Hello World

```
package main
   (these lines are important)
import “fmt”
   (but omitted on this sheet)
func main() {
  fmt.Println(“Hello World”)
} → Hello World
```

For Loop

```
for i:=0; i<10; i++ {
  sum += i
}
```

```
sum:=1
for ; sum<1000 ; {
  sum += sum
} → for sum<1000 {
```

```
for {
  fmt.Println(“infinite ∞”)
} → endless loop
```

if

```
if x < 0 {
  fmt.Println(“neg”)
}
```

```
if x:=y+1; x < 0 {
  fmt.Println(“%d is neg”,x)
} → x in scope if{} only
```

```
if x:=y+1; x < 0 {
  fmt.Println(“%d is neg”,x)
} else {
  fmt.Println(“%d >= 0”,x)
} → else is part of if{}
```

Arrays

```
var arr [10]int

func man() {
  for i := range arr {
    arr[i] = I
  } (range is foreach in go)

  fmt.Println(arr)
} → [0 1 2 3 4 5 6 7 8 9]
```

Slices ⇓

```
  var s []int = arr[1:4]
  fmt.Println(s)
    → [1 2 3]

  s[0] = -1
  fmt.Println(arr)
} → [0 -1 2 3 4 5 6 7 8 9]
```

The array slice is [Y:Z]
Y → first element included
Z → last element noted
(Z is excluded)

Functions

```
func addsub(x, y int) (int,int) {
  return x + y, x-y
}

func main() {
  fmt.Println(add(2,3))
} → 5 -1
```

```
func addsub(x, y int) (a,s int) {
  a = x + y
  b = x – y
  return
}
```

Switch

```
switch x := y+1; x {
  case x < 0:
    fmt.Println(“%d is neg”,x)
  case 0:
    fmt.Println(“%d is zero”,x)
  default:
    fmt.Println(“%d is pos”,x)
} → x in scope switch{} only
```

```
switch {
  case x < 0:
    fmt.Println(“%d is neg”,x)
  case x < 1:
    fmt.Println(“%d is zero”,x)
  default:
    fmt.Println(“%d is pos”,x)
} → switch is an if-chain!
```

Defer

```
func main() {
  for i := 0; i < 10; i++ {
    defer fmt.Print(i, ” “)
  }
} → preprocess, stack execute

→ 9 8 7 6 5 4 3 2 1 0
```

Structs

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

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

Declarations

```
var i int (→ i=0)
var j,k int = 1,2
l := 3 (implicit type)
m,n,s := 4, 5, ”str”
```

Types

```
int  int8  int16
int32  int64 → 0

uint  uint8  uint16
uint32  uint64
uintptr  → 0

byte ↔ uint8
rune ↔ int32 (unicode)

float32  float64 → 0

complex64  complex128 → 0

bool   → false
string → “”
```

Type conversion

```
var i int = 42
f := float64(i)
u := uint(f)
```

Constants

```
const Pi = 3.1415926535897
```

Pointers

```
var p *int → nil
i := 42
p = &i

*int → pointer to an int
&i   → the address of I
p=&i → address ⇒ ptr
```

*struct

```
func main() {
  v := Vertex{}
  p := &v
  p.X = 100
  fmt.Println(v)
} → {100 0}
```

nommed from tour.golang.org

More on Slice bounds

```
s := []int{0,1,2,3,4,5,6,7}
s = s[1:6]
fmt.Println(s)
    → [1 2 3 4 5]
s = s[:4]
fmt.Println(s)
    → [1 2 3 4]
s = s[1:]
fmt.Println(s)
    → [2 3 4]
```

Slice recovery ⇓

```
fmt.Printf(%d %d %v, len(s),
cap(s), s)
    → 3 6 [2 3 4]
s = s[:6]
fmt.Printf(%d %d %v, len(s),
cap(s), s)
    → 6 6 [2 3 4 5 6 7]
```

We cannot recover earlier in the array. Slices are a pointer to an array with a record of their max size. No pointer backward exists.

slices being pointers, this slice is **nil**:

```
var myslice []int
```

Slice make

```
a := make([]int, 5)
fmt.Printf(%d %d %v, len(s),
cap(s), sl)
    → 5 5 [0 0 0 0 0]
```

This slice points to a *new* zero array, length 5

```
b := make([]int, 0, 5)
fmt.Printf(%d %d %v, len(s),
cap(s), sl)
    → 0 5 []
```

And this one to another, length 0, cap 5

Slice append

```
var s []int  → nil
s = append(s,0)
s = append(s,1,2,3,4)
fmt.Printf(%d %d %v, len(s),
cap(s), sl)
    → 5 5 [0 1 2 3 4]
```

Appending ***copies*** the array values if the slice needs to be extended. This can be a positive: to free memory of a much larger array, or negative: as copying is expensive

Range

```
s := make([]int, 3)
for index, value := range s {
  fmt.Printf("%d is %d; ", index,
value)
} → 0 is 0; 1 is 0; 2 is 0;
```

```
for index := range s {
  s[index] = index
}
for _, value := range s {
  fmt.Printf("%d ", value)
} → 0 1 2
```

Maps

```
var m map[string]string
func main() {
  m = make(map[string]string)
  m["key1"] = "value1"
  m["key2"] = "value2"
  fmt.Println(m)
  → map[key1:value1 key2:value2]
```

```
  delete(m, "key1")
  fmt.Println(m)
  → map[key2:value2]
  v,ok := m["key1"]
  fmt.Println("val:",v,"ok?",ok)
  → val:  ok? False
  _,ok := m["key2"]
  if ok { fmt.Println("good!")
  → good!
```

Interfaces

```
type Pet interface {
  Nom(string) string
}
type Dog struct {}
func (p Dog) Nom(f string) string
  return "wolfs " + f
}
type Cat struct {}
func (p Cat) Nom(f string) string {
  return "ignores " + f
}
type Rabbit struct {}
func (p Rabbit) Nom(f string) stri
  switch f {
    case "carrot":
      fallthrough
    case "lettuce":
      return "crunches " + f
    default:
      return "THUMP!!"
  }
}
```

All three implement Pet

Interfaces Continue

```
func main() {
  pets := []Animal{Dog{},
Cat{}, Rabbit{}}

  for _, p := range pets {
    fmt.Println(p.Nom(
"kibble"))
  }
}
→ wolfs kibble
  ignores kibble
  THUMP!!
```

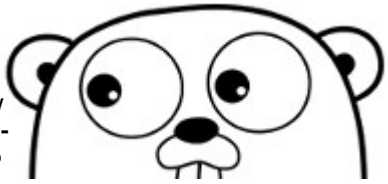
Type conversion

Constants

Pointers

*struct

also nommed:
http://jordanorelli.com/post/32665860244/how-to-use-interfaces-in-go



(You should nom it too)