Golang Tutorial

https://golangr.com

Google Go

Go is a computer programming language. It is sometimes called *“Google Go”* or *“golang”*. Go programs run on popular operating systems (Windows, OSX, Linux).

Introduction to Go

Go files

Go programs are stored in files. Computer systems contain millions of files. These files can be video files, images, sound and other types of data.

In the case of Go programs, they are stored in code files. A program consists of lines of text.

Each file can have an extension. An extension is the word that comes after the dot. If you have a file “beach.bmp”, the first part before the dot is the filename. The last part after the dot is the exension (bmp).

For Go programs, the extension is (.go).

Folders (or directories) are used to group files together. For each software project, you’ll want to have your files in a project folder.

Terminal

The terminal, sometimes called command line interface, is very often used by programmers. Typically this is a black and white screen which only displays text, on OSX its often white on black.

The programmer types commands, to which the computer reacts.

You can start Go programs from the terminal.

To open a terminal:

- **Windows**: hold the windows key and press r. Then type cmd.exe and hit the return key.
- **OSX**: Go to Finder -> Applications -> Utilities -> Terminal.

You can then navigate to your project folder with the cd command. On OSX that may be:

```
1 cd /Users/dog/Projects/hello
```

On Windows the folder system is different, it may be

```
1 cd C:\Users\dog\Projects\go
```

You can also navigate a single directory down (cd folder) or up (cd ..). This works on both systems. To see the files in a map, type “ls” or “dir”.

Run Go program

You can run Go programs from the terminal. If you have a program named “icecream.go”, you can start it with the command:

```
1 go run icecream.go
```

The output will be shown on the screen.

To run this command, Go must be installed.

Installing the Go programming language

The Go programming language was created in 2007 at Google. Many projects are created in Golang including Docker, Kubernets, Gogs, InfluxDB and others.

The Go language can be used on various operating systems including Linux systems, Windows, Mac OS X and FreeBSD. You can even run it online, from your browser.

Installing Go

If you have a package manager, you can install it from a repository.

Ubuntu / Debian Linux

Install a set of tools to run Go programs.

```
1 apt-get install golang-go
```

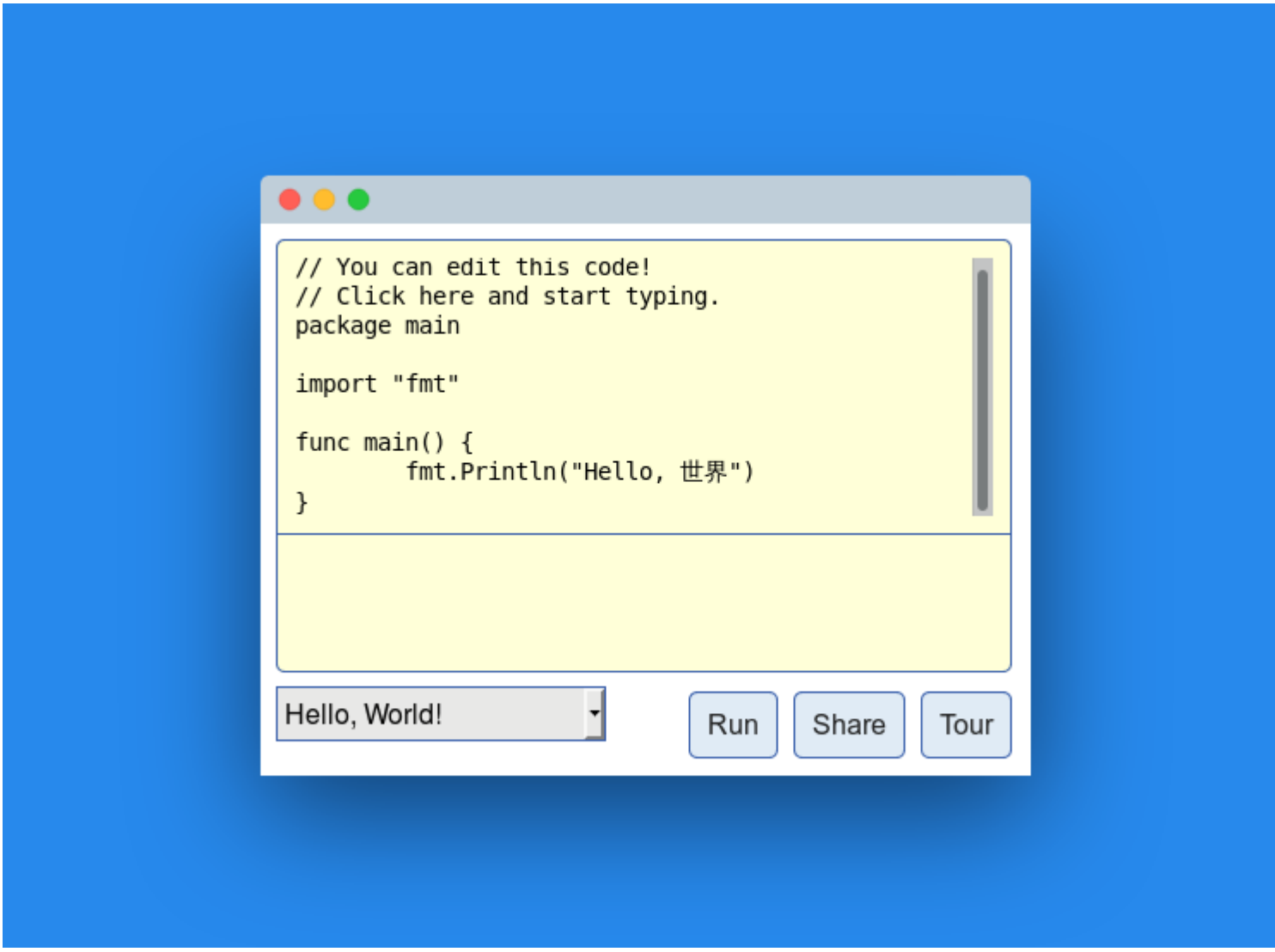
CentOS 7 / Redhat Linux

If you use Redhat or CentOS, you can use yum to install go

```
1 yum install golang golang-godoc golang-vet golang-src golang-pkg-linux-amd64 -y
```

Running go online

You can run golang programs online on the Go Playground.
On that page simply type your code and click run.



Manual install

For other systems, you can run a manual install.

Check version

To check the current version of Golang, you can use the command

```
1 [root@master ~]# go version
```

Hello World

Welcome to the Golang Tutorial! This is your very first program. There are exercises below the tutorials.

Quick Start

Create a hello world app

One of the easiest programs:
a program that displays “Hello world”.

