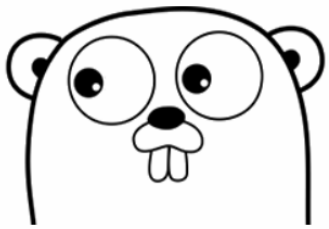
SECTION 1 – GETTING STARTED

Introduction to Golang

- Golang: Open source; Simple, Reliable, and Efficient



- Designers: **Robert Griesemer**, **Rob Pike**, and **Ken Thompson**.

  Ref: https://en.wikipedia.org/wiki/Go_(programming_language)

Robert Pike is a Canadian programmer and author. He is best known for his work on Go (programming language) and at Bell Labs, where he was a member of the Unix team and was involved in the creation of the Plan 9 from Bell Labs and Inferno operating systems, as well as the Limbo programming language.
Ref: https://en.wikipedia.org/wiki/Rob_Pike

Ken Thompson, an American pioneer of computer science is a member of the Unix team at Bell Labs, one of the fathers of C, Unix and Plan 9 operating systems, co-developed UTF-8 with Rob Pike. Having worked at Bell Labs for most of his career, Thompson designed and implemented the original Unix operating system. He also invented the B programming language, the direct predecessor to the C programming language, and was one of the creators and early developers of the Plan 9 operating systems. Since 2006, Thompson has worked at Google, where he co-invented the Go programming language.
Ref: https://en.wikipedia.org/wiki/Ken_Thompson

Robert Griesemer is known for his work at the Java HotSpot Virtual Machine. One of the first designers of the Go (golang) programming language. Previously worked on code generation for Google's V8 JavaScript engine and Chubby. Worked on the Sawzall language, the Java HotSpot virtual machine, and the Strongtalk system.
Ref: https://www.computerhope.com/people/robert_griesemer.htm

- **Meet the Go Team (2012):**
  https://www.youtube.com/watch?v=sln-gJaURzk&t=321s

- What's happening with **other languages**?
- **Why Go?**
- **Why is Go successful?**
- **Go's goals**.

- Go is a **statically typed** compiled language in the tradition of C, with **memory safety**, **garbage collection**, **structural typing**, and **concurrent programming** features added.
- Go is a **general purpose** programming language. Go can be used for development of different kinds of software (systems, cloud, game, network/server programming, …)
- Go is called a '**C for the 21st century**'. It belongs to the C-family and is inspired by languages such as C++, Java, C#, Pascal, and Modula.
- Go-syntax is close to the **C-syntax**, however it's been **greatly simplified**, it's cleaner and more concise.

- Many well-known **companies use Go**: Google, Adobe, Dropbox, Facebook (github, blog), Soundcloud, Docker, Cloudflare, BBC, IBM, Shopify tweet, ...
  Ref: https://github.com/golang/go/wiki/GoUsers

- Go is **fast**: to learn, develop, design, compile, deploy, test, run, …
  Go has the speed of a compiled language, but the feel of an interpreted language.
  Compiled programming language: source code is translated into binary code; therefore a Go compiler is required to complete your code.

- Go is a **modern language** addressing needs of cloud computing, modern hardware architectures, …

- Golang has a **clean syntax**, following a **minimalistic design**; focused on best practices in software engineering.

- Golang offers **built-in concurrency** (using channels and goroutines). This feature is not available or hard to use in many back-end programming languages.

- Rapid development, **growing community**, good documentation.

- Go comes with **fast garbage collection** and **memory-management**. In languages such as C++ developers are responsible for managing memory.

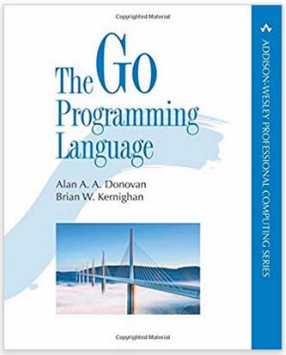
- Current release: **go1.10** (released 2018/02/16)
  Ref: https://golang.org/doc/devel/release.html

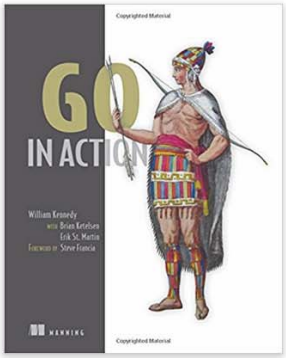
Some useful links

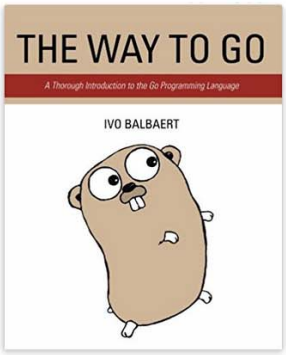
- Go Official Site: https://golang.org/
- Go Playground: https://play.golang.org/
- Go Github: https://github.com/golang
- Go Language Specification: https://golang.org/ref/spec
- Go Installation: https://golang.org/doc/install
- Frequently Asked Questions (FAQ): https://golang.org/doc/faq
- The Go Blog: https://blog.golang.org
- Effective Go: https://golang.org/doc/effective_go.html
- Go Forum: https://forum.golangbridge.org/
- Google discussion-group (Go Nuts): https://groups.google.com/forum/#!forum/golang-nuts
- Wikipedia Go (programming language): https://en.wikipedia.org/wiki/Go_(programming_language)

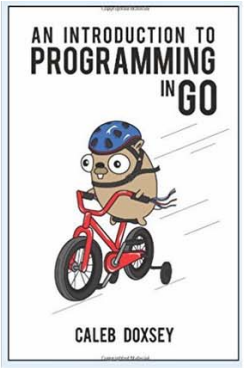
Go Programming Books

- **The Go Programming Language** (Addison-Wesley), by Alan A. A. Donovan and Brian W. Kernighan
- **Go in Action**, by William Kennedy and Brian Ketelsen
- **The Way To Go**: A Thorough Introduction To The Go Programming Language, by Ivo Balbaert
- **An Introduction to Programming in Go**, by Caleb Doxsey
- And many others …







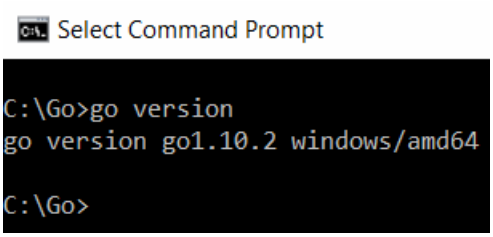


Editors and Integrated Development Environments

- **Visual Studio Code (VSCode)** used in this course:
  https://code.visualstudio.com/download
- SublimeText: https://www.sublimetext.com/
- The GEdit Linux text-editor: http://gohelp.wordpress.com/
- Golang LiteIDE: https://github.com/visualfc/liteide

Installation

- Install the **Go Compiler**
- Installation instructions: https://golang.org/doc/install
- Download a binary release suitable for your system: https://golang.org/dl/
- Follow the instructions to install Go in a folder of your choice (such as C:\Go\)
- Open a command prompt to validate installation:
  > **go version**

