Variable Scope

Ian Harris

1 Scope of a Variable

So, we're going to talk about what the scope of a variable is. Roughly, /the scope of a variable is that the places in the code where a variable can be accessed./ So, *variable scope* defines how a variable reference is resolved in the code. So, if you reference a variable x, how does the program figure out which variable actually talking about. That's basically what variable scope is.

1.1 Examples of Variable Scope

So, these little examples just to show you an example of scope. If we look at the first block of code, we got this variable x, I'm highlighting in red. This variable x is defined outside of these two functions. So, I've got these two functions defined. Function f and function g. But outside of both of them, I define this variable x. So, var x equals to, wherever it is, one. Then, I defined my function f and function g. Now, if you look inside function f and g, the're very simple. All they doing is they're printing x, they're both printing x, is all they doing. So, inside function f and inside function g, when you call these functions, they've got to figure out where to find the value for x. Now, in this case, they're going to find the value that I've defined outside. So, where I say var x equals one, when they print out x, they going to print out one for x. Because of the scoping rules allow that. So basically, because it define outside of either one of these functions, then both of them will have access to it. We'll define the formal rules for that in a few slides.

Now, the next block of code, you get a problem. Because the next block of code again, you've got function f and function g, but this time variable x is defined inside function f, but not inside function g. It's only inside f. So, function f will print properly because it'll look at x and resolve it since it's defined locally, right there inside, it's functional. So, x is equal to one, and it'll be happy. But function g will have no reference to x. It won't be able to see that and it'll throw an error when you try to run that, because they won't know what x is.

1.2 How Does the Compiler Resolve Variable Scope

So, this is the type of problem that needs to be resolved, you don't want to run into this type of problem. You want to be able to know where your variables get resolved to. So that you don't have problems like this. So, we're going to talk about variable scoping right now. So, how does the compiler figure out where a variable reference should be resolved to? Which x are you talking about? It's this x versus that x.

1.2.1 Variable Scope is Resolved Using Blocks

So, in Go, variable scoping is done using blocks. Now, block is a sequence of declarations and statements within matching curly brackets.

Explicit Blocks So, those curly brackets right there, you have an open curly bracket, closed curly bracket, everything in between is called a block. So, that's how you explicitly define blocks. You notice how these blocks can be hierarchical. You can have curly brackets and then within that you can have some other curly brackets and within that you can have some more. So, you can have this hierarchy of blocks. These are explicit blocks. When you put the curly brackets in your own code, those are explicit blocks that you, as a programmer included. Our function definitions notice, are defined by curly brackets. We haven't gotten to functions, we'll talk about it more later. But every function definition, you define the function, gives the name of the function, and then you have open curly brackets, closed curly brackets. Every function, you got curly brackets. So, there's a hierarchy of these curly brackets and hierarchy of these blocks.

Implicit Block Hierarchy Now, also there implicit blocks inside this hierarchy. So, there are blocks that are implicitly defined without the curly brackets.

Universe Block. So, just to list those blocks, first is the universe block. That's all Go source code, that is the biggest block, the universe block.

Package Block There's a package block. So, every package, you don't put curly brackets around every package at all the source code in a particular package, that's all within one block, which is inside the universe block.

File Block Then, there's file block. File block, all the source code in a single file is within the file block, and remember that package can be composed of many files. So, there can be one package block that has many file block if you got a lot of files inside the package.

Statement Blocks Now, then other implicit blocks include the if statement, for statement, and switch statements. All these have curly brackets that define their own blocks. Also, the clauses inside a switch or select. We'll get to these in more detail later.

But these are all, all the ones I'm listing here are these implicit blocks that you don't have to put explicit curly brackets for. Well, so you can. So, for instance, an if statement, you can use it curly brackets too. But like the universe block, the package block, file block, these are all implicit blocks and there's a hierarchy of these. So, this is my point, that there's this hierarchy of these blocks and each block can have its own environment variables associated with it.

1.3 Lexical Scoping

So, lexical scoping. This defines how variable references are resolved. So, Go is a lexically scoped language using blocks. So, when we talk about lexical scoping, we've got to talk about this relationship of one block being defined inside another block. So, I'm using this terminology here, I'm saying bi is greater than/equal to bj. If bj, b is a block. If bj is defined inside bi, then bi is greater than or equal to bj. So, I would say that bj, if it's defined inside bi, bj would be, I refer to it as an inner scope, where the outer scope that includes bj is bi.

1.3.1 Example of Lexical Scoping

So, just as an example of this, it's a transitive relationship, but just as an example of this, look at the code. We got, we've seen this code before. Got variable x, we'd initialize it to one. Then, you got a function f and function g. Now, if we look at the blocks inside here, first you got, this is all in one file. So, all of this is inside the file block and I'm calling that b1. So, b1 is my file block and everything is inside the file block. But in addition to the file block, I'm defining two function, f and g. Each one of these function blocks, the functions gets its own function block, so b2 is the function block for f and b3 is the function block for g. So, if I were to look at how these blocks are related, a b2 and b3 are both defined inside b1. So, I say b1 is greater than b2, and b1 is greater than b3 by my definition, because b2 and b3 are defined inside b1. But notice that there's no relationship between b2 and b3. Because they're not defined within each other. So, we need to know this, because the scoping rules, when you're resolving a variable, you go to the greater including scope.

So, for instance, if inside b1 the function f and inside of that block, you're referring to variable x. Then, it's going to look for that variable x inside b2 itself, inside its local block. But then, it looks for the next bigger block that is defined inside. So, it starts in b2, looks for the block, it looks for variable x. If it doesn't see it there, which is not defined there, in this case, then it says, "Okay what is the next bigger block that I've defined inside?" It would get b1, and when it look inside b1 and say, "Ah, there is a definition of x here." That's the definition of x that it would use. Same goes for b3. If you look at the function g, it uses variable x, it access variable x. First it looks inside its local block b3, it doesn't see it. So, it looks inside the next bigger block that's defined inside, b1 and it sees it there. So, that's why this works, this why this code will work. The x will be resolved properly, so that x equals one, that variable that we defined inside b1.

Variable Declarations and Scope So, when you're talking about scope variables, a variable is accessible from a block bj, if the variables declared in some block bi, and block bi is greater than/equal to bj. So, it's either the variables you declared right there in bj or it's declared in some outer block that's greater than bj. So, that's why in the first block of code, first block of the first step code sequence, you can see where x is defined inside the file block. Both of those functions, which are also inside the same file block, they can both properly access the x variable because their function blocks are within the file block. But in the next block of code, next sequence of code, the x is defined inside the function block of f. But it's not inside the function block of g. So, when g tries to reference x, the variable x, it doesn't see it in its local block. It also doesn't see it in its file block, because now the definition is inside the function block for function f. So, that's why this fails, x doesn't get resolved back to anything and there's an error.