important for writing clean, efficient code that avoids common errors and bugs. By understanding how hoisting works, you can ensure that your code is structured in a way that is easy to read and maintain.

How Hoisting Works

Javascript Engine executes code in two phases: The creation phase and the execution phase. In the creation phase, the variables are registered in the scope; this is what makes Hoisting possible, while the execution phase comes after the creation phase, where the JS engine executes the code line by line.

In the first phase, called the "creation" phase, the JavaScript engine scans the code and identifies all variable and function declarations. It then creates space in memory for these variables and functions but does not yet assign any values or execute any code. This is where hoisting occurs, as variable and function declarations are moved to the top of their respective scopes.

In the second phase, called the "execution" phase, the JavaScript engine assigns values to variables and executes the code. This is where the actual code is run and where variables and functions are used in calculations or other operations.

It's important to note that only the declarations themselves are hoisted, not the assignments or initializations. So if a variable is declared but not assigned a value until later in the code, its value will still be "undefined" until it is assigned a value.

Undefined vs ReferenceError

In JavaScript, "undefined" is a primitive data type that represents a variable that has been declared but has not been assigned a value. It is also used to indicate that a function has no return value.

In hoisting, the declarations themselves are hoisted, not the assignments or initializations. This means that if a variable is declared but not assigned a value until later in the code, its value will be "undefined" until it is assigned a value.

On the other hand, a "ReferenceError" is a type of error that occurs when you try to reference a variable or function that has not been declared. This can happen when you misspell a variable name, or when you try to use a variable that is declared in a different scope. We have seen this in 3rd example.

Hoisting is applicable to variable and function declarations in JavaScript. In this lecture, we will be looking at variable hoisting and in the next lecture, we will look at the hoisting of function declarations.

Variable hoisting

Variable hoisting is a mechanism in JavaScript that moves variable declarations to the top of their respective scopes before the code is executed. This means that you can declare a variable anywhere in your code, and it will still be recognized as if it were declared at the top of the scope.

```
console.log(name); // undefined
var name = "PW Skills"; // Variable name is initialised and declared.
console.log(name); // PW Skills
```

The above code is visualized by the js engine as

```
var name; // Variable "name" is initialized: memory is allocated with the value undefined.
console.log(name); // undefined
name = "PW Skills"; // Variable "name" is assigned with a value "PW Skills".
console.log(name); // PW Skills
```

The above code is executed as

1. In the creation phase, the name variable is initialized, which means the variable name is allocated with memory and given the value undefined.
2. When the execution phase begins and the second line is encountered undefined is printed to the console.
3. Then the variable is assigned a value in the third line and the name variable has the value "PW Skills".
4. The same is printed to the console when the fourth line is executed.

From the above example, we can simply say, variable Hoisting is a mechanism that makes it seem like variables declarations are moved to the top of the code and lets us access variables before they are declared.

We will look at function hoisting in the next session.