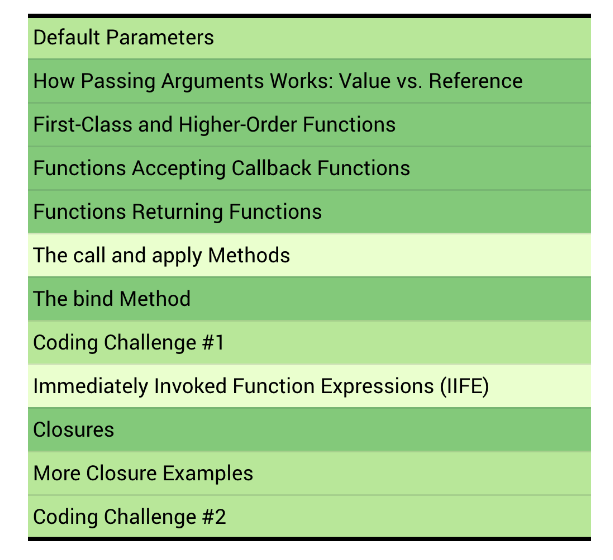
126. Section Intro

Function are one of the most fundamental building blocks of any JavaScript program.

So, in this section we’re going to study some quite advanced topics, such as higher order function to bind method also closures.

127. Section Roadmap



128. Default Parameters

Default parameter set function.

129. How Passing Arguments Works: Value vs. Reference

In programming there are two terms that are used all the time when dealing with functions, which is passing by value, and passing by reference.

Have some confusion between these terms and how it works in JavaScript.

So, JavaScript does not have passing by reference, only passing by value, even though it looks like it’s passing by reference. JavaScript dose not have pass by reference. For objects, we do in fact pass in reference. So, the memory address of the object. However, that reference itself is still a value. It’s simply a value that contains a memory address. So, basically, we pass a reference to the function, but we do not pass by reference, and this is an important distinction.

130. First-Class and Higher-Order Functions

A fundamental property of the JavaScript language. Which is the fact that it has first class functions. This enables us to write higher order functions.

But what's that all about?

Well, let's see. So, JavaScript is a language that has first class functions which in technical terms means that functions are so-called first citizens. In practice, that means that functions are simply treated as values.

Now, why does JavaScript work this way?

Well, it's simply because functions are really just another type of objects in JavaScript. And since objects are values, functions are values too. And since functions are values, there is a bunch of interesting things that we can do with them, like storing them in variables or object properties.

So, the function values here are marked in red, and then you see, we create a function expression in the first example and an object method in the second example. So, the value in the red rectangle is the function value itself, that we can store wherever we like. We can also pass functions as arguments to other functions. Finally, remember that functions are objects. And many types of objects in JavaScript have methods, right? Like array methods, for example. And actually, there are also function methods. So, methods that we can call on functions.

This bind method here is an example of that. And again, we will learn about the bind method as we go through the section. All right. Now the fact that JavaScript has first-class functions makes it possible for us to use and write higher order functions. So, a higher order function is either a function that receives another function as an argument, or a function that returns a new function. So, let's check out both types here. First, for functions that receive another function. We have the same example as before. So here, the add event listener function is the higher order function. And why? Well, because it receives another function as an input. In this case, the greet function. And we usually say that the function that is passed in is a callback function. Sure! Here are the sentences from the given transcript: that's because the callback function will be called later. It's like the greet function saying, "hey there, don't greet me yet, but call me back once you're ready. "And this works, not only in the context of the add event listener function, but in many other use cases as well. Okay, second, we can also have functions that return another function. So, we have the higher order function here. So basically, this whole code block, which clearly returns a new function, which is this one. And this style of functions is also used a lot in JavaScript. But it's also more advanced, and I guess harder to understand. I will show it to you in the next lecture, but we will explore this deeper a bit later. Now, just to finish, there seems to be some confusion between first-class functions and higher order functions. Some people think that they are the same thing, but actually they mean different things. So, first-class functions are just a feature that a programming language either has or does not have. All it means is that all functions are values. There are no first-class functions in practice, okay? It's just a concept. There are, however, higher order functions in practice, which are possible because the language supports first-class functions. So, it's a subtle difference, but still worth noting if you want to be able to talk like a true JavaScript master. Great. Now in the next lecture, let's actually create our own higher order functions. I have removed the line numbers and provided you with the sentences from the transcript. Let me know if there's anything else I can help you with!

131. Functions Accepting Callback Functions

Callback function allow us to create abstraction. Abstract is something really important in programming. So basically, what abstract and means, is that we hide the detail of some code implementation because we don’t really care about all that detail. And this allows us to think about problems at a higher more abstract level. And that’s why it’s called an obstruction.

