GO - VARIABLES

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in Go has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because Go is case-sensitive. Based on the basic types explained in previous chapter, there will be the following basic variable types:

Type	Description
byte	Typically a single octet*onebyte*. This is an byte type.
int	The most natural size of integer for the machine.
float32	A single-precision floating point value.

Go programming language also allows to define various other types of variables, which we will cover in subsequent chapters like Enumeration, Pointer, Array, Structure, Union, etc. For this chapter, let us study only basic variable types.

Variable Definition in Go:

A variable definition means to tell the compiler where and how much to create the storage for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows:

```
var variable_list optional_data_type;
```

Here, **optional_data_type** is a valid Go data type including byte, int, float32, complex64, boolean or any user-defined object, etc., and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here:

```
var    i, j, k int;
var   c, ch byte;
var  f, salary float32;
d = 42;
```

The line **var i, j, k;** both declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j and k of type int.

Variables can be initialized *assignedaninitialvalue* in their declaration. The type of variable is automatically judged by the compiler based on the value passed to it. The initializer consists of an equal sign followed by a constant expression as follows:

```
variable_name = value;
```

Some examples are:

```
d = 3, f = 5;    // declaration of d and f. Here d and f are int
```

For definition without an initializer: variables with static storage duration are implicitly initialized with nil *allbyteshavethevalue*0; the initial value of all other variables is zero value of their data type.

Static type declaration

A static type variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable declaration at the time of linking of the program.

Example

Try following example, where variable has been declared with a type, and have been defined and initialized inside the main function:

```
package main

import "fmt"

func main() {
   var x float64
   x = 20.0
   fmt.Println(x)
   fmt.Printf("x is of type %T\n", x)
}
```

When the above code is compiled and executed, it produces the following result:

```
20
x is of type float64
```

Dynamic type declaration / Type Inference

A dynamic type variable declaration requires compiler to interpret the type of variable based on value passed to it. Compiler don't need a variable to have type statically as a necessary requirement.

Example

Try following example, where variables have been declared without any type, and have been defined and initialized inside the main function. Notice, in case of type inference, we've initialized the variable y with **:=** operator wheree as x is initilized using **=** operator.

```
package main

import "fmt"

func main() {
   var x float64 = 20.0

   y := 42
   fmt.Println(x)
   fmt.Println(y)
   fmt.Printf("x is of type %T\n", x)
   fmt.Printf("y is of type %T\n", y)
}
```

When the above code is compiled and executed, it produces the following result:

```
20
42
x is of type float64
y is of type int
```

Mixed variable declaration

Variables of different types can be declared in one go using type inference.

Example

```
package main

import "fmt"

func main() {
   var a, b, c = 3, 4, "foo"

   fmt.Println(a)
   fmt.Println(b)
   fmt.Println(c)
   fmt.Printf("a is of type %T\n", a)
   fmt.Printf("b is of type %T\n", b)
   fmt.Printf("c is of type %T\n", c)
}
```

When the above code is compiled and executed, it produces the following result:

```
3
4
foo
a is of type int
b is of type int
c is of type string
```

Lvalues and Rvalues in Go:

There are two kinds of expressions in Go:

1. **lvalue :** Expressions that refer to a memory location is called "lvalue" expression. An lvalue may appear as either the left-hand or right-hand side of an assignment.

2. **rvalue :** The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and can not appear on the left-hand side. Following is a valid statement:

```
x = 20.0
```

But following is not a valid statement and would generate compile-time error:

```
10 = 20
```

Loading [MathJax]/jax/output/HTML-CSS/jax.js