GO

Go, a menudo referido como "Golang", es un lenguaje de programación multiplataforma de código abierto, desarrollado por Google en 2007. Es un lenguaje **compilado y fuertemente tipado**, con una sintaxis basada en C/C++.

Principales Usos y Razones para Usar Go:

• **Usos**: Se utiliza principalmente para desarrollo web (lado del servidor/back-end), desarrollo de aplicaciones en red, aplicaciones concurrentes, desarrollo de aplicaciones multiplataforma y desarrollo nativo en la nube.
• **Razones para recomendarlo**: Es **fácil de aprender** y **portable a diferentes plataformas** (Windows, Mac, Linux). También es valorado por su **rapidez en tiempo de ejecución y compilación**, soporte para concurrencia y genéricos, y gestión eficiente de la memoria.

Estructura Básica de un Programa Go:

• Un programa Go se organiza en **paquetes (packages)**.
• Todo programa ejecutable debe pertenecer al paquete `main`.
• La ejecución comienza en la función `main()`.
• Los **identificadores** (variables, funciones, etc.) que comienzan con **mayúscula son "exportados"** (visibles desde otros paquetes, es decir, públicos), mientras que los que no, son privados.

Manejo de Variables, Constantes y Tipos de Datos:

• **Variables**: Se declaran con la palabra clave `var` y pueden incluir inicialización. Go permite **inferir el tipo** si se inicializa, usando la sintaxis corta `:=` (solo dentro de funciones). Los nombres de variables son sensibles a mayúsculas y minúsculas.
• **Constantes**: Se declaran con `const` y pueden ser de tipos carácter, string, boolean o numéricos, y también pueden ser sin tipo.

• **Tipos Básicos y Valores por Defecto (Zero Values)**:
  ◦ `bool`: `false`.
  ◦ `string`: `""` (cadena vacía).
  ◦ `int`: `0`.
  ◦ `float64`: `0`.
  ◦ Otros tipos incluyen `uint`, `byte`, `rune`, `float32`, `complex64`, `complex128`.
• **Conversión de Tipos**: Se realiza explícitamente usando `T(v)`, donde `T` es el tipo deseado y `v` el valor a convertir.

Estructuras de Control:

• **`for` (Iteración)**: Go solo tiene la palabra clave `for` para bucles. Puede funcionar como:
  ◦ Un bucle `for` tradicional con inicialización, condición e incremento (ej. `for i := 0; i <= 9; i++`).
  ◦ Un bucle `for` sin sentencia de inicialización o post-sentencia, similar a un `while` en otros lenguajes (ej. `for sigo {...}`).
  ◦ Un bucle infinito (que se rompe con `break`).
  ◦ Para iterar sobre colecciones con `for...range`.

• **`if` (Selección)**: Permite la ejecución condicional de código. Puede incluir una sentencia de inicialización opcional antes de la condición.
• **`switch` (Selección)**: Se utiliza para la selección de casos.
  ◦ **Sin selector**: Funciona como una cadena de sentencias `if-else if-else`, ejecutando el primer `case` cuya expresión es verdadera. Es útil para condiciones complejas basadas en múltiples expresiones booleanas.
  ◦ No necesita `break` al final de cada `case`, ya que solo ejecuta el bloque coincidente y sale.

Funciones y Métodos:

• **`fmt` Package**:
  ◦ `fmt.Print()`: Imprime argumentos, pone un espacio entre ellos (excepto para cadenas), no añade nueva línea al final.
  ◦ `fmt.Println()`: Imprime argumentos, pone un espacio (incluso para cadenas), y agrega un "newline" al final.
  ◦ `fmt.Printf()`: Permite impresión formateada usando "verbos" (ej. `%s`, `%d`, `%v`), no añade nueva línea a menos que se especifique (`\n`).
• Parámetros de Retorno Nombrados: Permiten declarar las variables de retorno directamente en la firma de la función. Al usar `return` sin argumentos, los valores asignados a estas variables se retornan automáticamente.
• Pasaje de Parámetros: Go utiliza un modelo de **pasaje de parámetros "por valor"**. Una copia de cada argumento se pasa a la función.
  ◦ **Arrays y Structs**: Si se pasa un array o un struct a una función, se pasa una copia completa. Las modificaciones dentro de la función no afectarán al original.
  ◦ **Modificación de Valores Externos**: Para modificar un valor externo, se deben **pasar punteros** a esos valores como parámetros. Aunque el puntero en sí es una copia, el contenido al que apunta puede ser desreferenciado y modificado, afectando el valor original.
• Receptores de Métodos:
  ◦ **Receptor de valor**: Se pasa una copia del valor del receptor. El método no puede modificar el valor original del receptor.
  ◦ **Receptor de puntero**: Se recibe la dirección de memoria del valor, permitiendo que el método modifique el valor original.
  ◦ Un receptor puede ser `nil`. Si no se maneja explícitamente y se intenta desreferenciar un puntero `nil`, causará un "runtime error" (panic).

