**ˇBasic syntax**

|  |  |  |
| --- | --- | --- |
| *// All files start with a package declaration***package** main*// Import statements, one package on each line***import** ( **"errors"** **"fmt"**)*// Main method will be called when the Go executable is run***func** main() { fmt.Println(**"Hello world!"**) basic() add(1, 2) divide(3, 4) loops() slices() maps() sharks()}*// Function declaration***func** basic() { *// Declare x as a variable, initialized to 0***var** x int *// Declare y as a variable, initialized to 2***var** y int = 2 *// Declare z as a variable, initialized to 4* *// This syntax can only be used in a function*z := 4 *// Assign values to variables*x = 1 y = 2 z = x + 2 \* y + 3 *// Print the variables; just use %v for most types*fmt.Printf(**"x = %v, y = %v, z = %v\n"**, x, y, z)}*// Function declaration; takes in 2 ints and outputs an int***func** add(x, y int) int { **return** x + y} |  | *// Function that returns two things; error is nil if successful***func** divide(x, y int) (float64, error) { **if** y == 0 { **return** 0.0, errors.New(**"Divide by zero"**) } *// Cast x and y to float64 before dividing***return** float64(x) / float64(y), nil}**func** loops() { *// For loop***for** i := 0; i < 10; i++ { fmt.Print(**"."**) } *// While loop*sum := 1 **for** sum < 1000 { sum \*= 2 } fmt.Printf(**"The sum is %v\n"**, sum)}**func** slices() { slice := []int{1, 2, 3, 4, 5, 6, 7, 8} fmt.Println(slice) fmt.Println(slice[2:5]) *// 3, 4, 5*fmt.Println(slice[5:]) *// 6, 7, 8*fmt.Println(slice[:3]) *// 1, 2, 3*slice2 := make([]string, 3) slice2[0] = **"tic"**slice2[1] = **"tac"**slice2[2] = **"toe"**fmt.Println(slice2) slice2 = append(slice2, **"tom"**) slice2 = append(slice2, **"radar"**) fmt.Println(slice2) **for** index, value := **range** slice2 { fmt.Printf(**"%v: %v\n"**, index, value) } fmt.Printf(**"Slice length = %v\n"**, len(slice2))}**func** maps() { myMap := make(**map**[string]int) myMap[**"yellow"**] = 1 myMap[**"magic"**] = 2 myMap[**"amsterdam"**] = 3 fmt.Println(myMap) myMap[**"magic"**] = 100 delete(myMap, **"amsterdam"**) fmt.Println(myMap) fmt.Printf(**"Map size = %v\n"**, len(myMap))} |

**Basic syntax**

|  |  |  |
| --- | --- | --- |
| *// Object oriented programming**// Convention: capitalize first letter of public fields***type** Shark **struct** { Name string Age int}*// Declare a public method**// This is called a receiver method***func** (s \*Shark) Bite() { fmt.Printf(**"%v says CHOMP!\n"**, s.Name)}*// Because functions in Go are pass by value**// (as opposed to pass by reference), receiver**// methods generally take in pointers to the**// object instead of the object itself.***func** (s \*Shark) ChangeName(newName string) { s.Name = newName}*// Receiver methods can take in other objects as well***func** (s \*Shark) Greet(s2 \*Shark) { **if** (s.Age < s2.Age) { fmt.Printf(**"%v says your majesty\n"**, s.Name) } **else** { fmt.Printf(**"%v says yo what's up %v\n"**, s.Name, s2.Name) }}**func** sharks() { shark1 := Shark{**"Bruce"**, 32} shark2 := Shark{**"Sharkira"**, 40} shark1.Bite() shark1.ChangeName(**"Lee"**) shark1.Greet(&shark2) *// pass in pointer*shark2.Greet(&shark1)} |  | *// Launch n goroutines, each printing a number**// Note how the numbers are not printed in order***func** goroutines() { **for** i := 0; i < 10; i++ { *// Print the number asynchronously***go** fmt.Printf(**"Printing %v in a goroutine\n"**, i) } *// At this point the numbers may not have been printed yet*fmt.Println(**"Launched the goroutines"**)}*// Channels are a way to pass messages across goroutines***func** channels() { ch := make(**chan** int) *// Launch a goroutine using an anonymous function***go func**() { i := 1 **for** { *// This line blocks until someone* *// consumes from the channel*ch <- i \* i i++ } }() *// Extract first 10 squared numbers from the channel***for** i := 0; i < 10; i++ { *// This line blocks until someone sends into the channel*fmt.Printf(**"The next squared number is %v\n"**, <-ch) }}*// Buffered channels are like channels except:**// 1. Sending only blocks when the channel is full**// 2. Receiving only blocks when the channel is empty***func** bufferedChannels() { ch := make(**chan** int, 3) ch <- 1 ch <- 2 ch <- 3 *// Buffer is now full; sending any new messages will block* *// Instead let's just consume from the channel***for** i := 0; i < 3; i++ { fmt.Printf(**"Consuming %v from channel\n"**, <-ch) } *// Buffer is now empty; consuming from channel will block*} |