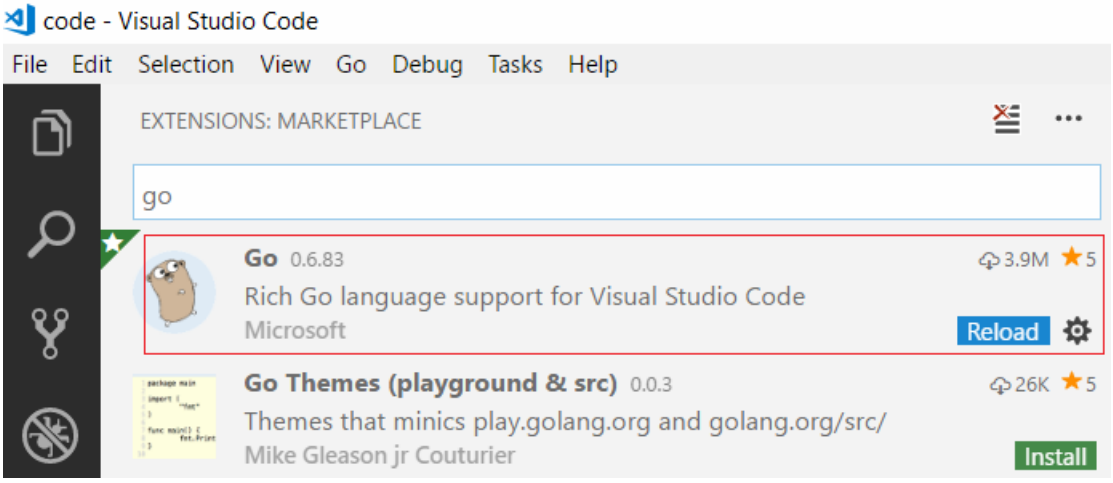


- Set **environment variables** as per Go installation page.
  Here's an example:
  GO_HOME=C:\Go
  GOPATH=C:\Users\tyler\go

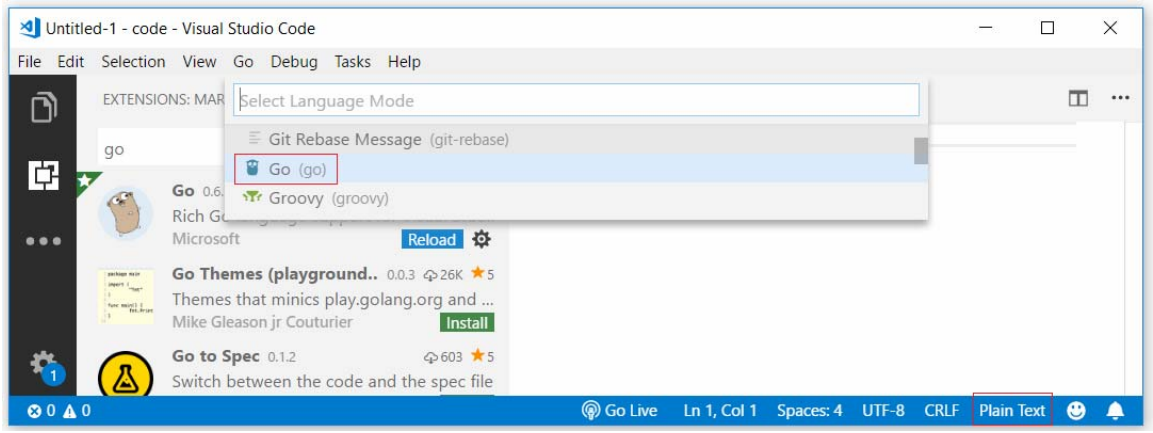
  Add the following to the 'path' environment variable:
  %GO_HOME%\bin\; %GOPATH%\src\; %GOPATH%\src\goworkspace\;

- Download **Visual Studio Code (VSCode)** as your editor:
  https://code.visualstudio.com/download

- **VSCode Configuration**
  - Start VSCode
  - Select View / Extensions; search for ‘go’; click ‘install’.



  - Close the VSCode and start it again (for the new installation to take effect).
  - Select File / New File
    On the bottom right hand side of the page select “plain text” and change it to “Go”



  - Then on the bottom right hand side of the page, you see the message “**Analysis Tool is Missing**’. Click on it and let the VSCode install the required dependencies. This may take several minutes to install.

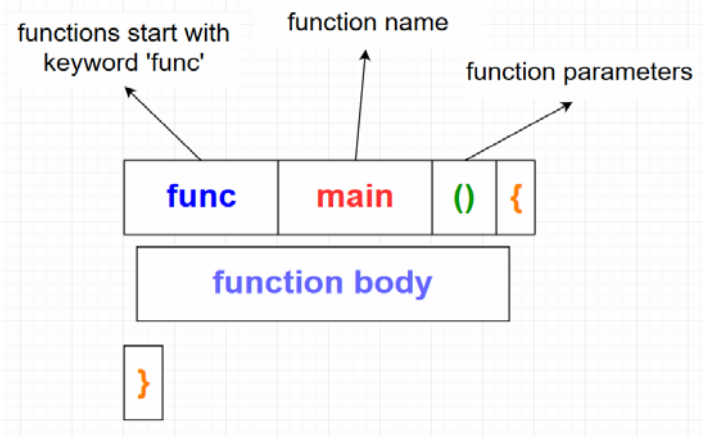
- **Git for Windows:** https://gitforwindows.org/

SECTION 2 – LANGUAGE FUNDAMENTALS

Your First Program

*****code: ...\s02\01_helloworld\main.go***

```
// File name: ...\helloworld\main.go
// I am a line comment!
/*
a multi-line
comment!
*/

package main
import "fmt"

func main() {
    fmt.Println("Hello World")
}
```



How to run the program?

> cd *yourDirectory*
> **go build** main.go
> **main**

or instead, use the following:
> **go run** main.go

Types of Go programs:

- **Executables**: can run directly; e.g., main();
- **Libraries:** packages of code that can be used in other programs

Some Notes:

- On **Windows** c:\users\tyler\main.go
- On **OSX:** /users/tyler/main.go *(forward slash)*

- For Windows users**:** you may need to install Linux bash on your computer to be able to run some of the code during the training.
  '**gitbash**' is a choice that can be found at: https://gitforwindows.org/

- On Windows, use "**windows key + r**' and then type '**cmd**' to open a command prompt.

Variable Names

- Begins with a **letter** or an **underscore**, plus any number of additional letters, digits, and underscores.
  Ref: https://golang.org/ref/spec#Identifiers

- **Case-sensitive**

- Go has **25 keywords**; can't be used as variable names.

