**¿Qué es Go?**

Es un leguaje open source creado por Google, su aparición fue en el año 2009 y sus diseñadores son Robert Griesemer, Rob Pike y Ken Thompson. Además es un lenguaje de **programación concurrente, compilado, estructurado, tipado estático e imperativo.** También no es un lenguaje orientado a objetos y cuenta con un recolector de basura.

**Características de Go**

* Go utiliza una sintaxis parecida a la de C.
* Posee un tipado estático y resulta ser igual de eficiente que C.
* Tiene muchas características y facilidades de lenguajes dinámico como Python.
* No es orientado a objetos debido a que no existe una jerarquía de tipos, pero si implementa la interface.
* Declaraciones de variable simple.
* Duck typing – Tipificación dinámica de datos.
* No tiene excepciones.

**Declaración de variables**

Para declarar una variable se usa la palabra reservada VAR, haciendo referencia que es una variable después el nombre de la variable y por último el tipo.

A diferencia de otros lenguajes de programación la declaración en GO es al revés. GO inicializa la variable con un valor por defecto.



Figure 1: Declaración de variable en Go

En este caso **numero,** Go lo inicializará con el valor de 0 pero podemos asignarle un valor por defecto.

Figure 2: Asignación de variable

Para declarar una variable de manera rápida determinando el tipo de manera dinámica una vez que la variable se le asigna una tipo este ya no se puede cambiar el operador **:=** solo se cuándo se declara una nueva variable.



Figure 3: Declaración rápida de una variable

Otra forma de declarar variables omitiendo el tipo es usando **var + nombre** y dándole una asignación.

Figure 4: Otra forma de declaración usando el Duck typing

En GO se pueden asignar valores de forma consecutiva.



Figure 5: Asignación de valores de forma consecutiva

Esta asignación de valores puede resultar útil al momento de intercambiar valores entre variables del mismo tipo.



Figure 6: Intercambio de valores entre variables



Figure 7: Variables Intercambiadas

Esta forma de asignación de valores se puede utilizar cuando se declara una nueva variable.

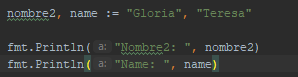


Figure 8: Declaración de una variable y asignación

En este caso **nombre2** es la variable creada y su valor es **Gloria**, mientras que **name** era una variable previamente declarada por lo cual su valor asignado es **Teresa**.

Para declarar múltiples variables se usa **var(),** dentro de los paréntesis se declaran las variables con sus asignaciones. Al usar esta forma de declaración se puede asignar el tipo o bien usando el Duck typing (tipado dinámico).

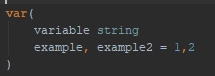


Figure 9: Declaración de múltiples variables

**NOTAS:**

En **Go** toda variable que se es declarada debe de ser utilizada de la misma forma sucede con los **import** sino se usa una librería importada el compilador obliga a eliminarla o usarla.

La librería fmt es una abreviación de format, esta se usa para poder imprimir e ingresar datos.

**Alcance de las variables**

En Go el alcance de las variables es como cualquier otro lenguaje de programación es decir las variables pueden funcionar de manera global o de manera local.

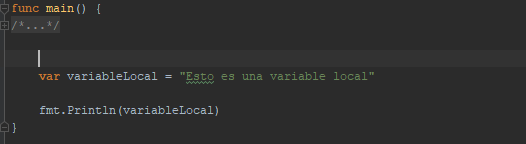


Figure 10: Ejemplo de variable local

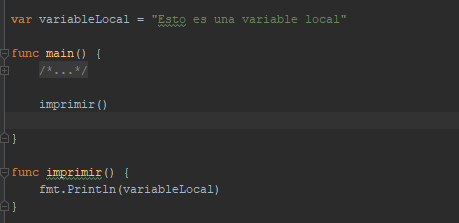


Figure 11: Ejemplo de variable global

**NOTAS:**

Cuando se declara una variable global no se permite usar el operador **:=** por lo que es obligado declararla de forma apropiada.

**Tipos de datos en Go**

Comencemos por los tipos básicos, estos pueden ser **números (int), booleanos y cadenas.**

Tipos numéricos:

|  |  |  |
| --- | --- | --- |
| Tipo | Rango | Consumo de memoria |
| uint8 | 0 a 255 | 1 byte |
| uint16 | 0 a 65535 | 2 byte |
| uint32 | 0 a 4294967295 | 4 byte |
| uint64 | 0 a 18446744073709551615 | 8 byte |

**Enteros sin signo**

**Enteros con signo**

|  |  |  |
| --- | --- | --- |
| Tipo | Rango | Consumo de memoria |
| int8 | -128 a 127 | 1 byte |
| int16 | -32768 a 32767 | 2 byte |
| int32 | -2147483648 a 2147483647 | 4 byte |
| int64 | -9223372036854775808 a 9223372036854775807 | 8 byte |

