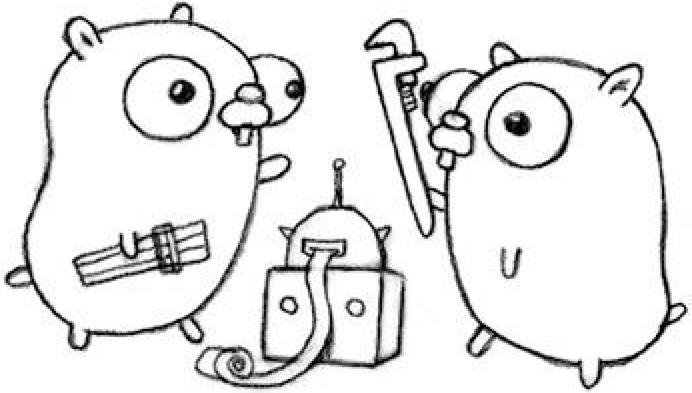
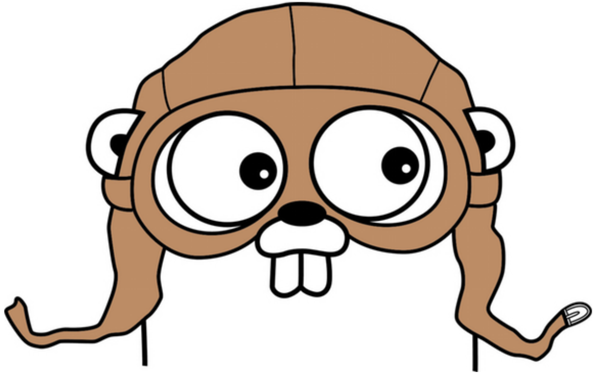
Idiomatic Go

Part 1



What does “idiomatic” mean?

“Having a distinct style or character”



Design Principles of Go

- Simplicity
- Reducing repetition in code
- Minimal set of keywords
- No type hierarchy
- Well separated (orthogonal) modules which are highly reusable

How do you write Idiomatic Go?

- Learn the constructs and concepts of the language

- Look at code of successful modules, and most of all, the standard library

- Practice, practice, practice…

What is this presentation about?

- Types
- How to use them
- The tricky aspects that might trip you up

Go Types

- Go is a strongly typed language
- No automatic conversions, even between numerical types

Therefore, knowing types is crucial to writing effective Go programs!

Built-in types: integers

Signed	Unsigned
int8 (-128..127)	**uint8** = byte (0..255)
int16 (-32768..32767)	**uint16** (0..65535)
int32=rune (-2,147,483,648..2,147,483,647)	**uint32**(0..4294967295)
int64 (-9,223,372,036,854,775,808.. 9,223,372,036,854,775,807)	**uint64** (0..18,446,744,073,709,551,615)

int, **uint** are machine dependent: 16,32 or 64 bit

Built-in types: integer operators

Arithmetic operators	+ - * / % (remainder) var++ (postfix only!) var--
Bitwise operators	& | ^(xor) &^(and-not) << (shift left) >> (shift right)

Built-in types: integer overflows

- No runtime check for overflows
- example:

```
var a int8
a=-128
a--
fmt.Println(a)
```

Outputs: 127

Built-in types: floating point

- **float32**, **float64**
- **complex64**, **complex128**

Arithmetic operators:

+ - * / %

- Package “math” operates on float64
- Package “math/cmplx” operates on complex128

Built-in types: example of complex

```
c:=complex(0,1) //real=0, imaginary=1
a:=c*c
fmt.Println(a)
```

Outputs: (-1+0i)

Built-in Types: Strings

- encoding is always UTF-8
- 1 character = 1-4 bytes!!
- literals:
  - “one line string \nwith escaping”
  - `multiline

        string with no “escaping” `

Only operator is: **+**

Built-in Types: Example of string

```
name:="László Szenes"
fmt.Println("Number of bytes:",
  len(name))
fmt.Println("Number of characters:",
  len([]rune(name)))
```

```
Number of bytes: 15
Number of characters: 13
```

Built-in types: Booleans

- Literals: **true** or **false**
- **if** statement only accepts booleans.
- No concept of truthy or false values
- No (x ? “yes” : “no”) operator - reason: code readability

Operators:

&& **||** **!**

Built-in Types: Arrays

- Size fixed with declaration
- Example: **var a [5]int** ⇨ array of 5 integers
- Length is part of type! Eg: **[5]int** and **[6]int** are not compatible
- Indexing starts at 0
- **len()** gives the size of the array

Built-in Types: Example of array

```
//let the complier calculate the size
names:=[...]string{"Joe","Jane","Bill"}
names[1]="Janet"
fmt.Println(names)
i:=3
names[i]="Earl"
```

```
[Joe Janet Bill]
panic: runtime error: index out of
range
```

Built-in types: Slices

- Slice = flexible array
- Example: **var a [ ]int**
- Has length and capacity; can be changed
- Need to be initialized before use:

  a:=make([ ]int,5,10)
- Passed by reference!
- Access past length causes runtime panic

Built-in Types: Example of slice

```
func buggy(mynames []string) {
    mynames[1]="Janet"
}

…

names:=[]string{"Joe","Jane","Bill"}
buggy(names)
fmt.Println(names)
```

```
[Joe Janet Bill]
```

Built-in Types: Maps

- Map = Associative array
- Can be used with almost any type of key
- Must be initialized: **make(map[int]string)**
- Basic functions:
  - len() ⇨ number of elements in the map
  - delete() ⇨ remove an element
- Passed by reference!