Keywords

The following keywords are reserved and may not be used as identifiers.

```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

Ref: https://golang.org/ref/spec#Keywords

- Go programmers use "**camel case**" (e.g., firstName,)
- Go programs lean toward **short names** (e.g., i, s, …)

Go's Types

1. **basic** types: numbers, strings, and Booleans;
2. **aggregate** types: arrays and structs;
3. **reference** types: pointers, slices, maps, functions, and channels;
4. **interface** types.

Basic Data Types

- *Ref: https://golang.org/pkg/builtin/ (search for 'type bool')*

- **Basic Data Types in Summary**
  bool
  string
  int int8 int16 int32 int64
  uint uint8 uint16 uint32 uint64
  byte // alias for uint8
  rune // alias for int32; represents a Unicode code point
  float32 float64
  complex64 complex128

- Go is **statically typed**, means that when type of a variable is set, it can't be changed later.

- **Integer Numbers**
  - **Signed**
    - **int8** (8 bits=1 byte) (Range: -128 through 127)
    - **int16** (Range: -32768 through 32767)
    - **int32** (Range: -2147483648 through 2147483647)
    - **rune** (an alias for **int32**; the two names may be used interchangeably; used for **Unicode code point** values;)
    - **int** (at least 32 bits) (not an alias for any specific type, say, int32) (this is the default for integer values; **by far the most widely used numeric type**)
    - **int64** (Range: -9223372036854775808 through 9223372036854775807)

  - **Unsigned**
    - **uint8** (Range: 0 through 255)
    - **byte** (an alias for **uint8**; equivalent to uint8; used for **raw data** rather than a numeric quantities)
    - **uint16** (Range: 0 through 65535)
    - **uint32** (Range: 0 through 4294967295)
    - **uint** (at least 32 bits) (not an alias for any specific type, say, uint32)
    - **uint64** (Range: 0 through 18446744073709551615)
    - **uintptr** (it's is an unsigned integer type; it is large enough to hold the bits of any pointer value; used only for low-level programming, such as interaction of Go with a C library or an OS) (see the documentation)

- **Floating Point Numbers (real numbers)**
  - **float32** (32 bits) (the set of all IEEE-754 32-bit floating-point numbers) (e.g., 1.23, 0.0987)
  - **float64** (this is the mostly used type for floating point numbers)
  - Note: Not a Number(**NaN**) (for 0/0, +∞ and −∞)

- **Complex numbers:** complex64, complex128 (see the documentation)

- **Boolean:** bool (true and false); && (and), || (or), ! (not)

- **Strings:** string
  - String literals: "Hello World" or `Hello World` (back ticks);
  - set of individual bytes (**8-bits**), normally one byte per character;
  - Strings are 'indexed' starting at 0;
  - with a definite length;
  - conventionally but not necessarily representing UTF-8-encoded text;
  - values of string type are immutable;
  - a string may be empty, but not nil;

Variables (Declaration & Initialization)

- Form 1 (**explicit**):
```
var name type = expression
var i int = 12
```

- Form 2 (**default initialization**):
```
var name type
var i int
```

- Form 3 (**short variable**): *(recommended)*
```
name := expression
i := 12
```

Note 1: **:=** is a declaration **=** is an assignment

Note 2: x, y, z := 10, 20, 30 (***Tuple assignment*** / **multi-variable declaration**)
Usage of Tuple assignment to make a series of trivial assignments

x, y := y, x *(swaps the content of x and y)*
The right-hand side expressions are evaluated first

Use with caution; only when increases readability;

Note 3: nil may be assigned to any variable.

Zero value of the type

To make sure that variables hold well-defined values.

Zero Value	Type
0	numbers
false	booleans
""	string
nil	reference types (slice, pointer, map, channel, function)

Note: Zero value of an aggregate type (array, struct): the zero value of all of its elements/fields.

Lexical Elements

Ref: https://golang.org/ref/spec#Lexical_elements
The following is a summary of what is on this web page.

- **Comments**: // or /* */
- **Identifiers**: such as a, _x9, ThisVariableIsExported, …
- **Keywords**: break, default, func , if, ...
- **Integer literals**: 42, 0600, 0xBadFace, 170141183460469231731687303715884105727
- **Operators and punctuation**: + - & += ... := . : ; and others
- **Floating-point literals**: 0., 2.71828, 6.67428e-11, .25, .12345E+5
- **String literals**: "abc", `abc`, "\"", "日本語"
- **Constants**: const i=12
- **Variables**: var x int
- **Boolean types**: true, false
- **Numeric types**: int, int8, ..., uint, uint8, ..., float32, float64, complex64 , complex128 , byte, rune
- **Array types**: [10] int, [32]byte, ...
- **Slice types**: make([]int, 50, 100)
- **Struct types**: struct { name string, int age }
- **Rune literals**: A rune literal represents a rune constant, an integer value identifying a Unicode code point. A rune literal is expressed as one or more characters enclosed in single quotes, as in 'x'.

- **Tokens**:
  - line breaks or ";" as terminator;
  - an identifier;
  - an integer, floating-point, imaginary, rune, or string literal;
  - one of the keywords break, continue, fallthrough, or return;
  - one of the operators and punctuation ++, --,),], or }
- and many more …

Go's operators in order of decreasing precedence.

```
* / % << >> & &^                (left to right)
+ - | ^
== != < <= > >=
&&
||
```

comparison operators: == != < <= > >=

bitwise binary operators
& bitwise AND
| bitwise OR
^ bitwise XOR
&^ bit clear (AND NOT)
<< lef t shift
>> right shift

a	b	a&&b	a||b	^a
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Literals vs. Variables vs. Constants

- **variable**: a storage location; with specific type name;
  var i = 20; i++ // its value can change

- **const** pi = 3.14159
  checked at compile time; variables whose values cannot be changed later

- fmt.Println(25, pi, i) // printing a literal, a constant, and a variable

Some of the single-character escapes:

Ref: https://golang.org/ref/spec#Rune_literals

- \n line feed or newline
- \t horizontal tab
- \\ backslash
- \' single quote (valid escape only within rune literals)
- \"double quote (valid escape only within string literals)
- \b backspace
- \f form feed
- \r carriage return
- \v vertical tab
- \a alert or bell

Printing Verbs

Ref: https://golang.org/pkg/fmt/#Printing

Some of the more common verbs:

%v the value in a default format-when printing structs,
the plus flag (%+v) adds field names
%T a Go-syntax representation of the type of the value
%d base 10
%x base 16, with lower-case letters for a-f
%f decimal point but no exponent, e.g. 123.456
%g %e for large exponents, %f otherwise. Precision is discussed below.
%s the uninterpreted bytes of the string or slice

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**

Examples:

```
var i uint8 = 20                                    // type: unsigned int
var f float32 = 214.437                             // type: float32
var b bool = true                                   // type: boolean
var msg string = "hello"                            // type: string

type myStringT string                               // defining a new type
var myMsg myStringT = "special message"             // type: myStringT

fmt.Printf("%d %v %T \n", i, i, i)                  // 20 20 uint8
fmt.Printf("%5.3f %.2f %g %T\n", f, f, f, f)        // 214.437 214.44 214.437 float32
fmt.Printf("%t %v %T\n", b, b, b)                   // true true bool
fmt.Printf("%s %T\n", msg, msg)                     // hello string
fmt.Printf("%v %T\n", myMsg, myMsg)                 // special message main.myStringT
```

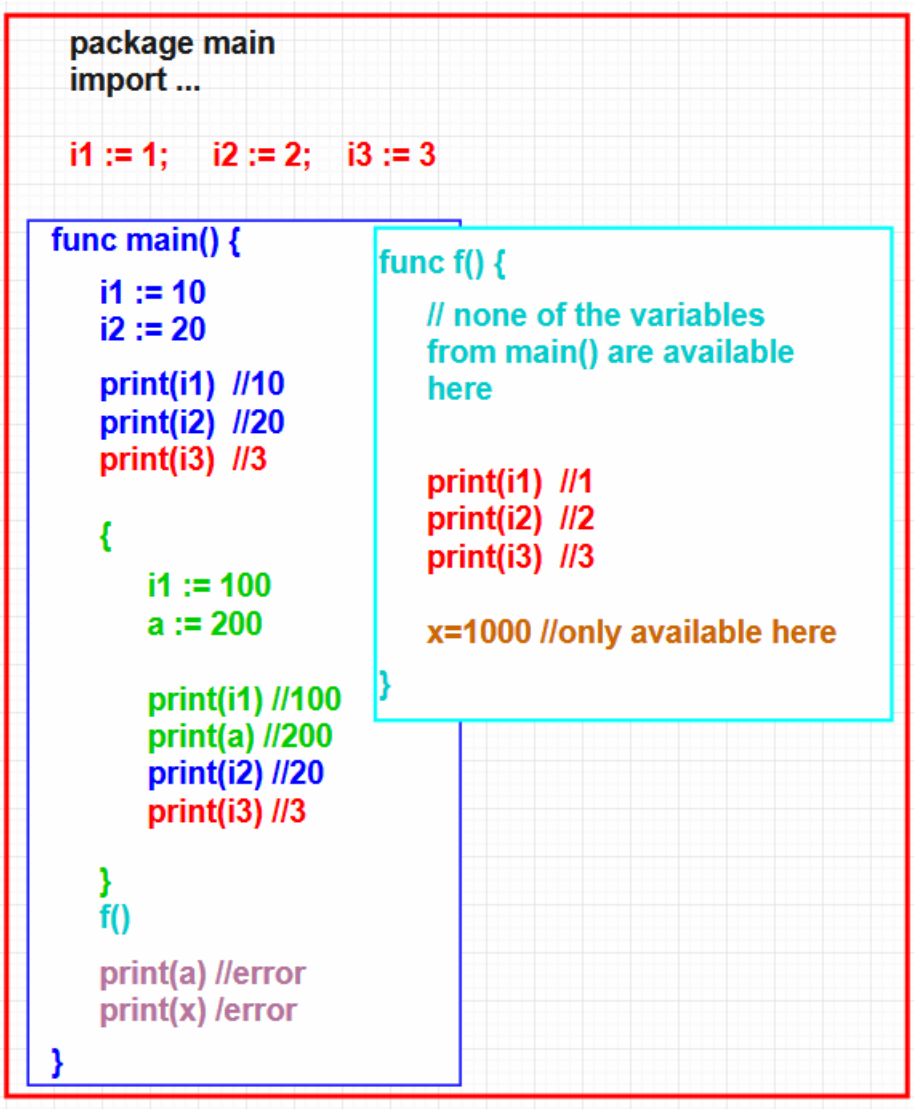
code: ...\s02\ review examples 1 to 24

Scope

- **Local-Level**:
  Scope of a **local** variable: only to its own block;
  lifetime: end of the enclosing block;

- **Global-Level**:
  lifetime: the entire execution of the program;
  - **File-Level**: If it starts with a **lowercase**: anywhere within the **file**;
  - **Package-Level**: If it starts with an **uppercase**: anywhere within the **package**;

- '**Scope**' of a variable is a compile-time property.
- '**Lifetime**' of a variable is a run-time property.



- **{ }** in the preceding code is called a **"syntactic block"**.

- **Lexical blocks**: The following 'for loop' contains two lexical blocks:

  1) an **explicit block** for the loop body;
  2) an **implicit block** for variable 'i' declared in the initialization clause;

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

code: ...\s02\25_scope\main.go

SECTION 3 – CONTROL STRUCTURES

if / else statement

Ref: https://golang.org/ref/spec#If_statements

```
if condition {
	// do something
}
```

```
if condition {
	// do something
} else {
	// do something else
}
```

```
if condition1 {
	// do something
} else if condition2 {
	// do something else
} else {
	// catch-all or default
}
```

code: ...\s03\ review examples 1 to 3

for statement

Ref: https://golang.org/ref/spec#For_statements

- A "for" statement specifies repeated execution of a block.
- There's no 'while', 'do while' in Go.
- **Three forms of a for loop:**
  - Single condition

```
for a < b {
        a *= 2
}
```

  - With **init** and **post** statement

    1) Initialize variable 'i' with zero;
    2) check the condition; execute the body lines; increase 'i' by 1;
    3) repeat step 2 until i<5;

```
for i := 0; i < 5; i++ {
        fmt.Println(i)
}
```

  - With a **range** clause

```
x := […]int {10, 20, 30}          // an array of integers
for i, val := range x {
        fmt.Println(i, val)       // 0 10, 1 20, 2 30
}
```

code: ...\s03\ review examples 4 to 9

switch statement

Ref: https://golang.org/ref/spec#Switch_statements

"Switch" statements provide multi-way execution. An expression or type specifier is compared to the "cases" inside the "switch" to determine which branch to execute.

There are two forms: expression switches and type switches.

expression switches

```
switch tag {
	case 0, 1, 2, 3: s1()
	case 4, 5, 6, 7: s2()
	default: s3()
}

switch x := f(); {		// missing switch expression means "true"
	case x < 0: return -x
	default: return x
}

switch {
	case x < y: f1()
	case x < z: f2()
	case x == 4: f3()
}
```

Type switches: A type switch compares types rather than values.