**Alias**

|  |  |  |
| --- | --- | --- |
| Tipo | Rango | Consumo de memoria |
| byte (lo mismo que uint8) | 0 a 255 | 1 byte |
| rune (lo mismo que int32) | -2147483648 a 2147483647 | 4 byte |

**Tipos dependientes de la plataforma**

|  |  |
| --- | --- |
| Tipo | Rango |
| uint | De pende de la plataforma (32bits o 64 bits) |
| int | De pende de la plataforma (32bits o 64 bits) |
| uintptr | Entero suficientemente largo para almacenar un puntero |

Tipos float:

|  |  |  |
| --- | --- | --- |
| Tipo | Precisión | Consumo de memoria |
| Float32 | 6 digitos | 4 byte |
| Float64 | 15 digitos | 8 byte |
| Complex64 | Numero complejo para float32 | 8 byte |
| Complex128 | Numero complejo para float64 | 16 byte |

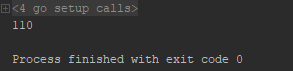
Tipos boolean:

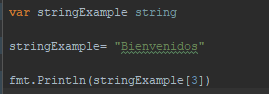
Los tipos de datos boolean solo pueden almacenar dos valores **false** o **true.**

Tipo string:

En Go a diferencia de los demás lenguajes de programación los string son una secuencia de bytes.

Otras características del tipo string en Go es que son indexables e inmutables. Cuando se intenta acceder a un carácter que conforma un string lo que se obtiene es el código Unicode (utf-8).





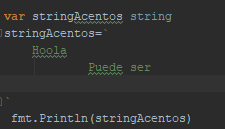
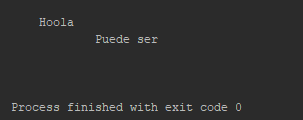
Como se aprecia en el ejemplo, si tratamos de acceder a la posición 3 nos devuelve un número que es el **110** el cuál en utf-8 corresponde a la letra n minúscula.

Para obtener un segmento de caracteres o subcadena como se conoce en java, Go lo hace con algo llamado **Slice**.



Figure 12: Uso de Slice para obtener una subcadena

En Go hay dos formas de crear cadenas, una es usar las comillas dobles (**“”**) o con el acento invertido (**`**), el uso del acento invertido es para que Go use o imprima la cadena tal y como esta, es decir si lleva sangría y espacio además de ignorar los caracteres especiales.



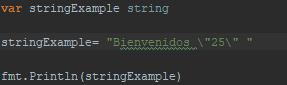
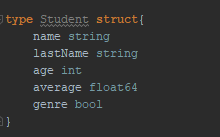
El uso de caracteres especiales se hace usando la pleca invertida (**\**)**.**

Figure 13: Uso de caracteres invertidos en una cadena

Para convertir un tipo de dato a string se usa la función **Itoa** del paquete **strconv**.

En los tipos de conjuntos tenemos los **arreglos** y las **estructuras.**



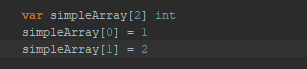


Figure 15: Ejemplo de un array con su inicialización

Figure 14: Ejemplo de una estructura

**Nota:**

* Para conocer la cantidad de caracteres que posee una cadena se usa la función **len.**

Los **Slice**:

Son iguales a los arrays con la diferencia que estos no se especifica la dimensión que va a tener, además de ser similar a los arrays dinámicos de otros lenguajes.



Figure 17: Declaración de Slice con su inicialización

Figure 16: Ejemplo de creación de un slice

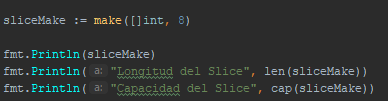
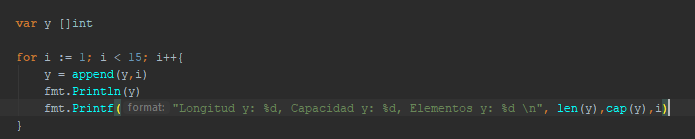
También se puede usar la función **make** para crear un slice

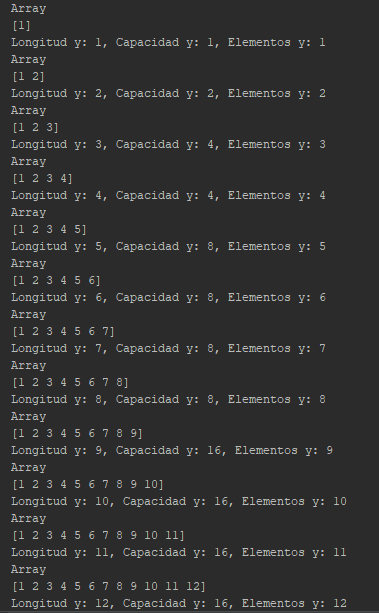
Figure 18: Uso de la función make

A esta función se le pueden pasar 3 parámetros, uno es el tipo del slice, segundo el tamaño y tercero la capacidad.