132. Functions Returning Functions

133. The call and apply Methods

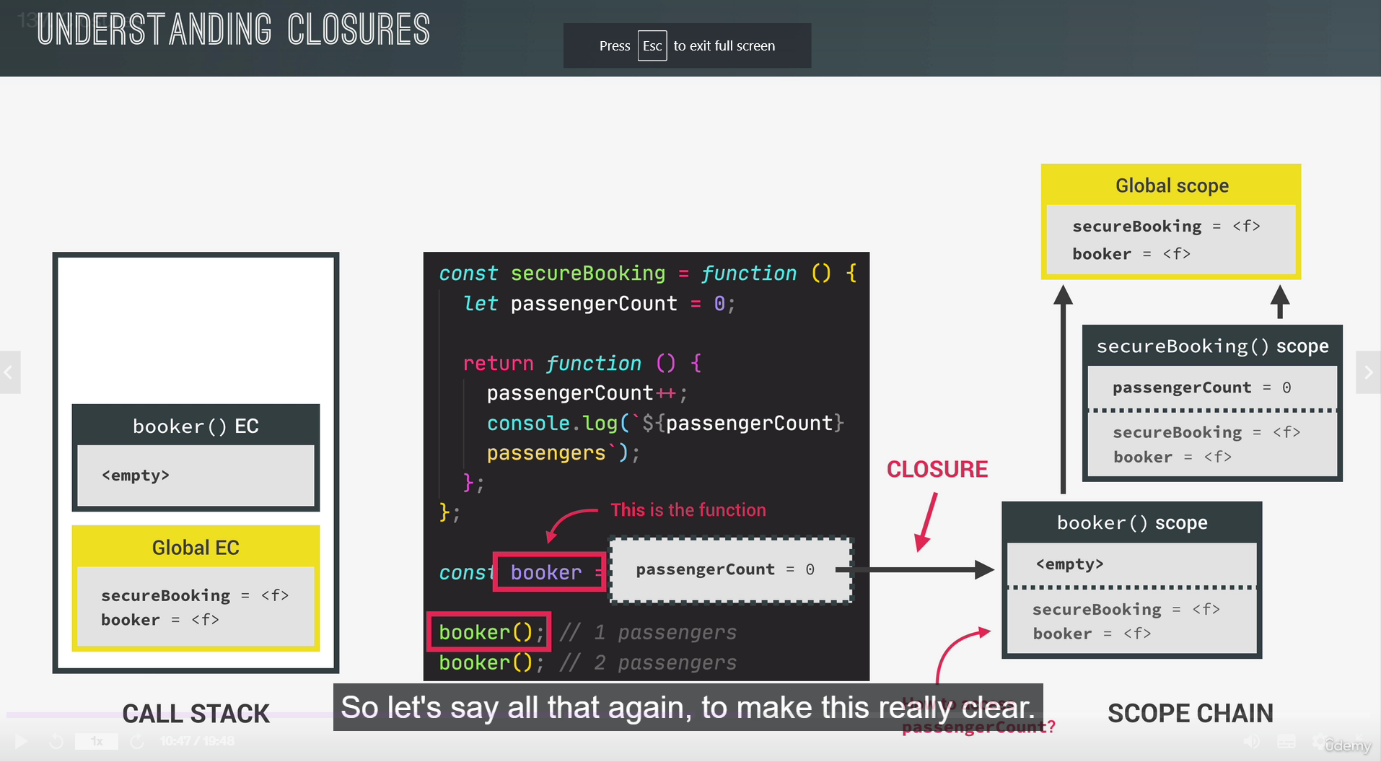
134. The bind Method

135. Coding Challenge #1

136. Immediately Invoked Function Expressions

137. Closures

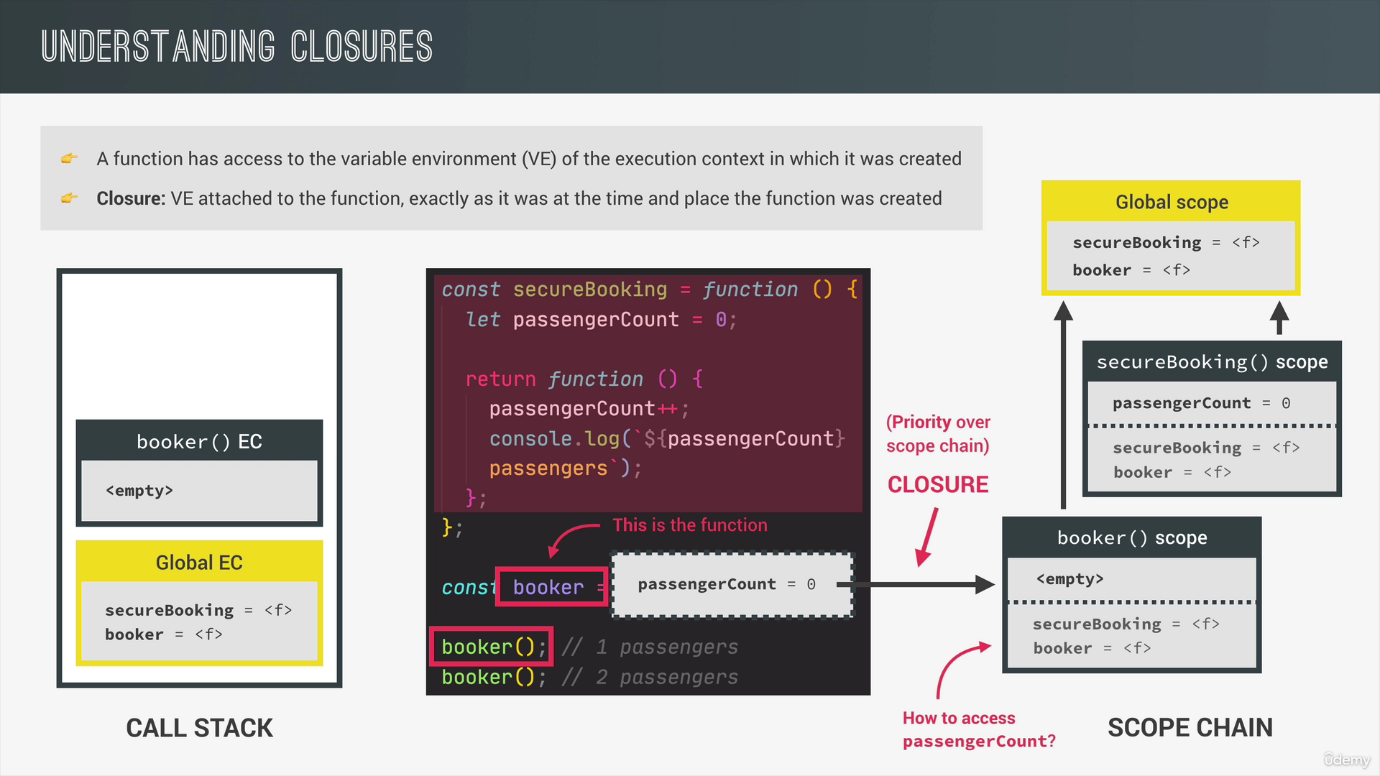
Note: Any function always has access to the variable environment of the execution context in which the function was created. Now in the case of booker, this function was created. It was born in the execution context of secure booking, which was pooped off the stack previously, remember? So, there for the booker function will get access to this variable environment. Which contain passengerCount variable. And this is how the function will be able to read and manipulate the passenger count variable. And so, it’s a connection this connection that we call closure.