```
switch x.(type) {
	// cases
}
```

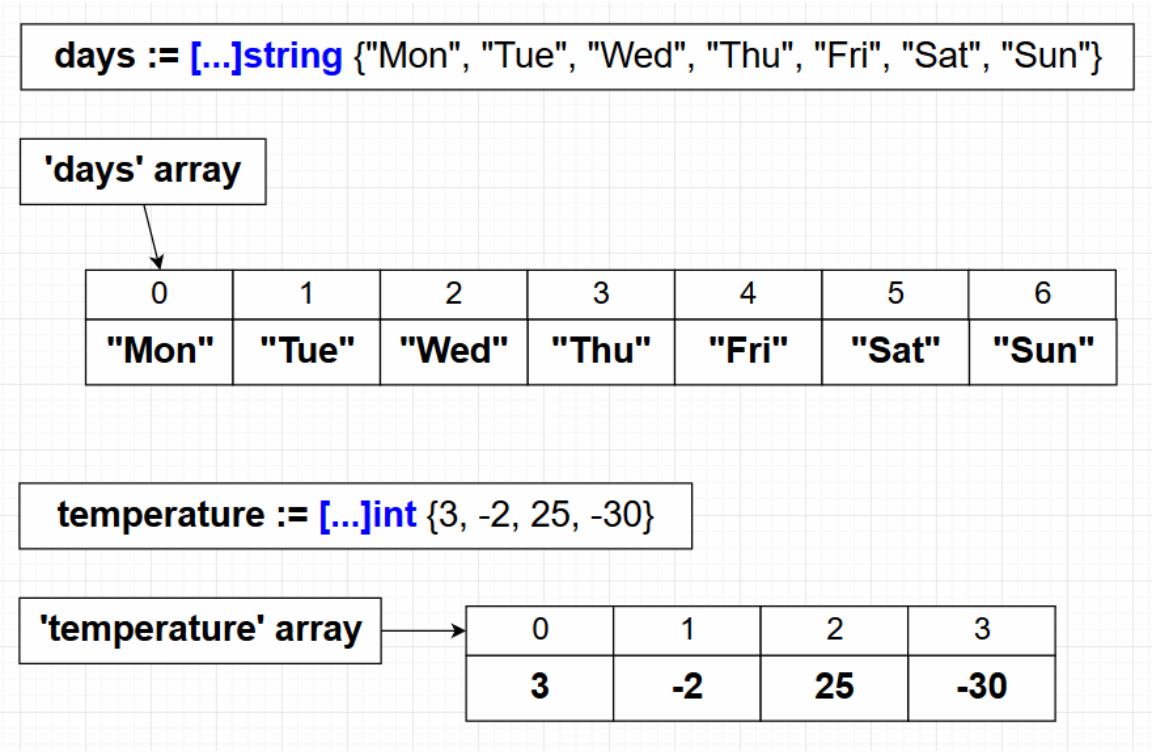
code: ...\s03\ review examples 10 to 16

SECTION 4 – ARRAYS & SLICES

Arrays

Ref: https://golang.org/ref/spec#Array_types

An **array** is **a numbered sequence of elements of a single type**, called the element type. The number of elements is called the **length** and is never negative.



Different ways to declare and initialize an array:

- nums := [...]int{10, 15, 30}
- nums := [3]int{10, 15, 30}
- var nums [3]int = [3]int{10, 15, 30} *// redundant*

Static data structures: fixed length; their length cannot change;
Dynamic data structures: can dynamically grow and shrink;

Composite Types:
1. **arrays** (static; homogeneous; rarely used directly)
2. **slices** (dynamic)
3. **structs** (static; heterogeneous)
4. **maps** (dynamic)

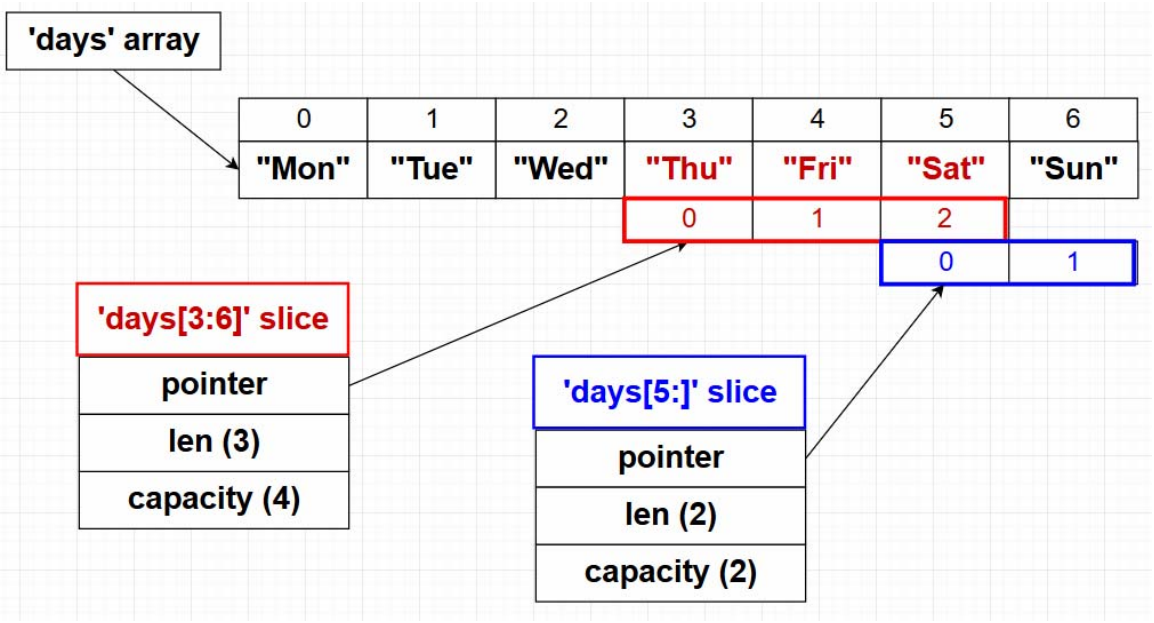
code: ...\s04\ review examples 1 to 6

Slice

Ref: https://golang.org/ref/spec#Slice_types

A **slice** is **a descriptor** for **a contiguous segment of an underlying array** and provides access to a numbered sequence of elements from that array. A slice type denotes the set of all slices of arrays of its element type. The value of an uninitialized slice is nil.

- A slice is **a segment** of an array; it's **indexed**;
- it has a **length**; it has a **capacity**;
- 'array' cannot change its size;
- 'slice' can change its size, but only up to the maximum length of its underlying array.



code: ...\s04\ review examples 7 to 15

assignment: ...\s04\16_slices_operations\main.go

Implementing basic operations with slice.
Use the slicing techniques you've learned so far to implement the following basic operations with slices:

- Append (a=[1 2 3] & b=[4 5 6 7] -> c=[1 2 3 4 5 6 7])
- Copy (a=[1 2 3] -> b=[1 2 3])
- Cut (a=[1 2 3 4 5 6 7] -> a=[1 2 5 6 7])
- Delete (with preserving order) (a=[1 2 5 6 7] -> a=[1 5 6 7])
- Delete (without preserving order) (a=[1 2 3 4 5 6 7] -> a=[1 2 7 4 5 6])
- Expand (a=[1 2 7 4 5 6] -> a=[1 2 0 0 0 0 7 4 5 6])
- Extend (a=[1 2 0 0 0 0 7 4 5 6] -> a=[1 2 0 0 0 0 7 4 5 6 0 0 0])
- Insert (a=[1 2 0 0 0 0 7 4 5 6 0 0 0] -> a=[1 2 0 9 10 0 0 0 7 4 5 6 0 0 0])

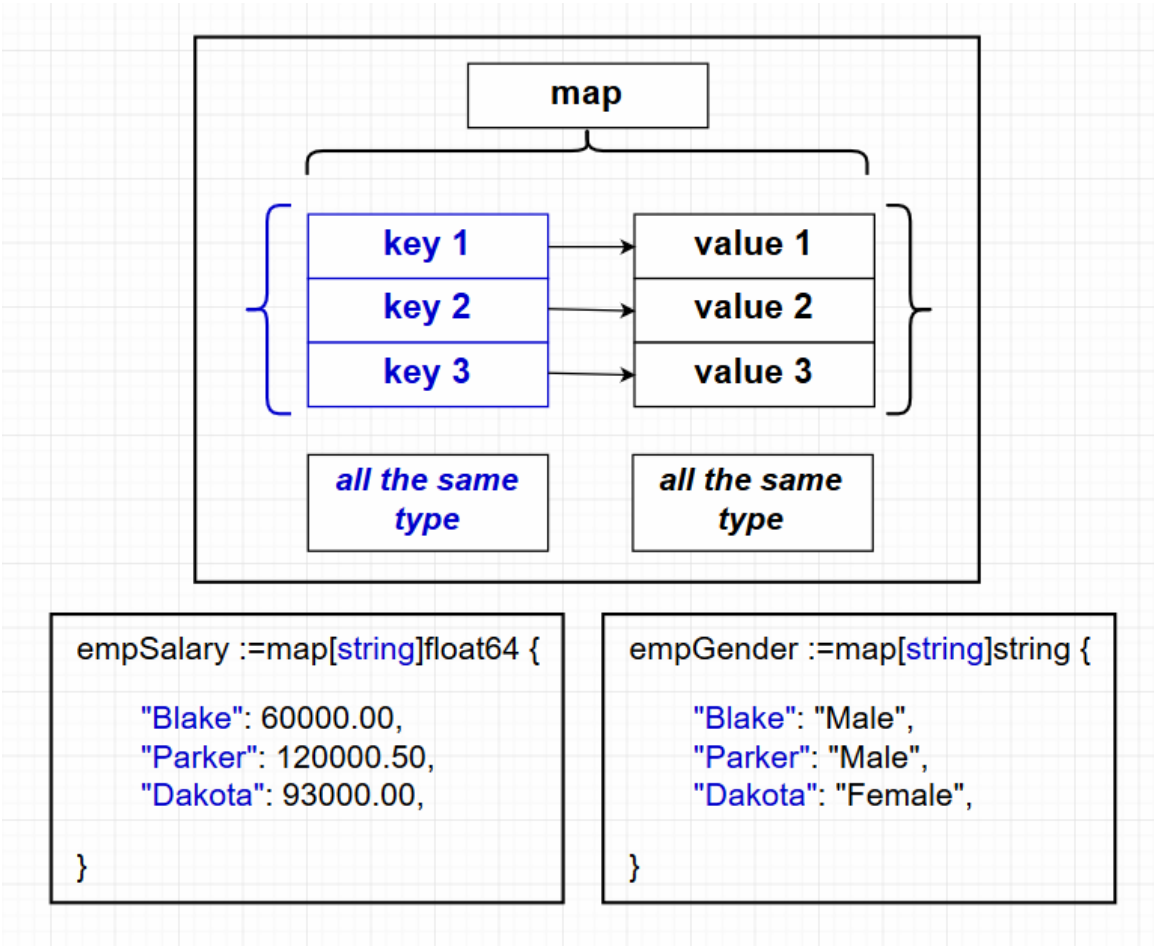
Note: More ‘slice’ examples will be reviewed in the function section.

SECTION 5 – MAPS

Ref: https://golang.org/ref/spec#Map_types

- A map type is written as **map[k]v** (key/value);
- an **unordered** collection of key/value pairs;
- all the **keys** are of the same type;
- all the **values** are of the same type;
- type of keys and values could be different;
