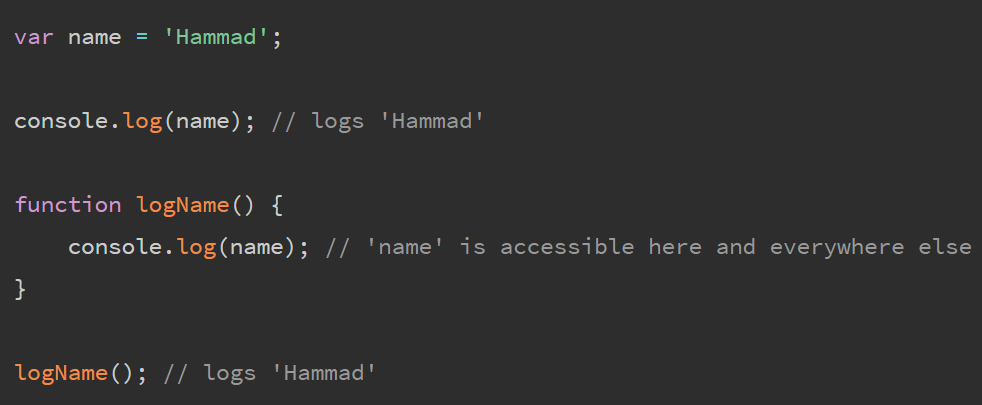
Scope is the accessibility of variables, functions, and objects in some parts of your code during runtime. In other words, scope determines the visibility of variables and other resources in areas of your code.

The advantage of the scope is that it provides some level of security to your code. Scope also solves the naming problem when you have variables with the same name but in different scopes.

In the JavaScript language there are two types of scopes:

* Global Scope
* Local Scope

Variables inside the Global scope can be accessed and altered in any other scope.



Variables defined inside a function or any other block of code are in the local scope. This means that variables having the same name can be used in different functions.

*Lexical scope* means that in a nested group of functions, **the inner functions have access to the variables and other resources of their parent scope**. This means that the child functions are lexically bound to the execution context of their parents. Lexical scope is sometimes also referred to as *static scope*.

the *least exposure* principle (POLE) encourages us to use block (and function) scoping to limit the scope exposure of variables.

Closure builds on this approach: for variables we need to use over time, we can encapsulate them and still preserve access to them from inside functions. Functions *remember* these referenced scoped variables via closure.

Closure is one of the most important language characteristics ever invented in programming—it underlies major programming paradigms, including Functional Programming (FP), modules, and even a bit of class-oriented design.

**Closure is a behavior of function**s and only functions. An object cannot have closure, nor does a class have closure (though its functions/methods might). Only functions have closure.

**For closure to be observed, a function must be invoked in a different branch of the scope chain from where it was originally defined**. A function executing in the same scope it was defined cannot be considered as closure;

**Example in code - slide closure in action**

**greetStudent(..) makes reference to both students and studentID**, identifiers which come from the enclosing scope of lookupStudent(..). **Each of those references from the inner function to the variable in an outer scope is called a *closure***.

At a later time when either instance of the greetStudent(..) function is invoked, those variables are still there, holding their current values.

**Adding Up Closures**

**Every time the outer adder(..) function runs, a *new* inner addTo(..) function instance is created, and for each new instance, a new closure**.

**By closing over a variable in a function, we can keep using that variable as long as that function reference exists in the program, and from anywhere we want to invoke that function.** This is why closure is such a powerful technique used widely across so many areas of programming!



**An Alternative Perspective**

**Closure is the link-association that connects that function to the scope/variables outside of itself, no matter where that function goes.**

**Closure preserves a hidden link back to the original scope to facilitate the access to the closed-over variables.**



Each call to adder(..) still creates a new BLUE(2) scope containing a num1 variable, as well as an instance of the GREEN(3) addTo(..) scope. But what's different from Figure 4 is, now these GREEN(3) instances remain in place, naturally nested inside of their BLUE(2) scope instances. The addTo10 and addTo42 references are moved to the RED(1) outer scope, not the function instances themselves.

When addTo10(15) is called, the addTo(..) function instance (still in place in its original BLUE(2) scope environment) is invoked. Since the function instance itself never moved, of course it still has natural access to its scope chain.

**The previous model (Figure 4) is not *wrong* at describing closure in JS. It's an academic perspective on closure. By contrast, the alternative model (Figure 5) could be described as a bit more implementation focused, how JS actually works.**

Both models are useful in understanding closure. Whichever you choose, the observable outcomes in our program are the same.