Start by creating a new file (hello.go) with this contents:

```
1
2 package main
3 import "fmt"
4 func main() {
5    fmt.Println("Hello, World!")
6 }
7
```

We import the essentials. Then create the main function which is where the program starts. Finally we output the text “Hello world” with the line:

```
1 fmt.Println("Hello, World!")
```

Run app

To run a go program, open the command line. Move into the directory that has your program, like:

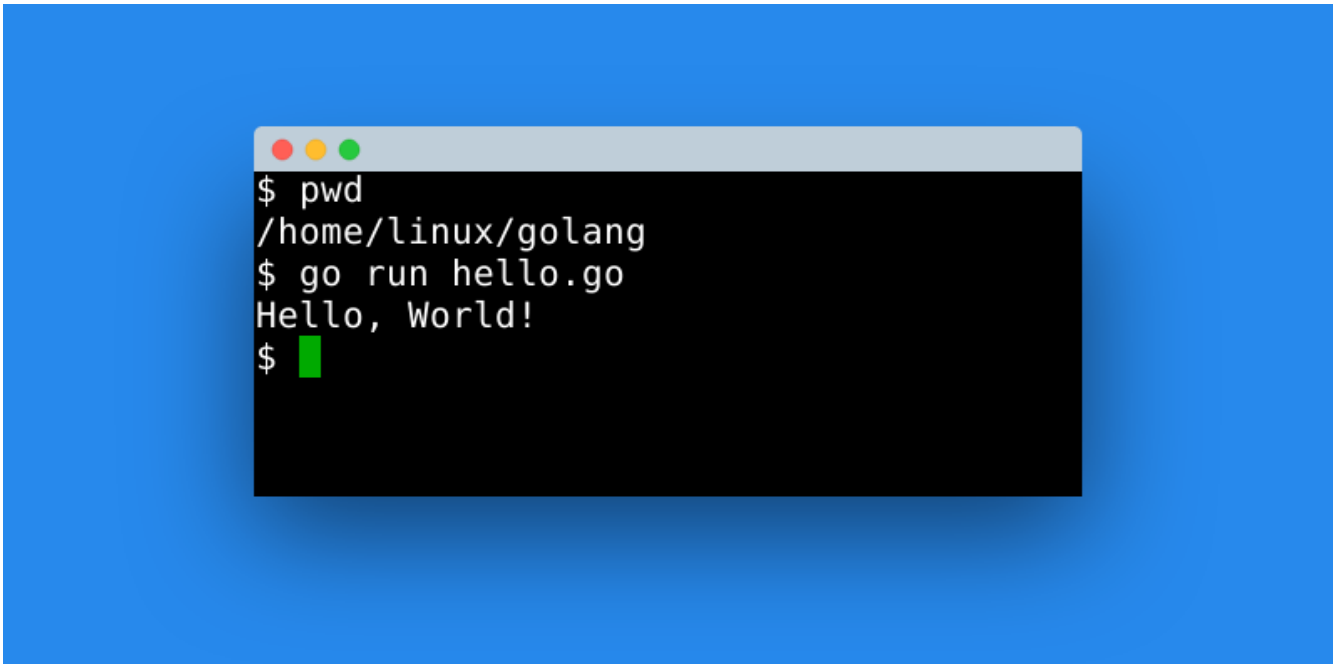
```
1 cd /home/tux/GoProjects/
```

You should now be in the directory that has your program (hello.go).

Run the program with the command below.

```
1 $ go run hello.go
```

This will then output the message “Hello, World!” to the screen



Exercise

1. Create a program that shows your name
2. Create a program that shows your address

Download Answers

Comments

golang supports comments. A comment is text that is ignored on execution. but may be useful to the programmer. You can add any kind of text inside your code.

golang ignores comments. Comments can be single line or multi line.

Quick Start

Comments in golang

Write some code. In this example we output some text.
Then add single and multi line comments.

Copy the code below and save the file as hello.XYZ

```
1
2
3  package main
4  import "fmt"
5  func main() {
6     /* This is a multi line comment.
7        You can type within this section */
8        fmt.Println("Go!")
9        // single line comment
10       fmt.Println("Lang!")
11  }
12
```

Exercises

1. Create a program and add a comment with your name

Download Answers

Strings

golang Strings can be seen as a collection of characters. A string can be of length 1 (one character), but its usually longer. A string is always written in double qoutes. This means a *string* variable in golang can hold *text*: words, sentences, books

Programming languages have variables. These variables have a *data type.* A variable can be of the data type string.

String example

String variable

In the example below we use golang to print text. First define a string variable, then print the variable.

```
package main
import "fmt"
func main() {
   var str1 =  "This is a string variable!"
   fmt.Println(str1)
}
```

Multiple lines

Generally there are 2 ways to print multiple lines.

- *Method 1.* call display function x times,
- *Method 2.* use the newline character inside the string

Both of these are common in programming.

```
package main
import "fmt"
func main() {
   var str1 =  "This is a string variable!"
   // 1: function calls
   fmt.Println(str1)
   fmt.Println(str1)
   // 2: use newline character
   fmt.Print("Hello World\n")

}
```

Exercises

1. Create a program with multiple string variables
2. Create a program that holds your name in a string.

Download Answers

Keyboard input

golang can read keyboard input from the console. In this section you will learn how to do that..

To get keyboard input, first open a console to run your program in. Then it will ask for keyboard input and display whatever you've typed.

Keyboard input in golang

Example

The golang program below gets keyboard input and saves it into a string variable. The string variable is then shown to the screen.

```
1  package main
2  import (
3      "bufio"
4      "fmt"
5      "os"
6  )
7  func main() {
8      reader := bufio.NewReader(os.Stdin)
9      fmt.Print("Enter your city: ")
10     city, _ := reader.ReadString('\n')
11     fmt.Print("You live in " + city)
12 }
13
```

The data that we read from the console (keyboard), is stored in a variable and printed to the screen. Sample output of program.

```
1 Enter city name: Sydney
2 You live in Sydney
```

This line will get keyboard input and store it in a variable:

```
1
2 # necessary
3 reader := bufio.NewReader(os.Stdin)
4 # read line from console
5 city, _ := reader.ReadString('\n')
```

You can get as many input variables as you want, simply by duplicating this line and changing the variable names.

Exercises

1. Make a program that lets the user input a name
2. Get a number from the console and check if it's between 1 and 10.

Download Answers

Variables

Variables often hold text or numeric data. In golang there are several types of variables, including strings and numeric variables.

Variables can be reused in your code. Arithmetic operations can be used on numeric variables. String variables can also be changed (sub-strings, concatenation).

Variables in golang

Numeric variables

Lets start with numeric variables. We create a program that calculates the VAT for a given price.

Define a series of products, sum the price ex. VAT, then calculate the VAT and add it to the price. Copy the code below and save the file as variables.XYZ

```
1
2
3  package main
4  import "fmt"
5  func main() {
6     apple := 3.0
7     bread := 2.0
8     price := apple + bread
9     fmt.Printf("")
10    fmt.Printf("Price:    %f",price)
11    vat := price * 0.15
12    fmt.Printf("VAT:      %f",vat)
13    total := vat + price
14    fmt.Printf("Total:    %f",total)
15    fmt.Printf("")
16
17 }
18
```

All arithmetic operations can be run on variables: division (/), substraction (-), addition (+) and multiplication (*)

Exercises

1. Calculate the year given the date of birth and age
2. Create a program that calculates the average weight of 5 people.

Download Answers

Scope

Scope is where a variable can be used. A variable can often be used inside a function, a `local variable`.

Sometimes a variable can be used everywhere in the program, a `global variable`.

There are also cases in which a variable only exists inside an statement or loop. These are also `local variables`.

