GoLang

Variables

Variable declaration

- var foo int
- var foo int = 42
- foo := 42

Can’t redeclare variables, but can shadow them

All variables must be used

Visibility

- Lower case first letter for package scope
- Upper case first letter to export (public)
- Variable inside a function has block scope
- No private scope

Naming conventions

- PascalCase or camelCase
- Capitalize acronyms variable i.e. HTTP, URL
- Name it as short as reasonable
- Longer names for longer lives

Primitives

Boolean type

- Values are true or false

- Not an alias for other types (e.g. int)
- Zero value is false

Numeric types

- Integers
    - Signed integers
        - int type has varying size, but min 32 bits
        - 8 bit (int8) through 64 bit (int64)
    - Unsigned integers
        - 8 bit (byte and unit8) through 32 bit (uint32)
    - Arithmetic operations
        - Addition, subtraction, multiplication, division, remainder
    - Bitwise operations
        - And, or, xor, and not
    - Zero value is 0
    - Can’t mix types in same family! (Uint16 + uint32 = error)
- Floating pint numbers
    - Follower IEEE-754 standard
    - Zero value is 0
    - 32 and 64 bit versions
    - Literal styles
        - Decimal (3.14)
        - Exponential (13e18 or 2E10)
        - Mixed (13.7e12)
    - Arithmetic operations
        - Addition, subtraction, multiplication, division

- Complex numbers
  - Zero value is (0+0i)
  - 64 and 128 bit versions
  - Built-in functions
    - complex - make complex number from two floats
    - real - get real part as float
    - imag - get imaginary part as float
  - Arithmetic operations
    - Addition, subtraction, multiplication, division

Text types

- Strings
  - UTF-8
  - Immutable
  - Can be concatenated with plus (+) operator
  - Can be converted to []byte
- Rune
  - UTF-32
  - Alias for int32
  - Special methods normally required to process
    - e.g. strings.Reader#ReadRune

Constants

Immutable, but can be shadowed

Replaced by the compiler at compile time

- Value must be calculable at compile time

Named like variables

- PascalCase for exported constants
- camelCase for internal constants

Typed constants work like immutable variables

- Can interoperate only with same type

Untyped constants work like literals

- Can interoperate with similar types

Enumerated constants

- Special symbol iota allows related constants to be created easily
- Iota starts at 0 in each const block and increments by one
- Watch out of constant values that match zero values for variables

Enumerated expressions

- Operations that can be determined at compile time are allowed
    - Arithmetic
    - Bitwise operations
    - Bitshifting

Array and Slices

Arrays

- Collection of items with same type
- Fixed size
- Declaration styles
    - a := [3]int{1, 2, 3}
    - a := […]int{1, 2, 3}
    - var a [3]int
- Access via zero-based index
    - a := [3]int{1, 3, 5} // a[1] == 3
- len() function returns size of array
- Copies refer to different underlying data

Slices

- Backed by array
- Creation styles
    - Slice existing array or slice
    - Literal style
    - Via make function
        - a := make([]int, 10) // create slice with capacity and length == 10
        - a := make([]int, 10, 100) // slice with length == 10 and capacity == 100
    - len() function returns length of slice
    - cap() function returns length of underlying array
    - append() function to add elements to slice
        - May cause expensive copy operation if underlying array is too small
    - Copies refer to same underlying array

Maps and Structs

Maps

- Collections of value types that are accessed via keys
- Created via literals or via make() function
- Members accessed via [key] syntax
    - myMap[“key”] = “value”
- Check for presence with “value, ok” form of result
- Multiple assignments refer to the same underlying data

Structs

- Collections of disparate data types that describe a single concept
- Keyed by named fields
- Normally created as types, but anonymous structs are allowed
- Structs are value types
- No inheritance, but can use composition via embedding
- Tags can be added to struct fields to describe field

If and Switch Statements

If statements

- Initializer
- Comparison operators
- Logical operators
- Short circuiting
- If - else statements
- If - else if statements

- Equality and floats

Switch statements

- Switching on a tag
- Cases with multiple tests
- Initializers
- Switches with no tag (use comparison statements)
- Fallthrough
- Type switches
- Breaking out early

Looping

For statements

- Simple loops
  - for initializer; test; increment {}
  - for test {}
  - for {}
- Exiting early
  - break
  - continue
  - labels
- Looping over collections
  - array, slices, maps, strings, channels
  - for k, v := range collections {}

Defer, Panic and Recover

Defer

- Used to delay execution of a statement until function exits
- Useful to group “open” and “close” functions together
    - be careful in loops
- Run in LIFO (last-in, first-out) order
- Arguments evaluated at time defer is executed, not at time of called function execution

Panic

- Occur when program cannot continue at all
    - don't use when file can’t be opened, unless it is critical
    - use for unrecoverable events - cannot obtain TCP port for web server
- Function will stop executing
    - deferred functions will still fire
- If nothing handles panic, program will exit