Colecciones (Arrays, Slices, Maps):

• **Arrays**:
  ◦ Secuencia indexada de elementos del mismo tipo, con una **longitud fija** que es parte de su definición de tipo. Su tamaño no puede cambiar una vez declarado.
  ◦ El primer índice es cero.
• **Slices**:
  ◦ Son "segmentos" de un array subyacente, **más flexibles** que los arrays.
  ◦ Tienen una **longitud (`len`)** (elementos actuales) y una **capacidad (`cap`)** (cuánto puede crecer sin reasignar memoria).
  ◦ Su longitud puede **cambiar dinámicamente**.
  ◦ Son **referencias a los arrays subyacentes**: modificar un slice puede afectar al array original y a otros slices que comparten el mismo array.

◦ **Creación**: A partir de un array existente (`s := a[2:4]`), usando un literal de slice (`s3 := []int{1, 2, 3}`), o con la función `make()` (`make([]Tipo, longitud, capacidad)`).
◦ **`append()`**: Se usa para agregar elementos. Si la capacidad es insuficiente, Go asigna un nuevo array subyacente y copia los elementos.
◦ **`copy()`**: Permite copiar elementos de un slice a otro.
◦ **Iteración (`for range`)**: Se puede acceder al índice y al elemento, solo al elemento, o solo al índice.
• **Maps**:
◦ Colecciones **no ordenadas** de pares clave-valor (también conocidos como arreglos asociativos, tablas hash o diccionarios).
◦ **No permiten claves duplicadas**.
◦ **Tipos de claves permitidos**: Deben ser **comparables** (booleans, números, strings, arrays). **No permitidos**: slices, maps y functions.
◦ **Zero value**: `nil`. No se pueden agregar elementos a un map `nil` (causaría un error en tiempo de ejecución); debe ser inicializado con `make` o un literal.
◦ **Operaciones**:
▪ Agregar o modificar: `m[key] = value.`
▪ Eliminar: `delete(m, key)`.
▪ Recuperar valor: `elem = m[key]` (devuelve el zero value si la clave no existe).
▪ Verificar existencia de clave: `elem, ok = m[key]` (retorna el valor y un booleano `ok`).
◦ **Iteración (`for range`)**: Devuelve la clave y el valor. **El orden de iteración no está garantizado**.
Punteros:
• Un puntero es la **dirección de memoria de un contenido** de cierto tipo.
• Su **"zero value" es `nil`**.
• **Operadores**:
◦ `&` (operador de dirección): Devuelve la dirección de memoria de una variable (ej. `p = &i`).
◦ `*` (operador de desreferencia): Accede al valor almacenado en la dirección de memoria a la que apunta el puntero (ej. `fmt.Println(*p)`).
• **`new(T)`**: Asigna memoria para una variable de tipo `T` y devuelve un puntero a esa memoria, inicializando el valor con su "zero value" (ej. `p := new(int)` inicializa un entero en 0).

Concurrencia:

Go fomenta la concurrencia a través de Goroutines y Channels.
• **Goroutines**:
◦ Funciones que se ejecutan **concurrentemente** con otras funciones. Son ligeras.
◦ Se lanzan añadiendo la palabra clave `go` antes de la llamada a la función (ej. `go f(0)`).
◦ **`sync.WaitGroup`**: Se usa para **esperar a que varias Goroutines terminen** su ejecución. Funciona como un contador.
▪ `Add(n int)`: Incrementa el contador.
▪ `Done()`: Decrementa el contador en 1.
▪ `Wait()`: Bloquea la goroutine que la ejecuta hasta que el contador llegue a cero.
▪ Uso típico: `wg.Add(1)` antes de cada goroutine, `defer wg.Done()` dentro de la goroutine, y `wg.Wait()` en la función principal.

