Ultimate GoLang Reference Card - basics

September 23, 2020 by Mateusz Urbanek

Hello World

```go
package main

import "fmt"

func main() {
  message := greetMe("world")
  fmt.Println(message)
}

func greetMe(name string) string {
  return "Hello, " + name + "!"
}
```

Variables

Variable definition and declaration (inside function body):

```go
var msg string
msg = "hello"
```

Variable (infers type):

```go
msg := "hello"
```

Variable declaration (outside function body):

```go
var msg = "hello"
```

Constants

```go
const X = 2.22
```

Strings

Strings are of type string.

```go
str := "Hello"
```

```go
str := ‘Multiline
       string‘
```

Numbers

Typical types:

```go
num := 3            // int
num := 3.           // float64
num := 3 + 4i       // complex128
num := byte(’a’)    // byte (alias
    for uint8)
```

Other common types:

```go
var u uint = 7          // uint (
    unsigned)
var p float32 = 22.7    // 32-bit
    float
```

Arrays

Arrays have a fixed size.

```go
// var numbers [5]int
numbers := [...]int{0, 0, 0, 0, 0}
```

Slices

Slices have a dynamic size, unlike arrays.

```go
slice := []int{2, 3, 4}
slice := []byte("Hello")
```

Type conversions

```go
i := 2
f := float64(i)
u := uint(i)
```

Pointers

Pointers point to a memory location of a variable. Go is fully garbage-collected.

```go
func main () {
  b := *getPointer()
  fmt.Println("Value is", b)
}

func getPointer() (*int) {
  a := 234
  return &a
}
```

```go
a := new(int)
*a = 234
```

Ultimate GoLang Reference Card - flow control

September 23, 2020 by Mateusz Urbanek

Conditional

If, else if, else:

```go
if day == "sun" || day == "sat" {
  rest()
} else if day == "mon" && isTired() {
  groan()
} else {
  work()
}
```

Statements in if

A condition in an if statement can be preceded with a statement before a ;. Variables declared by the statement are only in scope until the end of the if.

```go
if _, err := doThing(); err != nil {
  fmt.Println("Uh oh")
}
```

For loops

```go
for count := 0; count <= 10; count++ {
  fmt.Println("My counter is at", count)
}
```

```go
condition := 0

for condition < 5 { // while
    condition++
}
```

```go
condition := 0

for { // while true
    condition++
    if condition == 5 {
        break
    }
}
```

For-Range loops

```go
entry := []string{"Jack","John","Jones"}

for i, val := range entry {
  fmt.Printf("At position %d," +
             " the character %s" +
             " is present\n",
             i, val)
}
```

Switch

```go
switch day {
  case "sunday":
    // cases don't "fall through" by default!
    fallthrough

  case "saturday":
    rest()

  default:
    work()
}
```

Ultimate GoLang Reference Card - packages

September 23, 2020 by Mateusz Urbanek

Importing

Both are the same.

```go
import "fmt"
import "math/rand"
```

```go
import (
  "fmt"          // gives fmt.Println
  "math/rand"    // gives rand.Intn
)
```

Aliases

```go
import (
  r "math/rand"
  "fmt"
)

func main() {
  ri := r.Intn()
  fmt.Println(ri)
}
```

Packages

Every package file has to start with package.

```go
package hello
```

Exporting names

Exported names begin with capital letters.

```go
func Hello () {
  ...
}
```

Unexported names

Package private names begin with small letters.

```go
package unexp

func hello() string {
  return "hello"
}
```

```go
1 import (
2   "unexp"
3   "fmt"
4 )
5
6 func main () {
7     fmt.Printf("%s", unexp.hello())
8     // ./main.go:7: cannot refer to unexported name unexp.hello
9     // ./main.go:7: undefined: unexp.hello
10 }
```