Examples

Local vs global variables

The variable `x` below is a local variable, it can only be used inside the `main()` function.

```
package main
import "fmt"
func main() {
    x := 7
    fmt.Println(x)
}
```

If you move the variable x outside of the function, it becomes a global variable.

Global variables can be used by multiple functions. In this example `example()` and `main()`

```
package main
import "fmt"
var x = 7
func example() {
    fmt.Println(x)
}
func main() {
    fmt.Println(x)
    example()
}
```

Global variables are sometimes considered a bad practice.
When possible, you should pass variables as function parameters instead.

Exercises

- What's the difference between a local and global variable?
- How can you make a global variable?

Download Answers

Arrays

golang arrays can hold multiple values. The minimum elements of an array is zero, but they usually have two or more. Each element in an array has a unique index.

The index starts at zero (0). That means to access the first element of an array, you need to use the zeroth index.

Arrays in golang

Example

The program below is an example of an array loop in golang.
The array we define has several elements. We print the first item with the zeroth index.

```
1
2  package main
3  import "fmt"
4  func main() {
5     var a = []int64{ 1,2,3,4 }
6
7     fmt.Printf("First element %d", a[0] )
8     fmt.Printf("Second element %d", a[1] )
9  }
10
```

Upon running this program it will output the first (1) and second (2) element of the array. They are referenced by the zeroth and first index. In short: computers start counting from 0.

This program will output:

```
1 First element: 1
2 Second element: 2
```

The index should not be larger than the array, that could throw an error or unexpected results.

Exercises

1. Create an array with the number 0 to 10
2. Create an array of strings with names

Download Answers

For loops

golang can repeat a code block with a *for loop*. All for loops have a condition, this can be the amount of times or a list.

You need loops to repeat code: instead of repeating the instructions over and over, simply tell golang to do it n times.

For loops in golang

Example

The program below is an example of a for loop in golang. The for loop is used to repeat the code block. golang will jump out of the code block once the condition is true, but it won't end the program.

The code block can contain anything, from statements to function calls.

```
1
2 package main
3 import "fmt"
4 func main() {
5    for x := 0; x < 4; x++ {
6       fmt.Printf("iteration x: %d", x)
7    }
8 }
9
```

The code block can be as many lines as you want, in this example its just one line of code that gets repeated.

golang runs the code block only n times. The number of repetitions is called *iterations* and every round is called an iteration.

Exercises

1. Can for loops exist inside for loops?
2. Make a program that counts from 1 to 10.

Download Answers

Range

Range iterates over elements. That can be elements of an array, elements of a dictionary or other data structures.

When using range, you can name the current element and current index:
`for i, num := range nums {`.

But you are free to ignore the index: `for _, num := range nums {`.

Example

Range

Range is always used in conjunction with a data structure.
Thus, the first step is to create a data structure. Here we define a slice:

```
nums := []int{1,2,3,4,5,6}
```

Then iterate over it with range:

```
package main
import "fmt"
func main() {
    nums := []int{1,2,3,4,5,6}
    for _, num := range nums {
            fmt.Println(num)
    }
}
```

Index

You can use the index. This shows the current index in the data structure.
Instead of the underscore _, we named it index. That variable now contains the current index.

```
var a = []int64{ 1,2,3,4 }
for index, element := range a {
   fmt.Print(index, ") ", element,"\n")
}
```

Exercises

- What is the purpose of `range` ?
- What the difference between the line `for index, element := range a` and the line `for _, element := range a` ?

If statements

golang can make choices with data. This data (variables) are used with a condition: if statements start with a condition. A condition may be (x > 3), (y < 4), (weather = rain). What do you need these conditions for? Only if a *condition* is true, code is executed.

If statements are present in your everyday life, some examples:

- if (elevator door is closed), move up or down.
- if (press tv button), next channel

If statements in golang

Example

The program below is an example of an if statement.

```
1  package main
2  import "fmt"
3  func main() {
4     var x = 3
5     if ( x > 2 ) {
6        fmt.Printf("x is greater than 2");
7     }
8  }
9
```

golang runs the code block only if the condition (x >2) is true. If you change variable x to any number lower than two, its codeblock is not executed.

Else

You can execute a codeblock if a condition is not true

```
1  package main
2  import "fmt"
3  func main() {
4     var x = 1
5     if ( x > 2 ) {
6        fmt.Printf("x is greater than 2");
7     } else {
8        fmt.Printf("condition is false (x > 2)");
9     }
10 }
11
```

Exercises

1. Make a program that divides x by 2 if it’s greater than 0
2. Find out if if-statements can be used inside if-statements.

While loops

golang can repeat a code block with a *while loop*. The while loop repeats code until a condition is true.

While loops are used when you are not sure how long code should be repeated. Think of a tv that should continue its function until a user presses the off button.

While loops in golang

Example

The program below is an example of a while loop in golang.
It will repeat until a condition is true, which could be forever.

The code block can contain anything, from statements to function calls.

```
1
2
3  package main
4  import "fmt"
5  func main() {
6       i := 1
7       max := 20
8       // technically go doesnt have while, but
9       // for can be used while in go.
10      for i < max {
11          fmt.Println(i)
12          i += 1
13      }
14 }
15
16
```

In the example it repeats the code block until variable i is greater than max.
You must always increment the iterator (i), otherwise the while loop repeats forever.

The code block can be as many lines as you want, in this example its just one line of code that gets repeated.

Exercises

1. How does a while loop differ from a for loop?

Download Answers

File exists

There is a golang function that checks if a file exists. This function returns true if the file exists.

Why? If you try to open a file that doesn't exist, at best it will return an empty string and at worst it will crash your program. That would lead to unexpected results and thus you want to check if the file exists.

File exists in golang

Example

The following golang code will check if the specified file exists or not.

```
1  package main
2  import "fmt"
3  import "os"
4  func main() {
5    if _, err := os.Stat("file-exists.go"); err == nil {
6      fmt.Printf("File exists\n");
7    } else {
8      fmt.Printf("File does not exist\n");
9    }
10 }
11
```

If no file root is specified, it will look for the file in the same directory as the code.
If the file exists, it will return true. If not, it will return false.

Error checking

Sometimes you want to check if a file exists before continuing the program. This leads to clean code: first check for errors, if no errors continue.

```
1  package main
2  import "fmt"
3  import "os"
4  func main() {
5    if _, err := os.Stat("file-exists2.file"); os.IsNotExist(err) {
6      fmt.Printf("File does not exist\n");
7    }
8    // continue program
9
10 }
11
```

Exercises

1. Check if a file exists on your local disk
2. Can you check if a file exists on an external disk?

Download Answers

Read file

golang can be used to read files. You can either read a file directly into a string variable or read a file line by line.

These functionality golang provides out of the box is for read files on the hard disk, not on the cloud.

Read files in golang

Read file

The golang program below reads a file from the disk. golang will read the file from the same directory as your program. If the file is in another directory, specify its path.

If you want to read a file at once, you can use:

```
1
2
3  package main
4  import (
5      "fmt"
6          "io/ioutil"
7          )
8  func main() {
9      b, err := ioutil.ReadFile("read.go")
10     // can file be opened?
11     if err != nil {
12       fmt.Print(err)
13     }
14     // convert bytes to string
15     str := string(b)
16     // show file data
17     fmt.Println(str)
18 }
19
20
```

This reads the entire file into a golang string.