• **Channels**:
  ◦ Mecanismo principal para la **comunicación y sincronización entre Goroutines**. Son conductos "tipados".
  ◦ **Comportamiento por defecto (no buferizado)**: Las operaciones de **envío y recepción bloquean** a la goroutine que la ejecuta hasta que la del "otro extremo" esté lista.
  ◦ **Declaración**: `make(chan Tipo)` o `var nombre chan Tipo = make(chan Tipo)`.
  ◦ **Zero value**: `nil`.
  ◦ **Tipos de Channels**:
    ▪ **`chan <- int` (send-only)**: Solo la goroutine "sender" puede cerrarlo.
    ▪ **`<-chan int` (receive-only)**: Intentar cerrarlo produce un error en tiempo de compilación.
  ◦ Pueden tener un **búfer** para permitir envíos y recepciones sin bloqueo hasta que el búfer se llene.
  ◦ Se pueden **cerrar** con `close(chan)`.
• **`select`**:
  ◦ Permite que una goroutine **espere por más de un channel** (para envío o recepción).
  ◦ Puede incluir una cláusula **`default`** para realizar operaciones sin bloqueo si ninguna otra operación de `case` está lista.
• **Exclusión Mutua**: Para proteger recursos compartidos y evitar condiciones de carrera.
  ◦ **`sync.Mutex`**: Un semáforo binario que proporciona **exclusión mutua para escritura y lectura**.
    ▪ `Lock()`: Bloquea el Mutex. Si ya está bloqueado, la goroutine que invoca `Lock()` se bloquea hasta que otra invoque `Unlock()`.
    ▪ `Unlock()`: Desbloquea el Mutex. Si no está bloqueado, produce un error en tiempo de ejecución.
  ◦ **`sync.RWMutex`**: Un semáforo "un escritor – múltiples lectores". Permite que **múltiples lectores accedan concurrentemente** (`RLock()`), pero solo un único escritor acceda exclusivamente (`Lock()`).

Panic y Recover:

• **`panic`**: Ocurre cuando Go detecta un error en tiempo de ejecución, deteniendo la ejecución normal del programa. Puede ser generado explícitamente con `panic()`. Se usa para **errores "no esperables"**.
• **`recover()`**: Una función que permite **recuperar la ejecución ante un `panic`** o al menos dejar el estado prolijo. Solo tiene efecto si es invocada **dentro de una función diferida (`defer`)** durante un `panic`. Finaliza el estado de pánico y retorna el valor del panic, aunque la función que paniqueó no continúa.

Genéricos:

Go introdujo los genéricos para resolver el problema de tener que **escribir funciones duplicadas** para la misma operación sobre diferentes tipos de datos, permitiendo escribir código reutilizable y más limpio sin perder seguridad de tipos.
• **Funciones Genéricas**: Permiten definir funciones que aceptan **parámetros de tipo**. Los parámetros de tipo tienen "restricciones de tipo" que especifican qué tipos de datos pueden reemplazarlos.
  ◦ Ejemplo sintaxis: `func SumIntsOrFloats[K comparable, V int64 | float64](m map[K]V) V`.

• **Tipos Genéricos (Structs)**: Los `structs` pueden ser genéricos. Permiten definir estructuras de datos (como listas o árboles) que pueden almacenar elementos de cualquier tipo especificado por el parámetro de tipo.
  ◦ Ejemplo conceptual: `type List[T any] struct { ... }`.
• **"Type Constraints" Comunes**:
  ◦ **`any`**: Representa cualquier tipo de dato.
  ◦ **`comparable`**: Representa tipos que tienen definida la comparación por igualdad (`==`).
  ◦ **`Stringer`**: Representa cualquier tipo que implemente la interfaz `Stringer` (con el método `String() string`).
  ◦ Uniones de tipos específicos (ej. `int | float64`).

Espero que este resumen te sea útil para comprender los conceptos clave de Go presentados en las fuentes.