BUILT-IN TYPES

bool, string,
int, int8, int16, int32, int64,
uint, uint8, uint16, uint32, uint64, uintptr,
rune, byte,
float32, float64,
complex64, complex128

VARIABLES

```
var ninja = "Johnny"
var level, yoe int = 1, 2
var isSkilled bool
weapon := "Ninja Star"

fmt.Println(ninja, level, yoe, isSkilled, weapon)
// Johnny 1 2 false Ninja Star
```

CONSTANTS

```
const dojo string = "Golang Dojo"
const powerLevel = 9001

const opLevel = 3e20
// a numeric constant has no type
// until it's given one
fmt.Printf("%T\n", opLevel)
```

LOOPS

```
isSkilled := true
for isSkilled {
  fmt.Println("Ready for mission!")
  isSkilled = false
}

for level := 7; level < 9; level++ {
  fmt.Println(level)
  fmt.Println("Leveling up!")
}

for {
  fmt.Println("I'm a Golang Ninja")
  break
}
```

SWITCH

```
weapon := "Ninja Star"
switch weapon {
case "Ninja Star":
  fmt.Println("It's a Ninja Star!")
case "Ninja Sword":
  fmt.Println("It's a Ninja Sword!")
}

powerLevel := 9001
switch {
case powerLevel > 9000:
  fmt.Println("It's over...NINE THOUSAND!!!")
default:
  fmt.Println("It's a Baby Ninja")
}
```

ARRAYS

```
// an array is a numbered sequence
// of elements of a specific length
var evilNinjas [3]string
fmt.Println(len(evilNinjas))

evilNinjas[0] = "Johnny"
fmt.Println(evilNinjas)
fmt.Println(evilNinjas[0])
fmt.Println(len(evilNinjas))

moreEvilNinjas := [3]string{"Andy", "Tommy", "Bobby"}
fmt.Println(moreEvilNinjas)

var missionRewards [2][3]int
for i := 0; i < 2; i++ {
  for j := 0; j < 3; j++ {
  missionRewards[i][j] = i + j
  }
}
```

SLICES

```
// a slice, on the other hand, doesn't need
// to be given a specific length
var evilNinjas []string
fmt.Println(len(evilNinjas))
evilNinjas = append(evilNinjas, "Tommy")
fmt.Println(len(evilNinjas))
```

MAPS

```
// to create an empty map, use the built-in make
ninjaLevels := make(map[string]int)
ninjaLevels["Johnny"] = 7
ninjaLevels["Tommy"] = 13
fmt.Println(ninjaLevels)
fmt.Println(len(ninjaLevels))

fmt.Println(len(ninjaLevels))
delete(ninjaLevels, "Johnny")
fmt.Println(len(ninjaLevels))

// the optional second return value when getting
// a value from a map indicates if the key was
// present in the map
_, ok := ninjaLevels["Tommy"]
fmt.Println(ok)

// another option of initializing maps
moreNinjaLevels := map[string]int{"Bobby": 8, "Andy": 3}
fmt.Println(moreNinjaLevels)
```

RANGE

```
evilNinjas:= []string{"Tommy", "Johnny", "Andy"}
for index, evilNinja := range evilNinjas{
  fmt.Println("Attacking target", index, evilNinja)
}

evilNinjasWithLevels:= map[string]int{"Tommy": 2}
for evilNinja, level := range evilNinjasWithLevels {
    fmt.Printf("%s -> %d\n", evilNinja, level)
}
```

POINTERS

```
type ninja struct {
   name string
}

func main() {
  tommy := ninja{"Tommy"}
  tommyPointer := &tommy
  johnnyPointer := &ninja{"Johnny"}
  var ninjaPointer *ninja = new(ninja)
}
```

STRUCTS

```
type ninja struct {
   name string
   level int
}

func main() {

  fmt.Println(ninja{name: "Bobby", level: 20})

  fmt.Println(ninja{name: "Andy", level: 30})

// omitted fields will be zero-valued
  fmt.Println(ninja{name: "Johnny"})

  tommy := ninja{name: "Tommy", level: 50}
  fmt.Println(tommy.level)

  tommy.level = 51
}
```

INTERFACE

```
type ninjaWeapon interface{
  attack()
}

type ninjaStar struct{}

func(n ninjaStar) attack() {
  fmt.Println("Throwing Ninja Star")
}

type ninjaSword struct{}

func(n ninjaSword) attack() {
  fmt.Println("Throwing Ninja Sword")
}

func main() {
  weapons := []ninjaWeapon{
    ninjaStar{},
    ninjaSword{},
  }
  for _, weapon := range weapons {
    weapon.attack()
  }
}
```

FUNCTIONS

```
func useWeapon(ninja string, weapon string) string {
  return fmt.Sprintf(ninja + "is using" + weapon)
}

// multiple return values
func isValidLevel(level int) (int, bool) {
  if level > 10 {
    return level, true
  }
  return level, false
}

// variadic functions
func attack(evilNinjas ...string) {
  for _, evilNinja:= range evilNinjas{
    fmt.Println("Attacking target", evilNinja)
  }
}

func main() {
  usage := useWeapon("Tommy", "Ninja Star")
  level, valid := isValidLevel(11)

  fmt.Println(usage, level, valid)

  attack("Tommy", "Johnny")
  attack("Tommy", "Johnny", "Andy", "Bobby")

  // if you already have multiple args in a slice,
  // apply them to a variadic function
  // using func(slice...)
  evilNinjas:= []string{"Tommy", "Johnny", "Andy"}
  attack(evilNinjas...)

  // closures
  attackToo := attack
  attackToo(evilNinjas...)
  func() {
    fmt.Println("Attacking Evil Ninjas...")
  }()
}
```

GOROUTINES

```
func attack(target string) {
   fmt.Println("Throwing ninja stars at", target)
}

func main() {
  go attack("Tommy")
  time.Sleep(time.Second)
}
```

CHANNELS

```
func attack(target string, attacked chan bool) {
  time.Sleep(time.Second)
  fmt.Println("Throwing ninja stars at", target)
  attacked <- true
}

func main() {
  smokeSignal := make(chan bool)
  evilNinja := "Tommy"
  go attack(evilNinja, smokeSignal)
  fmt.Println(<-smokeSignal)

  // buffered channels
  moreSmokeSignal := make(chan bool, 1)
  moreSmokeSignal <- true
  fmt.Println(<-moreSmokeSignal)

  // closing channel to prevent deadlocks
  moreSmokeSignal <- true
  close(moreSmokeSignal)
  for message := range moreSmokeSignal {
    fmt.Println(message)
  }
}
```

FOLLOW US

www.programisci.eu