- map is a reference type; it can be compared with 'nil';
- The zero value for a map type is nil;

- **hashtable** is the underlying data structure of maps;
  Ref: https://en.wikipedia.org/wiki/Hash_table
- **array** is the underlying data structure of hash tables;
- **array -> hash table -> map**
- In a hashtable, all the **keys are distinct**, therefore values can be quickly retrieved;



Assume that 'days' is a map of strings to ints; It can be written in these forms:

1. days := **make(map[string]int)** (preferred form)

2. var days = make(map[string]int)

3. days := **map[string]int{}**

4. var days map[string]int
   Avoid this form, otherwise you would have to allocate memory to it using the following block, otherwise you would get this runtime error: "panic: assignment to entry in nil map"

   ```
   if days == nil {
           days = make(map[string]int)
   }
   ```

5. days := **map[string]int{** //declare and initialize; composite literal
   "Sun": 1,
   "Mon": 2,
   }

- ***code: ...\s05\01_maps_hashtables\main.go***
- ***assignment: ...\s05\02_maps_hashtables_generic\main.go***
  Improve the previous example by making it more generic.

- ***code: ...\s05\03_maps_make\main.go***
- ***assignment: ...\s05\04_maps_pow\main.go***
  implement i/i^2 using a map (example 2:4, 3:9, 4:16, ...)

- ***code: ...\s05\05_maps_employee\main.go***
- ***assignment: ...\s05\06_maps_sort\main.go***
  Maps are unordered. Write a program to use a "map[string]float64" data map and sort it.

- ***assignment: ...\s05\07_maps_unique\main.go***
  Create a slice of string, including some repeating words. Create a map that contains only the distinct words in our slice.

- ***assignment: ...\s05\08_maps_search\main.go***
  Create a map for data like "Golang": ".go" and search for some values.

- ***code: ...\09_maps_composite_literal\main.go***

- ****assignment: ...\s05\10_maps_unicode\main.go***
  In the following text, count the number of distinct Unicode code point.
  "How 你 cómo お元 Wie geht 잘 How 你 có お元 Wie geht 잘 Ho có Wie"
  map is a good data structure to address this assignment.

Note: More ‘map’ examples will be reviewed in the function section.

SECTION 6 – FUNCTIONS, Part 1 - Basics

Ref: https://golang.org/ref/spec#Function_types

A function is an independent section of code that maps zero or more input parameters to zero or more output parameters.

A **function declaration** contains:
- 'func' keyword;
- name;
- parameter-list; parameters are local to the body of the function;
- result-list (optional-results may be named);
- body;