Important packages

fmt	Package fmt implements formatted I/O with functions analogous to C's printf and scanf.
bytes	Package bytes implements functions for the manipulation of byte slices. It is analogous to the facilities of the strings package.
bufio	Package bufio implements buffered I/O. It wraps an io.Reader or io.Writer object, creating another object (Reader or Writer) that also implements the interface but provides buffering and some help for textual I/O.
io	Package io provides basic interfaces to I/O primitives. Its primary job is to wrap existing implementations of such primitives, such as those in package os, into shared public interfaces that abstract the functionality, plus some other related primitives.
ioutil	Package ioutil implements some I/O utility functions.
log	Package log implements a simple logging package. It defines a type, Logger, with methods for formatting output.
flag	Package flag implements command-line flag parsing.
context	Package context defines the Context type, which carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes.
sync	Package sync provides basic synchronization primitives such as mutual exclusion locks. Other than the Once and WaitGroup types, most are intended for use by low-level library routines.

Ultimate GoLang Reference Card - functions, structs and methods

September 23, 2020 by Mateusz Urbanek

Functions - Lambdas

Functions are first class objects.

```go
myfunc := func() bool {
  return x > 10000
}
```

Functions - Multiple return types

```go
func main() {
    a, b := getMessage()
}

func getMessage() (a string, b string) {
  return "Hello", "World"
}
```

Functions - Named return values

By defining the return value names in the signature, a `return` (no args) will return variables with those names.

```go
func split(sum int) (x, y int) {
  x = sum * 4 / 9
  y = sum - x
  return
}
```

Structs - Defining

```go
type Vertex struct {
  X int
  Y int
}

func main() {
  v := Vertex{1, 2}
  v.X = 4
  fmt.Println(v.X, v.Y)
}
```

Structs - Literals

You can also put field names.

```go
v := Vertex{X: 1, Y: 2}
```

```go
// Field names can be omitted
v := Vertex{1, 2}
```

```go
// Y is implicit
v := Vertex{X: 1}
```

Structs - Pointers to structs

Doing v.X is the same as doing (*v).X, when v is a pointer.

```go
v := &Vertex{1, 2}
v.X = 2
```

Methods - Receivers

There are no classes, but you can define functions with receivers.

```go
type Vertex struct {
  X, Y float64
}
func (v Vertex) Abs() float64 {
  return math.Sqrt(v.X * v.X + v.Y * v.Y)
}

func main() {
    v := Vertex{1, 2}
    v.Abs()
}
```

Methods - Mutation

By defining your receiver as a pointer (`*Vertex`), you can do mutations.

```go
func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func main() {
    v := Vertex{6, 12}
    v.Scale(0.5)
    // ‘v‘ is updated
}
```

Ultimate GoLang Reference Card - interfaces and error control

September 23, 2020 by Mateusz Urbanek

Interfaces - A basic interface

```go
type Shape interface {
  Area() float64
  Perimeter() float64
}
```

Interfaces - Struct

Struct Rectangle implicitly implements interface Shape by implementing all of its methods.

```go
type Rectangle struct {
  Length, Width float64
}
```

Interfaces - Methods

The methods defined in Shape are implemented in Rectangle.

```go
func (r Rectangle) Area() float64 {
  return r.Length * r.Width
}

func (r Rectangle) Perimeter() float64 {
  return 2 * (r.Length + r.Width)
}
```

Interfaces - Interface example

The methods defined in Shape are implemented in Rectangle.

```go
func main() {
  var r Shape = Rectangle{Length: 3, Width: 4}
  fmt.Printf("Type of r: %T, Area: %v, Perimeter: %v.",
             r, r.Area(), r.Perimeter())
}
```

Error control - Defer

Defers running a function until the surrounding function returns. The arguments are evaluated immediately, but the function call is not ran until later.

```go
func main() {
  defer fmt.Println("Done")
  fmt.Println("Working...")
}
```

Error control - Deferring lambdas

Lambdas are better suited for defer blocks.

```go
func main() {
  defer func() {
    fmt.Println("Done")
  }()
  fmt.Println("Working...")
}
```