Line by line

If you want to read a file *line by line*, into an array, you can use this code:

```
1  package main
2  import (
3    "bufio"
4    "fmt"
5    "log"
6    "os"
7  )
```

```
8
9
10 // read line by line into memory
11 // all file contents is stores in lines[]
12 func readLines(path string) ([]string, error) {
13   file, err := os.Open(path)
14   if err != nil {
15     return nil, err
16   }
17   defer file.Close()
18   var lines []string
19   scanner := bufio.NewScanner(file)
20   for scanner.Scan() {
21     lines = append(lines, scanner.Text())
22   }
23   return lines, scanner.Err()
24 }
25 func main() {
26   // open file for reading
27   // read line by line
28   lines, err := readLines("read2.go")
29   if err != nil {
30     log.Fatalf("readLines: %s", err)
31   }
32   // print file contents
33   for i, line := range lines {
34     fmt.Println(i, line)
35   }
36 }
37
38
```

Exercises

1. Think of when you'd read a file 'line by line' vs 'at once'?
2. Create a new file containing names and read it into an array

Download Answers

Write file

golang can open a file for reading and writing (r+) or writing (w+). This is for files on the hard disk, not on the cloud.

In this article we will cover how to write files in golang. If you are interested in reading files, see tthe read-file article instead.

Write files in golang

Write file

The golang program below writes a file from the disk. If the file exists, it will be overwritten (w+). If you want to add to end of the file instead, use (a+).

```
1  // Write file in go. You don't need to set any flags.
2  // This will overwrite the file if it already exists
3  package main
4  import "os"
5  func main() {
6      file, err := os.Create("file.txt")
7      if err != nil {
8          return
9      }
10     defer file.Close()
11     file.WriteString("write file in golang")
12 }
```

This writes the golang string into the newly created file. If you get an error, it means you don’t have the right user permissions to write a file (no write access) or that the disk is full.

Otherwise the new file has been created (file.txt). This file contains the string contents which you can see with any text editor.

Flags

If you use the w+ flag the file will be created if it doesn’t exist. The w+ flag makes golang overwrite the file if it already exists. The r+ does the same, but golang then allows you to read files. Reading files is not allowed with the w+ flag.

If you want to append to a file (add) you can use the a+ flag. This will not overwrite the file, only append the end of the file.

Exercises

1. Write a list of cities to a new file.

Rename file

Rename files with golang. Once you have a file in a directory you can simply rename it from your code.

The file will be renamed in the same directory. If you want to rename and move it to a new directory, change the variable to_file.

Rename file in golang

Example

The program below renames an existing file. Make sure the file exists before running the file. You can simply create an empty file.

```
1
2
3  package main
4  import "os"
5  func main() {
6    // source and destionation name
7    src := "hello.txt"
8    dst := "golang.txt"
9    // rename file
10   os.Rename(src, dst)
11 }
12
```

Run the program with the command:

```
1 go run rename.go
```

The file will be renamed to a new file.

Rename in shell

Now there are other ways to do this, for example on a Linux or Mac OS X system you can run the command

```
1 mv source.txt destination.txt
```

But this may or may not work on other platforms. Thats why you should always use the modules provides by the programming language.

Exercises

- Which package has the rename function?

Struct

A struct can bundle attributes together. If you create a struct, you can set a couple of variables for that struct. Those variables can be of any datatype.

A key difference from an array is that elements of an array are all of the same datatype. That is not the case with a struct.

If you want to combine variables, structs are the way to go. Unlike the concept of object oriented programming, they are just data holders.

Struct in golang

Struct example

The golang example creates a new struct. Then it sets the variables.

```
1  package main
2  import "fmt"
3  type Person struct {
4     name string
5     job string
6  }
7  func main() {
8     var aperson Person
9
10    aperson.name = "Albert"
11    aperson.job = "Professor"
12
13    fmt.Printf( "aperson.name =  %s\n", aperson.name)
14    fmt.Printf( "aperson.job  =  %s\n", aperson.job)
15 }
16
```

The program bundles the variables (job, name) into a struct named Person.Then that structure can be used to set variables.

In this example the struct has only two variables, but a struct can have as many as you need.

You can create multiple items with the same struct, all with different values. Elements of a struct can be accessed immediately.

Exercises

1. Create a struct house with variables noRooms, price and city
2. How does a struct differ from a class?

Maps

A Golang map is a unordered collection of key-value pairs. For every key, there is a unique value. If you have a key, you can lookup the value in a map.

Sometimes its called an associative array or hash table. An example of a map in Go:

```
1 elements := make(map[string]string)
```

Its defines as a string to string mapping. In this example we'll use the periodic elements.

You can map on other variable types too. Below an example of string to int mapping:

```
1 alpha := make(map[string]int)
2 alpha["A"] = 1
```

Map in golang

Map example

Create a new file named map.go, I like to use emacs, but any editor will do.

```
1 emacs -nw map.go
```

Copy the code below:

```
1
2
3  package main
4  import "fmt"
5  func main() {
6      elements := make(map[string]string)
7      elements["O"] = "Oxygen"
8      elements["Ca"] = "Calcium"
9      elements["C"] = "Carbon"
10     fmt.Println(elements["C"])
11
12
```

Run your go program:

```
1 go run map.go
```

Hashmap

The above creation of code works but it's a bit over expressive.
You can define a map as a block of data, in which there is the same key value mapping.

```
1 alpha := map[string]int{
```

```
2     "A" : 1,
3     "B" : 2,
4     "C" : 3,
5 }
```

This will do the exactly the same, but is a more elegant notation.

Store information

You can use a map to store information.
We change the map, into a map of strings to maps of strings to strings.

```
1 website := map[string]map[string]string {
```

Then you can store information like this:

```
1  website := map[string]map[string]string {
2      "Google": map[string]string {
3          "name":"Google",
4          "type":"Search",
5      },
6      "YouTube": map[string]string {
7          "name":"YouTube",
8          "type":"video",
9      },
10 }
```

Then get a value using two keys,

```
1 fmt.Println(website["Google"]["name"])
2 fmt.Println(website["Google"]["type"])
```

Exercises

- What is a map?
- Is a map ordered?
- What can you use a map for?

Random numbers

golang can generate random numbers. A random number is unknown before running: it's like telling the computer, give me any number.

Random numbers in computing are not truely random, they are often based on a pseudo random number generator algorithm. Eitherway for most program that degree of randomness is enough.
In this article you will learn how to generate random numbers.

Random number in golang

Example

The golang program below generates a number between 0 and 10. The starting number (0) is not given and thus 0 is assumed as lowest number.

```
1
2
3  package main
4  import (
5      "fmt"
6      "math/rand"
7      "time"
8  )
9  func random(min int, max int) int {
10     return rand.Intn(max-min) + min
11 }
12 func main() {
13     rand.Seed(time.Now().UnixNano())
14     randomNum := random(0, 10)
15     fmt.Printf("Random number: %d\n", randomNum)
16 }
17
```

To generate a number between 20 and 40 you can use the code below:

```
1 rand.Intn(max-min) + min
```

Exercises

1. Make a program that rolls a dice (1 to 6)
2. Can you generate negative numbers?

Pointers

Learn about pointers and memory addresses in Go. Some things are more easily done with pointers, others without. Eitherway pointers are fun to work with!