```
func name(parameter-list) (result-list) {
    body
}
```

Function signature (type):
- parameter-list has the same type and sequence;
- result-list has the same type and sequence;
- Names of parameters and results can be different.

```
package main
import "fmt"

func main() {
    fmt.Println("red")
    fmt.Println("blue")
    fmt.Println("yellow")

    fmt.Println("one")
    fmt.Println("two")
}
```

```
package main
import "fmt"

func main() {
    testCase1()
    testCase2()
}
```

```
func testCase1() {
    fmt.Println("red")
    fmt.Println("blue")
    fmt.Println("yellow")
}

func testCase2() {
    fmt.Println("one")
    fmt.Println("two")
}
```

code: ...\s06\ review examples 1 to 4

Variadic function: We can use **…type** as the last parameter of a function to show that a function can deal with a variable number of parameters of that type (zero to several).

- ***code: ...\05_func_variadic_1\main.go***
- ***assignment: ...\s06\06_func_variadic_2\main.go***
  Write a program to pass a slice of string to a function to loops through the elements and print them.

- ***assignment: ...\s06\07_func_slice_del_empty\main.go***
  Write a function to accept a slice of string, remove blank strings and return the final slice.

- ***assignment: ...\s06\08_func_slice_stack\main.go***
  Use 'slice' and variadic functions to implement a simple stack.

- ***code: ...\s06\09_func_map_of_maps\main.go***
- ***assignment: ...\s06\10_func_map_player\main.go***
  Using 'map of maps' implement the following functionality:
  1. addPlayer("firstName", "lastName")
  to add a new player to our map
  2. hasPlayer("firstName", "lastName") - returns bool
  to check if a pair of firstName/lastName exist in our map.

SECTION 7 – POINTERS

Ref: https://golang.org/ref/spec#Pointer_types

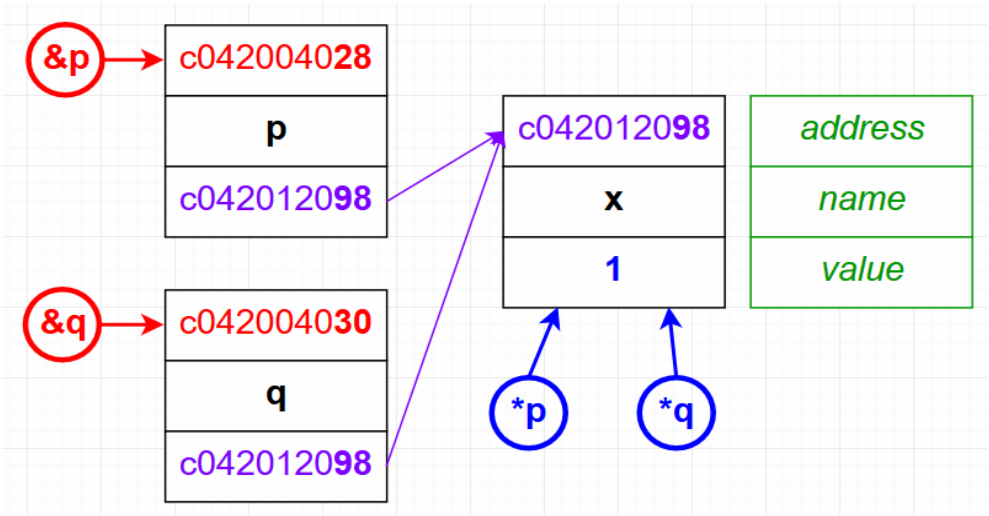
- **Pointer**: the location at which a value is stored.
- **Pointer value**: the address of a variable.
- With a pointer, we can read or update the value of a variable indirectly.
- Not every value has an address, but every variable does.
- **Pointer aliasing**: Using a pointer is like creating a new alias for a variable.
  x := 1; p := &x // *p is an alias for x

  &x (address of x): yields a pointer to the integer variable x that is of a value of type *int.

  p := &x indicates that 'p points to x (address of x)'
- With Go, pointers are **mainly used with 'structs'** (rarely used with primitive types).
- Both of the following create a pointer:
  var p ***int**
  p := **new(int)** // p, of type *int, points to an unnamed int variable

```
x := 1
p := &x
q := &x

fmt.Printf(" x  =>  type=%T value=%d  \n", x, x)
fmt.Printf("&x =>  type=%T value=%x  \n", &x, &x)

fmt.Printf("*p  =>  type=%T value=%d  \n", *p, *p)
fmt.Printf(" p  =>  type=%T value=%x  \n", p, p)
fmt.Printf("&p =>  type=%T value=%x  \n", &p, &p)

fmt.Printf("*q =>  type=%T value=%d  \n", *q, *q)
fmt.Printf(" q =>  type=%T value=%x  \n", q, q)
fmt.Printf("&q =>  type=%T value=%x  \n", &q, &q)
```

```
 x =>  type=int value=1
&x =>  type=*int value=c042012098

*p =>  type=int value=1
 p =>  type=*int value=c042012098
&p =>  type=**int value=c042004028

*q =>  type=int value=1
 q =>  type=*int value=c042012098
&q =>  type=**int value=c042004030
```



- ***code: ...\s07\01_pointers_demo\main.go***

- ***code: ...\s07\ review examples 2 to 6***

Note: The following examples cover call-by-value and call-by-reference.
More examples will be introduced in the upcoming sections.

- ***code: ...\s07\ review examples 7 to 9***

- ***assignment: ...\s07\10_pointers_slice\main.go***
  In previous sections we learned that a slice is a pointer to its underlying array. Create an array of strings and a slice of string. Using pointers prove that the slice uses the same address region of the array.

SECTION 8 – FUNCTIONS, Part 2

Ref: https://golang.org/ref/spec#Function_types
Ref: https://golang.org/ref/spec#Function_literals

Function Literal

- **Function Literal**: an inline function without a name.
- In other languages they're called '**anonymous function**', '**lambda**', ...
- ****code: ...\s08\ review examples 1 to 3***

Closure

- **Closure**: a function that captures the state of the surrounding environment.
- So far for functions A and B to have access to variable x, x must be in scope of both (for instance, in the package scope). With closure, both A and B can access x (implicitly limiting the scope of x), without a need to be in package scope.
- Closure and recursion form the basis of **functional programming**.

- ****code: ...\s08\04_func_closure_1\main.go***
- ****assignment: ...\s08\05_func_closure_2\main.go***
  Using 'closure' write an addBy() function to add x to the previous value of the function. For instance,
  addCounter := addBy(); Println(addCounter(2)); Println(addCounter(-1))
  Also write a multBy() function to multiply by x. For instance,
  multCounter := multBy(); Println(multCounter(3)); Println(multCounter(-2))

Callbacks

- **Callback**: passing the evaluated value of a function A as a parameter to a function B.
- They're valuable but increase the complexity of your program.

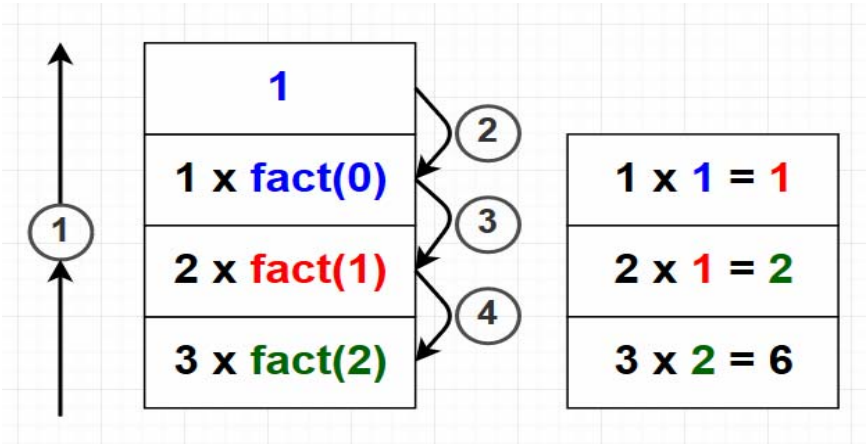
- ***code: ...\s08\06_func_callback\main.go***
- ***assignment: ...\s08\07_func_callback_variadic\main.go***
  Write a callback with two parameters:
  1. A function that accepts a variable number of integers and return an integer. In practice this could be a function to add or multiply, ... some numbers.
  2. The list of integers (with a variable number of elements) to be added, or multiplied, ...
  Use the techniques learned in the last example, as well the ones to handle a variable number of parameters.

Recursion

Ref: https://en.wikipedia.org/wiki/Recursion_(computer_science)

- Recursion in computer science is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem (as opposed to iteration).[1] The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science.
- Most computer programming languages support recursion by allowing **a function to call itself from within its own code**.

- ***code: ...\s08\08_func_recursion_factorial\main.go***



- ****assignment: ...\s08\09_func_recursion_fibonacci\main.go***
  In mathematics, the Fibonacci numbers are the numbers in the following integer sequence, called the Fibonacci sequence, and characterized by the fact that every number after the first two is the sum of the two preceding ones:
  0, 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , 34 , 55 , 89 , 144 , ...
  Implement the Fibonacci series using recursion.
  Ref: https://en.wikipedia.org/wiki/Fibonacci_number

Defer

Ref: https://golang.org/ref/spec#Defer_statements
https://blog.golang.org/defer-panic-and-recover

A 'defer' statement defers the execution of a function until the surrounding function returns.

The best place for a defer statement is right after the allocated resource.
Example: f, _ := os.Open(fileName)
defer f.Close()

- ****code: ...\s08\10_func_defer_1\main.go***

- ****assignment: ...\s08\11_func_defer_2\main.go***
  Without running the following program find its output.

- ****assignment: ...\s08\12_func_defer_sqaure\main.go***
  Use 'defer' statement to write a square function that hijacks the correct return type when the input parameter is 2 or 4.
  *For 2 => return 6; For 4 => return 20; formula: (x*x)+x*
  *For other values return (x*x)*

panic / recover

Ref: https://blog.golang.org/defer-panic-and-recover

- Go catches most of errors at compile time.
- Others (such as nil pointer, out-of-bounds array access, …) require to be checked at run time. When Go detects these mistakes at runtime, it **panics**.
- When Go panics, **normal execution stops**, and the program exits with a log message.
- When a panic occurs, all deferred functions are run in reverse order (in the stack).
- The usage of Go's panic mechanism is different than exceptions in other languages. Panic is normally used for critical errors that cause the program to crash.
- The **panic()** function stops the ordinary flow of control and begins panicking.
- The **recover()** function regains control of a panicking process.

- ***code: ...\s08\13_func_panic_normal\main.go***
- ***code: ...\s08\14_func_panic_recover\main.go***

SECTION 9 – STRUCTS

Ref: https://golang.org/ref/spec#Struct_types

- **struct** is an aggregate data type that is used to **define a new type** which consist of a number of properties/**fields** (named values of types);

