Variables

This lesson discusses how variables are used in Go.

WE'LL COVER THE FOLLOWING

Introduction

A value that can be changed by a program during execution is called a **variable**. The general form for declaring a variable uses the keyword **var** as:

```
var identifier type
```

Here, `identifier` is the name of the variable, and `type` is the type of the variable. As discussed earlier in this chapter, `type` is written *after* the `identifier` of the variable, contrary to most older programming languages. When a variable is declared, *memory in Go is initialized*, which means it contains the default zero or null value depending upon its type automatically. For example, 0 for int, 0.0 for float, false for bool, empty string ("") for string, nil for pointer, zero-ed struct, and so on.

Run the following program to see how declaring a variable works.

```
package main
import "fmt"

func main(){
    var number int          // Declaring  an integer variable
    fmt.Println(number)     // Printing its value
    var decision bool       // Declaring a boolean variable
    fmt.Println(decision)   // Printing its value
}
```

Declaring a Variable

You can see that in the above code, we declare a variable `number` of type `int` at **line 5**. As memory is initialized, the default value for `number` is printed at **line 6**, which is **0**. Similarly, a variable `decision` of type `bool` at **line** 7 is declared, and **false** is printed as its value at **line 8**.

> **Remark**: The naming of identifiers for variables follows the *camelCasing* rules (start with a small letter, and every new part of the word starts with a capital letter). But if the variable has to be exported, it must start with a capital letter, as discussed earlier in this chapter.

Assigning values

Giving a value to a variable is called *assigning a value*. A variable is assigned a value using the assignment operator(`=`) at compile time. But of course, a value can also be computed or changed during runtime. Declaration and assignment (initialization) can be combined in the general format:

```
var identifier type = value
```

Here, `value` can be a value of type `type`, or can even be a variable of type `type` or can also be an expression.

Run the following program to see how the assignment operator works on variables.

```
package main
import "fmt"

func main(){
    var number int = 5          // Declaring and initializing  an integer variable
    fmt.Println(number)          // Printing its value
    var decision bool = true     // Declaring a initializing a boolean variable
    fmt.Println(decision)        // Printing its value
}
```

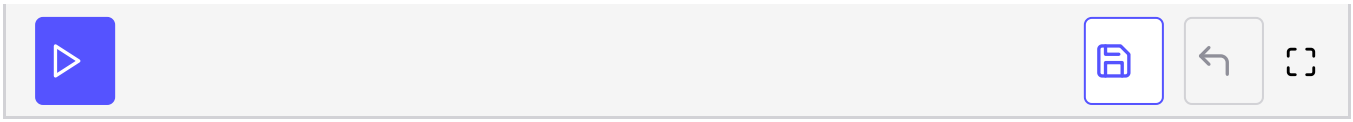
Declaring & Initializing a Variable

You can see that in the above code, we declared a variable `number` of type `int` at **line 5**, and initialized it with the value of **5**. Similarly, a variable `decision` of type `bool` at **line 7** was declared and initialized with **true**. These initialized values are printed later on by **line 6** and **line 8** respectively.

Go-compiler is intelligent enough to derive the type of a variable from its value dynamically, also called *automatic type inference* at runtime, so omitting the type of a variable is also a correct syntax. Let's see a program on automatic type inference.

```
package main
import "fmt"

func main(){
    var number = 5          // Declaring and initializing an integer variable without stating
    fmt.Println(number)     // Printing its value
    var decision = true     // Declaring and initializing a boolean variable without stating
    fmt.Println(decision)   // Printing its value
}
```



Automatic Type Inference

You can see that we just declared variable `number` at **line 5** and `decision` at **line 8**, without stating their types explicitly. The compiler infers type by itself, and the result is the same as the previous program. **5** and **true** are printed.

Short form with `:=` assignment operator

With the type omitted, the keyword **var** is pretty superfluous in **line 5** and **line 7** of the above program. So we can also write it as:

```
number := 5 // line 5
decision := true // line 7
```

Again the types of `number` and `decision` (int and bool) are inferred by the compiler. This is the preferred form, but it can only be used inside functions, *not* in package scope. This operator (:=) effectively makes a new variable; it is also called an initializing declaration.

If after the lines above in the same code block, we declare :

```
number:= 20
```

This is not allowed. The compiler gives the error: `no new variables on the left side of :=`. However, **number** = **20** is okay because then the same variable only gets a new value.

> **Note**: If a variable named `v` is used but not declared, it will give a compiler error: `undefined: v`. And, if `v` was declared as a local variable but not used, then the compiler will give the error: `v declared and not used`.

That's all for the introduction to variables. Let's see how the scope for a variable is defined and how referencing works in Go.