Not all programming languages support pointers, typically they are found in low level languages like C or C++. Go also supports them.

Examples

Memory addresses

Every variable is is stored in the memory and has a unique address. This can be accessed with the ampersand (&) operator.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6    var x int = 5
7    fmt.Printf("Address of variable x: %x\n", &x  )
8    fmt.Printf("Value of variable x: %d\n", x )
9 }
```

If you run the code with the command

```
1 go run program.go
```

You will see the program tells you the memory address and the contents in that memory address.

Something like:

```
1 Address of variable x: c4200160d8
2 Value of variable x: 5
```

The memory address will be different on your computer, value the same.

Pointers

A pointer is a variable whose address is the direct memory address of another variable.

The asterisk * is used to declare a pointer.
The form is:

```
1 var var_name *var-type
```

You have to write the variable type,

```
1
2 /* pointer to an integer */
3 var ipointer *int
4 /* pointer to a float */
5 var fpointer *float32
```

So lets see that in an example

```
1  package main
2
3  import "fmt"
4
5  func main() {
6     // variable
7     var x int = 5
8
9     // create pointer
10    var ipointer *int
11
12    // store the address of x in pointer variable
13    ipointer = &x
14
15    // display info
16    fmt.Printf("Memory address of variable x: %x\n", &x  )
17    fmt.Printf("Memory address stored in ipointer variable: %x\n", ipointer )
18    fmt.Printf("Contents of *ipointer variable: %d\n", *ipointer )
19 }
```

This will output something like:

```
1 Memory address of variable x: c4200160d8
2 Memory address stored in ipointer variable: c4200160d8
3 Contents of *ipointer variable: 5
```

Exercises

- Where are variables stored in the computer?
- What is a pointer?
- How can you declare a pointer?

Slices

golang slices are subsets. A slice can be a subset of an array, list or string. You can take multiple slices out of a string, each as a new variable.

A slice is never longer than then the original variable. This makes sense, if you take a slice of a pizza you don't suddenly have two pizzas. In programming it's similar.

Slices in golang

Example

The golang code will take a slice out of a list of numbers.

Start by creating a list of numbers (myset). Then take a slice by specifying the lower and upper bounds. In programming languages the lower bounds is zero (arrays start at 0).

```
1
2
3  package main
4  import "fmt"
5  func main() {
6      /* create a set/list of numbers */
7      myset := []int{0,1,2,3,4,5,6,7,8}
8      /* take slice */
9      s := myset[0:4]
10     fmt.Println(s)
11 }
12
```

The above program takes a slice and outputs it.

```
1 [0 1 2 3]
```

String slicing

golang strings can be sliced too. The resulting slice will then also be a string, but of smaller size.

When slicing, remember that the index 0 is the first character of the string. The last character is the numbers of characters minus 1.

```
1  package main
2  import "fmt"
3  func main() {
4      /* define a string */
5      mystring := "Go programming"
6      /* take slice */
```

```
7
8
9       s := mystring[0:2]
10      fmt.Println(s)
11  }
12
```

Exercises

1. Take the string ‘hello world’ and slice it in two.
2. Can you take a slice of a slice?

Functions

golang functions are reusable code blocks. By calling a golang method, all of the code in the method will be executed.

Functions should start with a lowercase character and only contain alphabetic characters.
A function can take one or more parameters which can be used in the code block.

Functions in golang

Example

The function below can be called as many times as you want: a function is reusable code.
Functions can also return output, this output can then be used in the program.

```
1
2
3  package main
4  import "fmt"
5  func main() {
6     hello("go")
7     hello("")
8  }
9  func hello(x1 string) {
10    fmt.Printf( "Hello %s", x1);
11 }
12
13
```

The method hello above is called with a parameter and without.
Sometimes parameters are necessary for your codeblock, but at times they are not.

The parameter in this example is x1, which is given a value outside the method.

Return valule

A value inside a golang function only exists there (local scope). It can be given to the program with the return statement, return x1. That then needs to be saved in an output variable.

```
1  package main
2  import "fmt"
3  func main() {
4     var a float64 = 3
5     var b float64 = 9
6     var ret = multiply(a, b)
7     fmt.Printf( "Value is : %.2f", ret )
8  }
9  func multiply(num1, num2 float64) float64 {
```

```
10
11
12      var result float64
13      result = num1 * num2
14      return result
15  }
16
```

Exercises

1. Create a function that sums two numbers
2. Create a function that calls another function.

Defer

Defer is a special statement in Go. The defer statement schedules a function to be called after the current function has completed.

Go will run all statements in the function, then do the function call specified by defer after.

Example

Defer

The normal execution in a Go function is top to bottom, if you had the function below, it would first call who() (top) and then hello (bottom).

```
1 func main() {
2     who()
3     hello()
4 }
```

If you add the **defer** statement, it will remember to call it after the function is finished.

```
1 func main() {
2     defer who()
3     hello()
4 }
```

It will first call hello() and then who().

Demo

The example below uses the defer statement to change the execution order.

```
1
2
3  package main
4  import "fmt"
5  func hello() {
6      fmt.Println("Hello")
7  }
8  func who() {
9      fmt.Println("Go")
10 }
11 func main() {
12     defer who()
13     hello()
14 }
15
16
17
```

Defer can also be called on simple function calls from packages, like this:

```
1 func main() {
2     defer fmt.Println("Hello")
3     fmt.Println("World")
4 }
```

Practical example

If you want to create and write a file, the steps would normally be:

1. create file
2. write file
3. close file

With the defer statement you could write:

1. create file
2. (defer) close file after function completes
3. write file

In code that looks like this:

```
1
2  package main
3  import "fmt"
4  import "os"
5  func main() {
6      f, _ := os.Create("hello.txt")
7      defer f.Close()
8      fmt.Fprintln(f,"hello world")
9  }
10
```

This has some advantages:

- readability
- if run-time panic occurs, deferred functions are still run
- if the function has return statements, close will happen before that

Exercise

1. Predict what this code does:

```
1 defer fmt.Println("Hello")
2 defer fmt.Println("!")
3 fmt.Println("World")
```

Multiple return

Go functions can return one or more values. Classical programming languages often only have zero or at most one return value.

Variables typically only exist in the function scope, but if they are returned they can be used in the program.

Example

Multiple return

A function that returns two values below:

```
1 func values() (int, int) {
2    return 2,4
3 }
```

The first line defines the parameters (*there are no paramaters*), then it shows the function the datatype to return. A function can have both multiple return variables and multiple parameters.

Beware of the round brackets: twice.

Then the function can be called and both values can be stored in new variables:

```
1 x, y := values()
```

The full code:

```
1
2
3  package main
4  import "fmt"
5  func values() (int, int) {
6     return 2,4
7  }
8  func main() {
9     x, y := values()
10    fmt.Println(x)
11    fmt.Println(y)
12 }
13
```

Exercise

1. Change the return values from 2,4 to “hello”,”world”. Does it still work?
2. Can a combination of strings and numbers be used?

Variadic functions

Variadic functions can be called with any number of arguments.

You've already used one variadic function: `fmt.Print(..)`. That function can be called with many arguments, like this: `fmt.Print("hello"," ","world","!","\n").`

A function is said to be variadic, if the number of arguments are not explictly defined.

Example

Variadic function

Define a variadic functions in this way:

```
func sum(numbers ...int) {
```

In this case it will take any amount of numbers (integers).

You can call a variadic function as you call normal functions. Any of the bottom calls will work:

```
sum(1,1)
sum(2,3,4)
sum(1,2,3,4,5,6,7)
```

Then in the sum function, add each number to the total amount with a for loop.