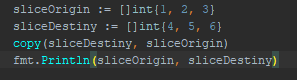
Los slice al ser similares a los arrays llega un punto en el cual se quedan sin capacidad y al ser estáticos esta no se puede modificar, para ello Go creo la función **append** la cual permite que el slice aumente su longitud y la capacidad. Pero esta función lo que realmente hace es crear otro slice con otro array de fondo y destruir el anterior.

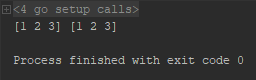


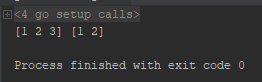
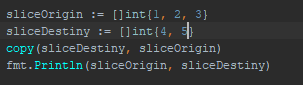
Este es un pequeño ejemplo para demostrar que append duplica la capacidad del array que esta de fondo.



En los slice no solo se puede añadir sino que también se puede copiar, con la función **copy.**





Algo muy importante sobre la función copy es que si el slice de destino tiene una longitud menor al slice de origen solo se copian los datos necesarios, lo mismo sucede a la inversa si los datos que contiene el slice de origen es menor a los datos que contiene el slice destino solo se copian los datos necesario.

**Casting de valores**

En Go el casting tiene que ser de forma explicito debido que Go no es un lenguaje de gran potencia como lo son Java o C# que se puede hacer un casteo de forma implícita.

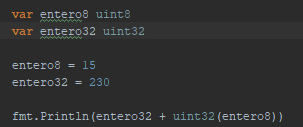


Figure 19: Ejemplo de castin implícito en Go

**Estructuras de control**

En Go solo existe una estructura iterativa que es el **for**.

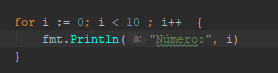


Figure 20: Estructura for en Go

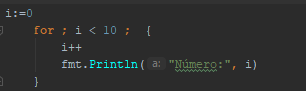
La estructura **for** en Go esta compuesta de la siguiente manera:

* La declaración de una variable antes de la primera iteración
* La condición a evaluar antes de cada iteración
* La sentencia después de cada iteración, es decir el aumento de la variable definida.

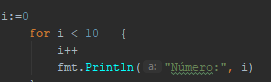
**Nota:**

A diferencia de otros lenguajes como C, Java o JavaScript no hay paréntesis alrededor de los componentes del **for** y las llaves son necesarias.

La declaración de la variable y la sentencia de incremento son opcionales.

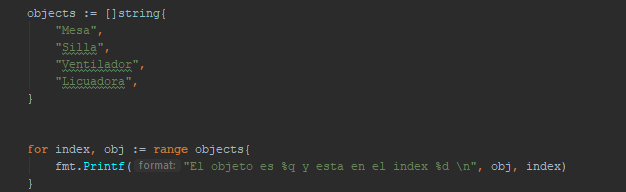


Si le quitamos los **;** el while que se conoce en C, en Go es el mismo for.



**For Range**

El for range es como el for each de otros lenguajes de programación recorre un arreglo o un slice del cual podemos tener acceso a sus valores.



Otra forma de usar el for range

El guión bajo se utiliza para omitir el índice ya que range devuelve el valor y el índice.

Estructura **if**

Al igual que for el if no es necesario que su condición esté rodeada de paréntesis, pero las llaves son necesarias.

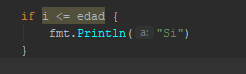


Figure 21: Estructura if en Go

Al igual que en otros lenguajes de programación existe el **if else**

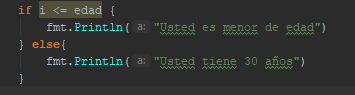


Figure 22: If Else en Go

Estructura **switch**

En Go la sentencia switch es igual que en otros lenguajes como C, C++, Java, PHP, etc. Con la excepción que en Go solo se ejecuta el caso seleccionado, por lo que la sentencia break necesaria en los demás lenguajes es automática en Go.

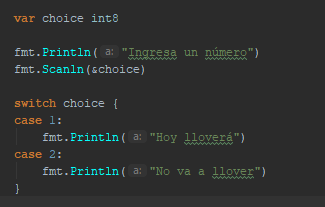


Figure 23: Switch en Go

A veces es necesario que el switch evalué más casos, para eso los desarrolladores de Go crearon la palabra **fallthrough** la cual indica que el programa siga evaluando más casos.