Built-in Types: Example of map

```
en2hu:=make(map[string]string)
en2hu["shoe"]="cipő"
en2hu["health"]="egészség"
for en, hu := range en2hu {
  fmt.Printf(`"%v" translates to "%v"`+
    "\n",en,hu)
}
```

```
"shoe" translates to "cipő"
"health" translates to "egészség"
```

Built-in Types: Structs

- Composite type made up of other types
- Reference: a.name=”Joe”
- Literal:
  - a:=person{“Joe”,35}
  - a:=person{age: 35,
    name: “Joe”}
    (preferred)

```
type person struct{
   name    string
   age     int
}
```

Built-in Types: Example of struct

```
type person struct {
    name string
    age  int
}

func printPerson(who person) {
    fmt.Println(who.name, "is",
      who.age, "yrs old.")
}
```

```
func main() {
    joe := person{name:"Joe"}
    printPerson(joe)
}
```

```
Joe is 0 yrs old.
```

Built-in types: pointers

- Pointer = reference to a memory address of a variable
- Eg: **var a *int**
- Getting a pointer to variable: **a := &b**
- Dereferencing: ***a = 2**
- No pointer arithmetic!
  Use the **unsafe** package for that
- Automatic dereferencing for struct fields

Built-in Types: Example of pointer

```
type person struct {
    name string
    age  int
}
func birthday(who *person) {
  who.age++
      //automatic dereferencing
  fmt.Println(who.name,
   "is now", who.age,
   "yrs old.")
}
```

```
func main() {
   joe := person{name:"Joe"}
   birthday(&joe)
}
```

```
Joe is now 1 yrs old.
```

Built-in types: functions

- Functions are values in Go
- Can be assigned to a variable and passed to a function
- Allow writing flexible & reusable code

Example: function as value

```
type operation func(int,int) int

func calc(num1, num2 int,
   op operation) int {
   return operation(num1, num2)
}
```

```
func add(a, b int) int {
  return a + b
}

func main() {
  fmt.Println(calc(2,2,add))
}
```

```
4
```

Built-in types: Interfaces

- Interface = collection of method signatures
- Any type that has the need methods satisfies the interface, regardless of underlying data structure
- Allow creating modular, highly reusable code
- Makes testing easy with mock types & methods
- Empty interface: `interface{}` matches any type

This is one of the most important features of Go and deserves a full presentation

Example: function as value

```
type storage interface {
  Read() []byte
  Write([]byte)
}

// an implementation
// that is compatible with the
// `storage` interface
type file struct{
  handle  int
  open    bool
}
```

```
func (f file) Read() []byte{
  …
}

func (f file) Write([]byte) {
  …
}
```

Built-in types: channels

- Special data type that allows communication and data exchange between goroutines
- defining a channel: `var A chan int`
- creating a channel:

  `A=make(chan int, _buffer_size_ )`
- Using unitialized channel will cause runtime panic
- Channel operator: **<-**
- Send: `A <- 12`
- Receive: `x= <-A`

Built-in Types: Example of channel

```
var out chan int

func process(num int) {
  out <- num * 2
}

func main() {
  out = make(chan int, 5)
```

```
  for i := 0; i < 5; i++ {
    go process(i)
  }
  //do some other stuff
  //...

  for i := 0; i < 5; i++ {
    fmt.Println(<-out)
  }
}
```

```
0
2
4
6
8
```

Types passed by reference & nil

- **pointer**, **channel**, **func**, **interface**, **map** and **slice** are reference types & passed by reference
- **nil** is a null pointer
- **nil** can be assigned to (and only to) the above types

Type conversions

- Type conversion has to be explicit.
- The syntax is: _target_type_ (_other_type_var_)

Example:

```
var a int=32
b:=float32(a)    //b is now 32.0
```

- This works only between similar types
- Conversion between other type (eg: string and int) requires using a package (eg: *strconv*)

Constants and types

- Constants don’t have to have a specific type
- The type gets figured out at the time of assignment

Example:

```
const A = 32
var x int = A   //same as if you typed `var x int = 32`
var y float32 = A  //same as if you types `var y float32 = 32`
//exactly typed constant:
const B = float32(12)
var z int = B  //will produce compiler error
```

The error type

- Error is a predefined type of:

```
type error interface {
    Error() string
}
```

- You can define your own error implementation
- **nil** value signifies “no error”