```
package main
import "fmt"
func sum(numbers ...int) {
    total := 0
    for _, num := range numbers {
        total += num
    }
    fmt.Println(total)
}
func main() {
    sum(2,3,4)
}

$ go run example.go
9
```

Exercises

- Create a variadic function that prints the names of students

Closure

Go Closures will be explained in this post. First we'll start of with an introduction: functions can contain functions. Then we'll give an example of a closure and finally a practical app.

Closure

Nested functions

Go functions can contain functions. Those functions can literally be defined in functions.

The example below outputs "hello world", but demonstrates this nested functions principle.

```
func main(){
    world := func() string {
        return "world"
    }
    fmt.Print("hello ", world(),"\n")
}
```

What is a closure?

In the next example, it creates a closure. The closure returns a number that increases each time its called.

You can use the variable x inside the increment function below. The function `increment` and variable `x` form the closure.

```
func main(){
    x := 2
    increment := func() int {
        x++
        return x
    }
    fmt.Println(increment())
    fmt.Println(increment())
    fmt.Println(increment())
}
```

x will keep being modified by the `increment` function. Each time the function `increment()` is called, `x` is modified.

> A closure is a type of function, that uses variables defined outside of the function itself.

Generator

You can use the idea of closures to make a number generator.

In the example below function `makeSequence()` returns an anonymous function that generates odd numbers. An anonymous function is just a function without a name.

```
func makeSequence() func() int {
    i:=1
    return func() int {
        i+=2
        return i
    }
}
```

`makeSequence()` returns a function, that function returns the numeric output.

Then we create a function: sequence generator:

```
sequenceGenerator := makeSequence()
```

And use it like this:

```
fmt.Println(sequenceGenerator())
fmt.Println(sequenceGenerator())
```

Full code:

```
package main
import "fmt"
func makeSequence() func() int {
   i:=1
   return func() int {
      i+=2
      return i
   }
}
func main(){
    sequenceGenerator := makeSequence()
    fmt.Println(sequenceGenerator())
    fmt.Println(sequenceGenerator())
    fmt.Println(sequenceGenerator())
}
```

Panic

Go has a `Panic(msg)` function. If the panic function is called, execution of the program is stopped. The panic function has a parameter: a message to show.

You can use the panic function if a failure occurs that you don't want or don't know how to deal with.

Example

Introduction

In the most basic scenario, you call the panic function. Call the panic function is a simple as this code:

```
package main
import "fmt"
func main(){
    panic("Something went wrong")
    fmt.Println("golang")
}
```

In a terminal, it shows:

```
$ go run demo.go
panic: Something went wrong

goroutine 1 [running]:
main.main()
    /home/linux/z/demo.go:6 +0x39
    exit status 2
```

Panic function

Real life scenario: Your program needs to create a file, but you don't want to deal with error processing. The `panic()` function will *make the program exit* if it cannot create the file.

```
package main
import "os"
func main(){
    _, err := os.Create("/root/example")
    if err != nil {
        panic("Cannot create file")
    }
}

$ go run demo.go
panic: Cannot create file

goroutine 1 [running]:
main.main()
    /home/linux/z/demo.go:8 +0x66    exit status 2
```

Recursion

Recursion functions are supported in Go. While recursive functions are not unique to go, they are useful for solving special kinds of problems.

Example

Introduction

In the previous articles we have discussed functions. What makes a function recursive?

A function is recursive if it:

1. Calls itself
2. Reaches the stop condition

The function below is *not* a recursive function:

```
func hello() {
    fmt.Println("hello world")
    hello()
}
```

Because it calls itself (1), but *it doesn't have a stop condition* (2).

But the function below is *a recursive function*. It matches both conditions;

```
func countdown(x int) int {
    if x == 0 {
        return 0
    }
    fmt.Println(x)
    return countdown(x - 1)
}
```

Factorial function

In the example we create a factorial function and make it calculate `3!`.

Here is the recursive function:

```
func factorial(x uint) uint {
    if x == 0 {
        return 1
    }
    return x * factorial(x-1)
}
```

If called, the function calls itself:

```
return x * factorial(x-1)
```

And it has a stop condition `x == 0`. After which it ends the function execution.

Full code below:

```
package main
import "fmt"
func factorial(x uint) uint {
    if x == 0 {
        return 1
    }
    return x * factorial(x-1)
}
func main(){
    x := factorial(3)
    fmt.Println(x)
}
```

Exercises

- When is a function recursive?
- Can a recursive function call non-recursive functions?

Errors

Go has a builtin type for errors. In Go, an error is the last return value (They have type error).

The line `errors.New` creates a new error with a message. If there is no error, you can return the `nil` value.

Example

Errors

The function `do()` is called, which returns an error. To use Go errors, you must include the package errors: `import "errors"`.

```
package main
import "errors"
import "fmt"
func do() (int, error) {
    return -1, errors.New("Something wrong")
}
func main() {
    fmt.Println( do() )
}
```

```
$ go run example.go
-1 Something wrong
```

You can combine both the return values:

```
r, e := do()
if r == -1 {
    fmt.Println(e)
} else {
    fmt.Print("Everything went fine\n")
}
```

Goroutines

A goroutine is a function that can run *concurrently*. You can see it as a lightweight thread.
The idea stems from *concurrency*: working on more than one task simultaniously.

To invoke a Go routine, write `go` before the function call.

If you have a function `f(string)`, call it as `go f(string)` to invoke it as goroutine. The function will then run asynchronously.

Example

Introduction

The code below invokes a goroutine, calls the function and waits for keyboard input. The goroutine is executed *concurrently*.

```
1 go f("go routine")
2 f("function")
3 fmt.Scanln()
```

Go doesn't wait for goroutines to finished, it will return to the next line and run a goroutine concurrently. Without `fmt.Scanln()` Go would finish the program.

Goroutine

The goroutine defined `f(msg string)` is a simple function that outputs a line of text. It is called both as a regular function and as a goroutine. Goroutines are light on memory, a program can easily have hundreds or thousands of goroutines. This example starts a goroutine:

```
1  package main
2  import "fmt"
3  func f(msg string) {
4     fmt.Println(msg)
5  }
6  func main() {
7     go f("go routine")
8     f("function")
9     fmt.Scanln()
10 }
11
```

Exercises

- What is a goroutine?
- How can you turn a function into a goroutine?
- What is concurrency?

Channels

Channels are a way for goroutines to communicate with eachother. A channel type is defined with the keyword `chan`.

Because goroutines run concurrently, they can't simply pass data from one goroutine to another. Channels are needed.

How do channels work?

If you type `c <- "message"`, "message" gets send to the channel. Then `msg := <- c` means receive the message and store it in variable msg.

Example

Goroutines

In this example there are two goroutines (f, f2). These goroutines communicate via a channel, `c chan string`.

In goroutine `f(c chan string)` we send a message into the channel. In goroutine `f2(c chan string)` the message is received, stored and printed to the screen.