Error control - Deferring lambdas w/params

The defer func uses current value of d, unless we use a pointer to get final value at end of main.

```go
func main() {
  var d = int64(0)
  defer func(d *int64) {
    fmt.Printf("& %v Unix Sec\n",
               *d)
  }(&d)
  fmt.Print("Done ")
  d = time.Now().Unix()
}
```

Error control - error return value

Error is a built-in type in Go and its zero value is nil. An idiomatic way to handle an error is to return it as the last return value of a function call and check for the nil condition.

```go
val, err := myFunction(arg1, arg2);
if err != nil {
  // handle error
} else {
  // success
}
```

Error control - custom errors

Custom error is struct implementing the error interface and Error method.

```go
import "fmt"

type MyError struct{}

func (myErr *MyError) Error() string {
  return "Something unexpected happend!"
}

func main() {
  myErr := &MyError{}

  fmt.Println(myErr)
}
```

Error control - error formatting

```go
err := fmt.Errorf("user %q (id %d) not found", name, id)
```

```go
myErr := errors.New("Something unexpected happend!")
```

Ultimate GoLang Reference Card - concurrency

September 23, 2020 by Mateusz Urbanek

Goroutines

Channels are concurrency-safe communication objects, used in goroutines.

```go
func main() {
  // A "channel"
  ch := make(chan string)

  // Start concurrent routines
  go push("Moe", ch)
  go push("Larry", ch)
  go push("Curly", ch)

  // Read 3 results
  // (Due to concurrency
  // the order isn't guaranteed!)
  fmt.Println(<-ch, <-ch, <-ch)
}

func push(name string, ch chan string) {
  msg := "Hey, " + name
  ch <- msg
}
```

Mutex

We can define a block of code to be executed in mutual exclusion by surrounding it with a call to Lock and Unlock as shown on the Inc method.

```go
import "sync"

type SafeCounter struct {
  v   map[string]int
  mux sync.Mutex
}

func (c *SafeCounter) Inc(key string) {
  c.mux.Lock()
  c.v[key]++
  c.mux.Unlock()
}
```

WaitGroup

A WaitGroup waits for a collection of goroutines to finish. The main goroutine calls Add to set the number of goroutines to wait for. The goroutine calls wg.Done() when it finishes. See: WaitGroup

```go
import "sync"

func main() {
  var wg sync.WaitGroup

  for _, item := range itemList {
    // Increment WaitGroup Counter
    wg.Add(1)
    go doOperation(item)
  }
  // Wait for goroutines to finish
  wg.Wait()

}

func doOperation(item string) {
  defer wg.Done()
  // do operation on item
  // ...
}
```

Buffered channels

Buffered channels limit the amount of messages it can keep.

```go
ch := make(chan int, 2)
ch <- 1
ch <- 2
ch <- 3
// fatal error:
// all goroutines are asleep - deadlock!
```

Closing channels

Close channel:

```go
ch <- 1
ch <- 2
ch <- 3
close(ch)
```

Iterate across a channel until its closed

```go
for i := range ch {
  ...
}
```

Closed if ok == false

```go
v, ok := <- ch
```

Context

Package context defines the Context type, which carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes.

```go
func main() {
  http.HandleFunc("/", handler)
  log.Fatal(http.ListenAndServe("127.0.0.1:80", nil))
}

func handler(w http.ResponseWriter, r *http.Request) {
  ctx := r.Context()
  select {
    case <-time.After(5 * time.Second):
      fmt.Fprintln(w, "hello")
    case <-ctx.Done():
      err := ctx.Err()
      http.Error(w, err.Error(), 404)
  }
}
```

Ultimate GoLang Reference Card - string format options

September 23, 2020 by Mateusz Urbanek

General:

%v	the value in a default format when printing structs, the plus flag (%+v) adds field names
%#v	a Go-syntax representation of the value
%T	a Go-syntax representation of the type of the value
%%	a literal percent sign; consumes no value

Boolean:

%t	the word true or false

Integer:

%b	base 2
%c	the character represented by the corresponding Unicode code point
%d	base 10
%o	base 8
%O	base 8 with 0o prefix
%q	a single-quoted character literal safely escaped with Go syntax.
%x	base 16, with lower-case letters for a-f
%X	base 16, with upper-case letters for A-F
%U	Unicode format: U+1234; same as ”U+%04X”

Floating-point and complex constituents:

%b	decimalless scientific notation with exponent a power of two, in the manner of strconv.FormatFloat with the ’b’ format, e.g. -123456p-78
%e	scientific notation, e.g. -1.234456e+78
%E	scientific notation, e.g. -1.234456E+78
%f	decimal point but no exponent, e.g. 123.456
%F	synonym for %f
%g	%e for large exponents, %f otherwise. Precision is discussed below.
%G	%E for large exponents, %F otherwise
%x	hexadecimal notation (with decimal power of two exponent), e.g. -0x1.23abcp+20
%X	upper-case hexadecimal notation, e.g. -0X1.23ABCP+20

String and slice of bytes (treated equivalently with these verbs):

%s	the uninterpreted bytes of the string or slice
%q	a double-quoted string safely escaped with Go syntax
%x	base 16, lower-case, two characters per byte
%X	base 16, upper-case, two characters per byte

Slice:

%p	address of 0th element in base 16 notation, with leading 0x

Pointer:

%p	base 16 notation, with leading 0x

The %b, %d, %o, %x and %X verbs also work with pointers, formatting the value exactly as if it were an integer.

The default format for %v is:

bool:	%t
int, int8 etc.:	%d
uint, uint8, etc.:	%d, %#x if printed with %#v
float32, complex64, etc.:	%g
string:	%s
chan:	%p
pointer:	%p

For compound objects, the elements are printed using these rules, recursively, laid out like this:

struct:	field0 field1 ...
array, slice:	[elem0 elem1 ...]
maps:	map[key1:value1 key2:value2 ...]
pointer to above:	&, &[], &map[]

Width is specified by an optional decimal number immediately preceding the verb. If absent, the width is whatever is necessary to represent the value. Precision is specified after the (optional) width by a period followed by a decimal number. If no period is present, a default precision is used. A period with no following number specifies a precision of zero. Examples:

%f	default width, default precision
%9f	width 9, default precision
%.2f	default width, precision 2
%9.2f	width 9, precision 2
%9.f	width 9, precision 0

Ultimate GoLang Reference Card - data types

September 23, 2020 by Mateusz Urbanek

Boolean types They are boolean types and consists of the two predefined constants: (a) true (b) false

Numeric types They are again arithmetic types and they represents a) integer types or b) floating point values throughout the program.

String types A string type represents the set of string values. Its value is a sequence of bytes. Strings are immutable types that is once created, it is not possible to change the contents of a string. The predeclared string type is string.

Derived types They include:

a) Pointer types,
b) Array types,
c) Structure types,
d) Union types,
e) Function types,
f) Slice types,
g) Interface types,
h) Map types,
i) Channel Types.

Integer Types predefined architecture-independent integer types

type	*specification*	*size*
uint8	Unsigned 8-bit integers	0 to 255
uint16	Unsigned 16-bit integers	0 to 65535
uint32	Unsigned 32-bit integers	0 to 4294967295
uint64	Unsigned 64-bit integers	0 to 18446744073709551615
int8	Signed 8-bit integers	-128 to 127
int16	Signed 16-bit integers	-32768 to 32767
int32	Signed 32-bit integers	-2147483648 to 2147483647
int64	Signed 64-bit integers	-9223372036854775808 to 9223372036854775807

Floating Types predefined architecture-independent float types

type	*specification*
float32	IEEE-754 32-bit floating-point numbers
float64	IEEE-754 64-bit floating-point numbers
complex64	Complex numbers with float32 real and imaginary parts
complex128	Complex numbers with float64 real and imaginary parts

Other Numeric Types a set of numeric types with implementation-specific sizes

type	*specification*
byte	same as uint8
rune	same as int32
uint	32 or 64 bits
int	same size as uint
uintptr	an unsigned integer to store the uninterpreted bits of a pointer value