Recover

- Used to recover from panics
- Only useful in deferred functions
- Current function will not attempt to continue, but higher functions in call stack will

Pointers

Creating pointers

- Pointer types use an asterisk (*) as a prefix to type pointed to

    - int - a pointer to an integer
- Use the address-of operator (&) to get address of variable

Referencing pointers

- Dereference a pointer by preceding with an asterisk (*)
- Complex types (e.g. structs) are automatically dereferenced

Create pointers to objects

- Can use the address-of operator (&) if value type already exists
    - ms := myStruct{foo: 42}
    - p := &ms
- Use address-of operator before initializer
    - &myStruct{foo: 42}
- Use the “new” keyword
    - Can’t initialize fields at the same time

Types with internal pointers

- All assignment operations in Go are copy operations
- Slices and maps contain internal pointers, so copies point to the same underlying data

Functions

Basic syntax

- func foo() {

…

}

Parameters

- Comma delimited list of variables and types
  - func foo(bar string, baz int)
- Parameters of same type list type once
  - func foo(bar, baz int)
- When pointers are passed in, the function can change the value in the caller
  - this is always true for data of slices and maps
- Use variadic parameters to send list of same types in
  - must be the last parameter
  - received as a slice
  - func foo(bar string, baz …int)

Return values

- Single return values just list type
  - func foo() int
- Multiple return value list types surrounded by parentheses
  - func foo() (int, error)
  - the (result type, error) paradigm is a very common idiom
- Can use named return values
  - initializes returned variable
  - return using return keyword on its own
- Can return addresses of local variables
  - automatically promoted from local memory (stack) to shared memory (heap)

Anonymous functions

- Functions don’t have names if they are:
  - immediately invoked
  - func() {

…

}()

- Assigned to a variable or passed as an argument to a function
  - a := func() {

…

}

a()

Functions as types

- Can assign functions to variables or use an arguments and return values in functions
- Type signature is like function signature, with parameter names
  - var f func(string, string, int) (int, error)

Methods

- Function that executes in context of a type
- Format
  - func (g greeter) greet() {

…

}

- Receiver can be value or pointer
  - value receiver gets copy of type

    - pointer receiver gets pointer to type

Interfaces

Basics

```
type Writer interface {
    Write([]byte) (int, error)
}
type ConsoleWriter struct {}
func (cw ConsoleWriter) Write(data []byte) (int, error) {
    n, err := fmt.Println(string(data))
    return n, err
}
```

Composing interfaces

```
type Writer interface {
    Write([]byte) (int, error)
}
type Closer interface {
    Close() error
}
type WriterCloser interface {
    Writer
    Closer
}
```

Type conversion

- var wc WriterCloser = NewBufferedWriterCloser()

  bwc := wc.(*BufferedWriterCloser)

The empty interface and type switches

- var interface {} = 0

```
switch i.(type) {
    case int:
        fmt.Println(“i is an integer”)
    case string:
        fmt.Println(“i is a string”)
    default:
        fmt.Println(“I don’t know what i is”)
}
```

Implementing with values vs pointers

- Methods set of value is all methods with value receivers
- Methods set of pointer is all methods, regardless of receiver type

Best practices

- Use many, small interfaces
  - Single method interfaces are some of the most powerful and flexible
    - io.Writer, io.Reader, interface{}
- Don’t export interfaces for types that will be consumed
- Do export interfaces for types that will be used by package
- Design functions and methods to receive interfaces whenever possible

Goroutines

Creating goroutines

- Use go keyword in front of function call
- When using anonymous functions, pass data as local variables

Synchronization

- Use sync.WaitGroup to wait for groups of goroutines to complete
- Use sync.Mutex and sync.RWMutex to protect data access

Paralelism

- By default, Go will use CPU threads equal to available cores
- Change with runtime.GOMAXPROCS
- More threads can increase performance, but too many can slow it down

Best practices

- Don’t create goroutines in libraries
  - let consumer control concurrency
- When creating a goroutine, know how it will end
  - avoids subtle memory leaks
- Check for race conditions at compile time
  - go run -race <file.go>

Channels

Channel basics

- Create a channel with make command

    - make(chan int)
- Send message into channel
    - ch <- val
- Receive message from channel
    - val := <- ch
- Can have multiple sender and receiver

Restricting data flow

- Channel can be cast into send-only or receive-only versions
    - send-only: chan<- int
    - receive-only: <-chan int

Buffered channel

- Channels block sender side till receiver is available
- Block receiver side till message is available
- Can decouple sender and receiver with buffered channels
    - make(chan, int, 50)
- Use buffered channels when sender and receiver have asymmetric loading

For…range loops with channels

- Use to monitor channel and process messages as they arrive
- Loop exits when channel is closed

Select statements

- Allows goroutine to monitor several channels at once
    - blocks if all channels block

- if multiple channels receive value simultaneously, behavior is undefined