```
package main
import "fmt"
func f(c chan string) {
    c <- "f() was here"
}
func f2(c chan string) {
    msg := <- c
    fmt.Println("f2",msg)
}
func main() {
   var c chan string = make(chan string)
   go f(c)
   go f2(c)
   fmt.Scanln()
}

$ go run example.go
f2 f() was here
```

Exercises

- When do you need channels?
- How can you send data into a channel?
- How can you read data from a channel?

Channels

Channels by default accept a single send (<-) if there is a receive. They are *unbuffered*. A buffer is like a memory, multiple items can be put in a buffer.

Channels usually are *synchronous*. Both sides (goroutines) need to be ready for sending or receiving. Not so with buffered channels, they are *asynchronous*.

Go supports channels that have a buffer, *buffered channels*.

Example

Buffered channel

A buffered channel (named c) can be created with the line:

`var c chan string = make(chan string,3)`.

The second parameter is the capacity. This will create a buffer with a capacity of 3.

Then multiple messages can be stored using `c <- message`. If you want to output a channel element, you can use the line `fmt.Println(<-c)`.

```
package main
import "fmt"
func main() {
    var c chan string = make(chan string,3)
    c <- "hello"
    c <- "world"
    c <- "go"
    fmt.Println(<-c)
    fmt.Println(<-c)
    fmt.Println(<-c)
}

$ go run example.go
hello
world
go
```

Because this channel is buffered, you can store values without a having another goroutine to save them immediately.

Channel synchronization

Channels can be used to synchronize goroutines. A channel can make a goroutine wait until its finished. The channel can then be used to notify a 2nd goroutine.

Imagine you have several goroutines. Sometimes a goroutine needs to be finished before you can start the next one (synchronous). This can be solved with *channels*.

Channel synchronization

Example

Define a function to be used in the program. It can be a simple function like the one below:

```
func task(done chan bool) {
    fmt.Print("running...")
    time.Sleep(time.Second)
    fmt.Println("done")
    done <- true
}
```

This will output “running…”, wait, then print done and send “true” in the channel named ‘done’. The channel named ‘done’ is of type boolean (true or false).

The code would be this:

```
package main
import "fmt"
import "time"
func task(done chan bool) {
    fmt.Print("running...")
    time.Sleep(time.Second)
    fmt.Println("done")
    done <- true
}
func main() {
   done := make(chan bool,1)
   go task(done)
   <- done
}

go run example.go
running...done
```

Synchronizing goroutines

You can then make one goroutine wait for another goroutine. You can do that with an if statement and reading the channel value.

```
1
2
3
4
5  package main
6  import "fmt"
7  import "time"
8  func task(done chan bool) {
9      fmt.Print("Task 1 (goroutine) running...")
10     time.Sleep(time.Second)
11     fmt.Println("done")
12     done <- true
13 }
14 func task2() {
15     fmt.Println("Task 2 (goroutine)")
16 }
17 func main() {
18    done := make(chan bool,1)
19    go task(done)
20    if <- done {
21        go task2()
22        fmt.Scanln()
23    }
24 }
25
26
27
```

Now goroutine task2 will wait for goroutine task to be finished.

Note

Thus these goroutines (task, task2) can be synchronous all the while running concurrently.

Because everything is concurrent, you can still use the `main thread` for your program, at the same time.

```
1  func main() {
2     done := make(chan bool,1)
3     go task(done)
4
5     fmt.Println("Im in the main thread!")
6
7     if <- done {
8         go task2()
9         fmt.Scanln()
10     }
11 }
```

Channel directions

Channels (as function parameters) can have a *direction*. By default a channel can both send and receive data, like `func f(c chan string)`.

But you can define a channel to be *receive-only* or *send-only*. If you then use it in the other direction, it would show a *compile-time error*. This improves the type-safety of the program.

Example

Receive only

You can define a channel as `func f(c <- chan string)` to have a receive-only chan. If you want a send only chan, create it as `func f(c chan <- string)`.

The program below creates a receive-only channel for the function f, `func f(c <- chan string)`. Then the line `fmt.Println(<-c)` gets data from the channel.

The line `c <- "hello"` sets data on the channel.

```
package main
import "fmt"
func f(c <- chan string) {
   fmt.Println(<-c)
}
func main() {
   c := make(chan string, 1)
   c <- "hello"
   f(c)
}

$ go run example.go
hello
```

If you then try to set data inside the function, it will tell it's receive-only:

```
$ go run example.go
./example.go:7:6: invalid operation: c <- "f was here" (send to receive-only type
<-chan string)
```

Send only channel

A send only channel can set its values in a function, but cannot receive.

Change function to `func f(c chan <- string)`.
Hint: Here the only change is the position of `<-`.

```
package main
import "fmt"
func f(c chan <- string) {
   c <- "send only channel"
}
func main() {
   c := make(chan string, 1)
   f(c)
   fmt.Println(<-c)
}
```

Select

Select waits on multiple channels. This can be combined with goroutines. `select` is like the switch statement, but for channels.

If multiple channels need to be finished before moving on to the next step, `select` is what you need.

Example

Goroutines

Two goroutines are started concurrently. Each routine loops forever and writes data into a channel.

To be explicit: goroutine `f1` writes data into channel `c1` forever, goroutine `f2` writes dat ainto channel `c2` forever. The goroutines are simply like this:

```
func f1(c chan string) {
    for {
      time.Sleep(1 * time.Second)
       c <- "1"
    }
}
```

Select

Because it runs concurrently, it may happen that one task finishes before the other.

The `select` statement will make the program wait for both tasks to be completed, but that doesn't mean both tasks are always finished in chronological order.

```
package main
import "fmt"
import "time"
func f1(c chan string) {
    for {
      time.Sleep(1 * time.Second)
       c <- "1"
    }
}
func f2(c chan string) {
    for {
      time.Sleep(1 * time.Second)
      c <- "2"
    }
}
func main() {
   c1 := make(chan string)
   c2 := make(chan string)
   go f1(c1)
```

```
    go f2(c2)
    for {
        select {
            case msg1 := <- c1:
                fmt.Println(msg1)
            case msg2 := <- c2:
                fmt.Println(msg2)
        }
    }
    fmt.Println("Goroutines finished.")
}

$ go run example.go
1
2
1
2
2
1
```

Timeout

You can create a `timeout` with the `select` statement. To use timeouts with concurrent goroutines, you must `import "time"`. Then, create a channel with `time.After()` which as parameter takes time. The call `time.After(time.Second)` would fill the channel after a second.

Example

Timeout

Combined with `select`, a simple timeout program can be created:

```
package main
import "fmt"
import "time"
func main() {
    select {
        case <- time.After(2 * time.Second):
            fmt.Println("Timeout!")
    }
}
```

This will print “Timeout!” after the time has passed.

Goroutine

Timeouts can be combined with goroutine calls. Call a goroutine `f1` with a channel `c1`. Making `go f1(c1)`. The goroutine writes in to the channel `c <- "message"` after waiting 10 seconds. Then a timeout is made with `time.After()`. As this:

```
package main
import "fmt"
import "time"
func f1(c chan string) {
    for {
        time.Sleep(10 * time.Second)
        c <- "10 seconds passed"
    }
}
func main() {
   c1 := make(chan string)
   go f1(c1)
   select {
       case msg1 := <- c1:
           fmt.Println(msg1)
       case <- time.After(3 * time.Second):
           fmt.Println("Timeout!")
   }
```

```
}
```

Close channel

Channels can be closed. *If a channel is closed, no values can be sent on it.* A channel is open the moment you create it, e.g. `c := make(chan int, 5)`.

You may want to close a channel, to indicate completion. That is to say, when the goroutines are completed it may be the channel has no reason to be open.

Example

Closing channel

The line `c := make(chan int, 5)` creates a buffered channel of capacity 5. A buffered channel can store multiple values without it being received.

Then data is sent to the channels with the lines `c <- 5` and `c <- 3`.

The channel is closed with the function `close(channel)`.

```
package main
func main() {
    c := make(chan int, 5)
    c <- 5
    c <- 3
    close(c)
}
```

If you sent data after the channel is closed, like `c <- 1` after the `close(c)` call, it will throw this error:

```
$ go run example.go
panic: send on closed channel

goroutine 1 [running]:
main.main()
    /home/linux/golang/example.go:11 +0x9b
    exit status 2
```

Receive data

If a channel is closed, you can still read data. But you cannot send new data into it.
This program reads both before and after closing the channel, which works. It's closed *only for sneding*, meaning a line like `c <- 9` won't work after `close(c)`.

```
package main
import "fmt"
func main() {
    c := make(chan int, 5)
    c <- 5
    c <- 3
    fmt.Println(<-c)
    close(c)
    fmt.Println(<-c)
}
```

Range channel

In the previous article you saw how to use `range` on data structures including slices and arrays. The `range` keyword can also be used on a channel.

By doing so, it will iterate over every item thats send on the channel. You can iterate on both buffered and unbuffered channels, but buffered channels need to be closed before iterating over them.

Range

Iterate over buffered channel

The range keyword can be used on a *buffered channel*. Suppose you make a channel of size 5 with `make(chan int, 5)`. Then store a few numbers into it `channel <- 5`, `channel <- 3` etc.

You can then iterate over every item that was sent into the channel with the `range` keyword.

```
1
2
3  package main
4  import "fmt"
5  func main() {
6      channel := make(chan int, 5)
7      channel <- 5
8      channel <- 3
9      channel <- 9
10     close(channel)
11     for element := range channel {
12         fmt.Println(element)
13     }
14 }
15


$ go run example.go
5
3
9
```

The channel is closed with `close(channel)` before iteration it. That's why it will terminate after 3 items.

Iterate over channel

You can use the `range` statement even with unbuffered channels. Say you create a goroutine `f(c chan int)`, which pushes the current second into the channel each second.

First create a channel, say of integers: `channel := make(chan int)`.

Then create the goroutine:

```
func f(c chan int) {
    for {
        c <- time.Now().Second()
        time.Sleep(time.Second)
    }
}
```

Then start the goroutine with `go f(channel)`. Every second there is a number pushed into the channel.

You can iterate over this channels with the `range` keyword:

```
for element := range channel {
    fmt.Println(element)
}
```

Timers

A `timer` is a single event in the future. A timer can make the process wait a specified time. When creating the timer, you set the time to wait.

To make a timer expire after 3 seconds, you can use `time.NewTimer(3 * time.Second)`. If you only want to wait, use `time.Sleep(3)` instead.

If you want to repeat an event, use `tickers`.

Timers

Example

This program waits for 3 seconds before continuing. The line `<- t1.C` blocks the timers channel `C`. It unblocks when the timer has expired.

```
package main
import "fmt"
import "time"
func main() {
    t1 := time.NewTimer(3 * time.Second)
    <- t1.C
    fmt.Println("Timer expired")
}
```

Stop timer

Unlike `time.Sleep()`, a `timer` can be stopped. In some scenarios you want to be able to cancel a process, like if you download a file or if you are trying to connect.

The program below allows a user to cancel the timer.

```
package main
import "fmt"
import "time"
func main() {
     t1 := time.NewTimer(time.Second)
     go func() {
         <-t1.C
         fmt.Println("Timer expired")
     }()

     fmt.Scanln()
     stop := t1.Stop()
     if stop {
         fmt.Println("Timer stopped")
     }
```

```
}
```

Tickers

Tickers can repeat execution of a task every n seconds. This is unlike timers which are used for timeouts. A ticker can repeat a block of code.

Goroutines run concurrently and can have tickers inside them.

Example

time.Tick

You can use the function `time.Tick(n)`. An example call is `time.Tick(time.Second * 1)`. Combined with `range` that repeats every second.

You can combine it with `time.Sleep(n)` to make it won't simply quit while running the goroutine.

```
package main
import "fmt"
import "time"
func task() {
    for range time.Tick(time.Second *1){
            fmt.Println("Tick ")
    }
}
func main() {
    go task()
    time.Sleep(time.Second * 5)
}
```

Tickers

You can create a ticker anywhere in the code with the line `time.NewTicker(n)`.

Then you can use it to tick every interval:

```
package main
import "fmt"
import "time"
func task() {
    ticker := time.NewTicker(time.Second * 1)
    for range ticker.C {
        fmt.Println("Tick ")
    }
}
func main() {
    go task()
    time.Sleep(time.Second * 5)
}
```

Date and time

golang can display date and time. In this article you will learn how to deal with date and time in golang.

Date is default, but time cannot go back earlier than 1970. Why? that's when the logic was added to computers.

Date and time in golang

Example

The program below is an example of date and time in golang. The formatting is explicitly defined (%y for year, %m for month etc).

```
1
2  // Golang doesnt have strftime (%Y,%M,%d) etc.
3  // Instead use the time package.
4  package main
5  import (
6      "fmt"
7      "time"
8  )
9  func main() {
10   current := time.Now().UTC()
11   fmt.Println("Date: " + current.Format("2006 Jan 02"))
12   fmt.Println("Time: " + current.Format("03:04:05"))
13 }
14
```

Strftime

But if you use the arrow library, you can use strftime style formatting.
Install the package with:

```
1 go get github.com/bmuller/arrow/lib
```

Then run the code below:

```
1  package main
2  import (
3          "fmt"
4          "github.com/bmuller/arrow/lib"
5  )
6  func main() {
7       // formatting
8       fmt.Println("Date: ", arrow.Now().CFormat("%Y-%m-%d"))
9       fmt.Println("Time: ", arrow.Now().CFormat("%H:%M:%S"))
10 }
11
```

```
12
```

The code above displays the date and time. You can use an alternative formatting if you want..
The output will be similar to this:

```
1 Date : 2018-06-07
2 Time : 10:38:01
```

Exercises

1. Display date in DD/MM/YYYY format