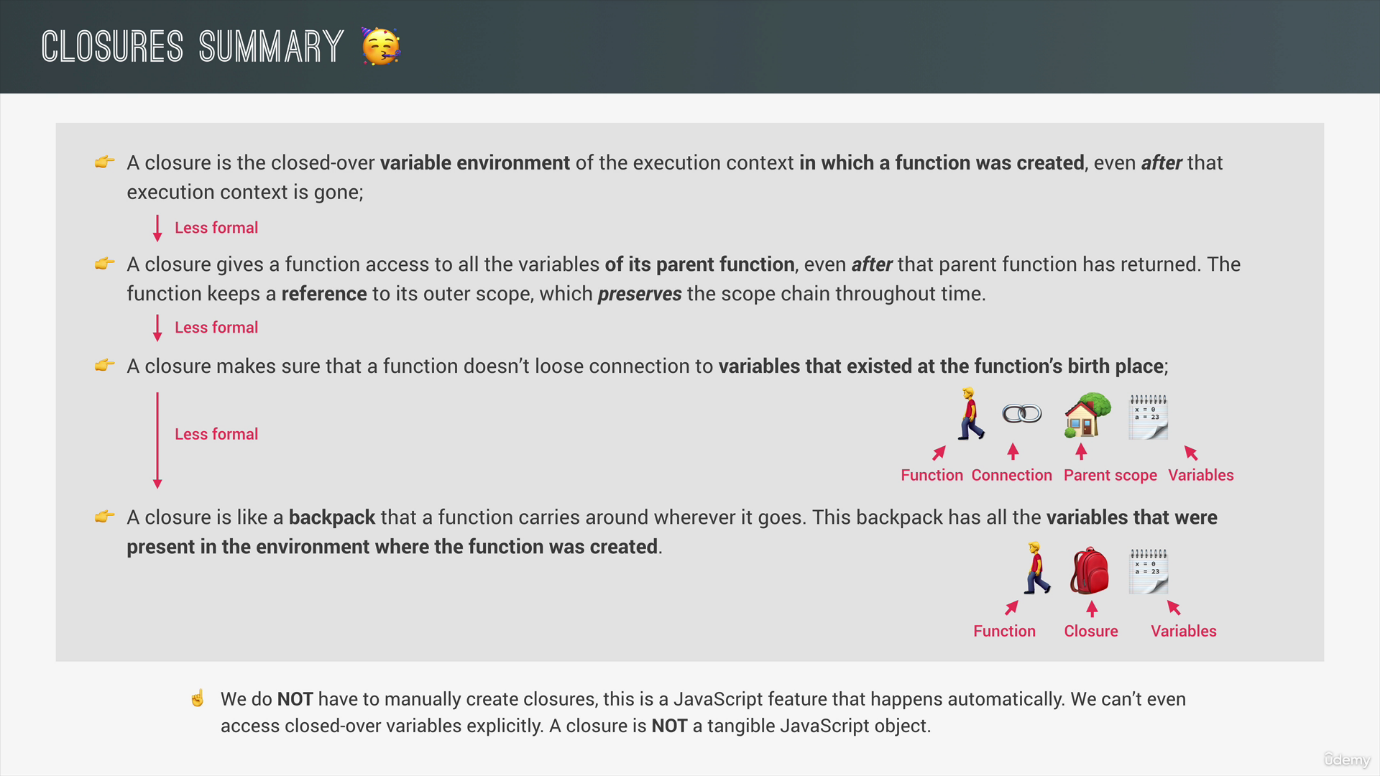


So, a function always has access to the variable environment of the execution context in which it was created, even after a debt execution context is gone. The closure is then basically this variable environment attached to the function. Exactly as it was at the time and place that the function was created.

The booker function has access to the passengerCount variable. Because it’s basically defined in the scope in which the booker function was actually created. So, in a sense, the scope chain is actually preserved through, even when a scope has already been destroyed.

This means that even though the execution context has actually been destroyed, the variable environment somehow keeps living somewhere in the engine.





////////////another example

Closures are an important concept in JavaScript that allow functions to retain access to variables from their outer (enclosing) lexical environment, even after the outer function has finished executing. In simple terms, a closure is a function bundled together with its surrounding state (lexical environment).

Here's a breakdown of closures and how they work:

1. Lexical Scoping:

- JavaScript has lexical scoping, which means that functions are executed using the variable scope chain that was in place when they were defined.

- Each time a function is invoked, it creates a new execution context that has access to its own local variables as well as variables from its outer lexical environment.

2. Closure Definition:

- A closure is created when a nested function (inner function) has access to variables from its outer function, even after the outer function has finished executing.

- The inner function "closes over" the variables from the outer function, hence the name "closure".

3. Retaining Access to Outer Variables:

- When an inner function references a variable, the JavaScript engine looks for that variable in the current function's scope. If it's not found, it continues searching through the outer function's scope chain until the variable is found or the global scope is reached.

- If the inner function is returned or passed as a callback to another function, it carries with it a reference to its outer variables, creating a closure.

Closures are commonly used in scenarios such as:

- Data Privacy: Closures allow you to create private variables and functions that are inaccessible to the outside world.

- Encapsulation: Closures enable you to encapsulate state and behavior within a function, making it self-contained.

- Function Factories: Closures help in creating functions with pre-configured settings or parameters.

- Asynchronous Operations: Closures are useful when working with asynchronous code, as they allow variables to persist across asynchronous callbacks.

Here's an example to illustrate closures:

*function* outer() {

*var* outerVariable = 'I am from outer';

*function* inner() {

    console.log(outerVariable);

  }

  return inner;

}

*var* closureFunction = outer();

closureFunction(); *// Output: "I am from outer"*

In this example, the `outer` function creates a closure by returning the `inner` function. Even after the `outer` function has finished executing, the `closureFunction` still has access to the `outerVariable` due to the closure. When `closureFunction` is invoked, it logs the value of `outerVariable`.

Closures are a powerful feature in JavaScript that enable advanced programming techniques. They provide a way to maintain and manipulate state within functions, leading to more flexible and modular code.

138. More Closure Examples

139. Coding Challenge #2