





















[SOUND] Okay, so

in this lecture we're going to talk about defining variables,

functions, and where these artifacts live,

otherwise known as scope. Now it's fairly straight

forward you define a variable. Variable definitions should always

start with the keyword var. Now notice that no types

are declared in JavaScript. And the reason that is, is because JavaScript is called

a dynamically typed language. Now what that means is

that the JavaScript engine figures out the type of

a particular variable at runtime. And it also means that the same variable

can hold different types during the life of the execution of the program. So a variable can start out to be

a string, then change to a number, then change to an object, and so on,

and all of that is perfectly legal. So that's as far as variable definitions. Now the way you define a function

is by using the keyword function followed by function name and

followed parentheses then curly braces. Now in JavaScript, when you define

a function name, it's very, very similar to defining a variable

whose value is a function. So for example, you could define

a function in JavaScript by either using the keyword function followed

by the function name. Or you could create a variable and

set it equal to a function. Now when you do it the second way, no name

is defined after the function keyword. However, either syntax let's you

refer to this function as a. Now notice that the value of

the function is assigned to a, not the returned result of

the execution of that function. So after either of these

definitions you can now invoke, otherwise known as execute, function

by referring to it by it's name a. Now once you define the function, either

you define the function the first way, that we talked about, or the second way where you store

the function inside of a variable. The way you invoke a function is by

taking the name, or the reference, to that function and

putting the parenthesis afterwards. So that is called a function execution,

or function invocation. The word execute and invoke are synonyms,

so you could really use either one. But in programming literature it's very

often that the execution of a function is referred to as invocation

of that function. Now you can also define arguments to

be passed to the JavaScript function. Now the defined arguments

are defined without the keyword var. In fact, if you place the key word var

in the function argument definition, that will be a syntax error. If you want your function to return

some value, you use the return keyword followed by whatever value it

is that you want to return. If the return keyword is

not followed by a value, what you're signaling to the JavaScript

engine is that it should terminate the function and

exit out of it without returning anything. Now the way you pass the arguments

to the JavaScript function is also kind of unique. So in the first line of this example here,

we have the function compare that we saw in the previous slide, and

it's defined here with arguments x and y. In the second line, we're executing or

invoking function compare and passing to it two arguments, 4 and 5. Now the result of the execution of that

function is then stored in the variable a. On the third line, you notice that we're

still executing the compare function but we're not storing the result anywhere,

and that is perfectly legal. And it's also legal since JavaScript

is a dynamically typed language to pass a as a second argument,

as a string, as opposed to a number. Now that may not make sense as far as the

logic to the function is concerned, but it's certainly perfectly legal syntax. Now the last line demonstrates yet

another rule about JavaScript functions. All arguments defined in

a JavaScript function are optional. You can call that same compare function

without any arguments at all, and it all will be perfectly legal. Again, it may not make sense

as far as the business logic of the execution of the compare function. But the syntax and the invocation of

this function is perfectly legal. Now each function or

variable lives within a particular scope. In JavaScript there's really two scopes,

global scope and function, or

otherwise known as lexical scope. Now lexical here means that it depends on

where something is physically defined. For example, a variable that's

defined within a function is physically defined

within that function. There's no block scope in JavaScript. In other languages, curly braces

themselves signify a new scope. In JavaScript,

functions signify a new scope. That's all you really need to

know about functions for now, and we're going to talk about functions

in much more detail in a later lecture. Now variables and functions defined in

a global scope are available everywhere. Meaning any other functions that

are defined in a global scope and so on can get access to these globally

defined functions and variables. As far as function and

lexical scope, variables and functions defined here are available

only within this function. And you heard it right, you can define

functions within other functions. And those newly defined functions, have a way to get at its outer

function variables and functions. Which leads us to the next topic,

the scope chain. Now in order to understand how

scope chain works in JavaScript, you have to understand how

a JavaScript engine executes things. Everything in JavaScript is

executed in an execution context. Now a function invocation

creates a new execution context within which that function is executed. Now each execution context

has the following. Number one,

it has its own variable environment. This is where it stores its newly

defined functions and variables. It also gets the special this object, which we're going to talk

about at a later lecture. And what's most important for our discussion right now, it gets

a reference to its outer environment. Meaning the execution environment within

which this particular function is defined. Now, global scope does not

have an outer environment simply because it's

the most outer there is. So here's how scope chain works. Referenced, notice referenced, not defined, variable will be searched for

in its current scope first. If not found,

the outer reference will be searched. If not found there, the outer reference's

outer reference will be searched and etc., and so on. If not found, the outer reference's outer

reference will be searched, and etc., and so on. Now this will keep going

until the global scope. If not found in global scope,

the variable is going to be undefined. Now we'll discuss the special value

undefined when we speak about JavaScript built in types. But for now, just know that the variable

undefined means it's just not found. Now in order to understand this

scope chain a little bit better, let's take a look at

the following visualization. Suppose you executed a couple of lines

of code inside the global scope. The first line defines variable x and

assigns a value of 2 to the variable x and the second line executes function A. Now function A itself is

defined in global scope and it has its own variable

x that is set to 5. Now then,

within function A you call function B. Function B is also defined

in the global scope. And all it does is just print out

the to the console the value of x. Now what will be printed

out to the console? Will it be x = 5 because B is

invoked inside function A? Or it will be x = 2 from the global scope? And the answer is that the result that

will be x = 2 from the global scope. Now why is that? Well, if you remember where we just said,

the reason's pretty simple. Even though B is called within A and

A has its own x, what actually matters as far as resolving where x is

coming from is its outer reference. And B is defined within the global scope. Therefore, the outer reference of function

B is the global scope, not function A. What this means is that it does not matter

where a particular function is executed. What matters as far as resolving

scope of a variable is concerned is where that function

is physically defined. Function B is physically defined

inside the global scope. Therefore a reference of x inside function

B Is going to look inside function B first. If it doesn't find it it's going to

use its outer reference to look for it there and

its outer reference is the global scope. Let's jump to the code editor and

see an example of this.