  ```
  type strcut_name struct {
    field1 type1
    field2 type2
    ...
  }
  ```

- **Object oriented programming languages** such as Java use concepts such as 'class', 'interface', 'encapsulation', 'inheritance', 'polymorphism', 'overriding', and others. Go's approach to implement these concepts is different.

```
type player struct {
    name  string
    sport string
    age   int
}
```

output:
Player 1= {Leo Messi Soccer 30}
name=Leo Messi age=30

```
player1 := player{"Leo Messi", "Soccer", 30}

fmt.Println("Player 1=", player1)
fmt.Printf("name=%s age=%d\n", player1.name, player1.age)
```

- ****code: ...\s09\ review examples 1 to 3***

- ****assignment: ...\s09\04_struct_pointers\main.go***
  Structs and pointers - can variables of a custom-type (using struct) be pointed at by pointers? Demonstrate by code.

- ****assignment: ...\s09\05_struct_compare\main.go***
  Comparing structs - considering the following struct write a simple program to compare two variables of this type. Write two versions of the comapre() function: call by value and call by ref.

- ****code: ...\s09\ review examples 6 to 8***

- ****assignment: ...\s09\09_struct_embedded_anonymous\main.go***
  embedded anonymous structs:
  change the previous example to use an anonymous field of type of struct.

- ****assignment: ...\s09\10_struct_embedded_multiple\main.go***
  Multiple embedded structs - write a program to demonstrate multiple embedded structs. Something like (info -> person -> student), or any other structures of your choice.

- Go does not have classes. However, you can define methods on struct types.
- The **method receiver** appears in its own argument list between the func keyword and the method name.

  By placing "(m movie)" before the function name, we're associating this function with the type "movie", meaning that any variable of type "movie" can also use this function.

```
type movie struct {
  name  string
  actor string
}
func (m movie) fullInfo() string {
  return m.name + " " + m.actor
}
…
m := movie{"The Godfather", "Marlon Brando"}
fmt.Println(m.fullInfo())
```

- Previously we learned the Types of Go programs:
  Executables (can run directly, e.g., main)
  Libraries (packages of code that can be used in other programs)
  "...\s09**12_struct_exported**\main.go" demonstrates an example of a library.

- **Exported struct / field**: any struct or field name that **starts with a capital letter can be imported** by other programs.

- As a side note, any struct or field name that starts with a capital letter is **automatically visible within its package**.

- ****code: ...\s09\ review examples 11 to 12***

- Method **overloading & overriding** in other languages *(pseudo-code)*

```
class ParentOp {
  // some fields ...
  int add(int, int)          //1
  int add(int, int, int)     //2 - method overload
  float add(float, float)    //3 - method overload
}

class ChildOp extends ParentOp  {
  // some fields ...
  int add(int, int)          //4-method override/overwrite
}
```

- **Method/Function overloading** that is writing more than one function with the same function name, but different signature (signature = parameter-list + return-list) **is not allowed in Go** and the compiler will yield an error.
  This is because function overloading forces the runtime to do additional type matching that reduces performance;

- Go uses **receivers** for implementing this functionality.
  Assuming that 'movie' and 'imdb' are two different structs, the first fullInfo() is associated to the 'movie' struct and second one is associated to the 'imdb' struct.

```
func (m movie) fullInfo() string {
  return m.name + "-" + m.actor
}

func (i imdb) fullInfo() string {
  return strings.ToLower(i.movie.name + "-" + i.movie.actor)
}
```

- ***code: ...\s09\13_struct_method-overload\main.go***
- ***code: ...\s09\14_struct_receiver_pointer\main.go***

SECTION 10 – INTERFACES

Ref: https://golang.org/ref/spec#Interface_types

- **Inheritance & Polymorphism in object oriented programming languages**:

  Interface is similar to class. It is a collection of abstract methods.
  Writing an interface is similar to writing a class. But a class describes the attributes and behaviours of an object. And an interface contains behaviours that a class implements.

  Polymorphism is the capability of a method to do different things based on the object that it is acting upon.
  Polymorphism allows you define one interface and have multiple implementations.

```
(pseudo-code)
interface Car {
  start();
  stop();
}

class BMW implements Car {
  start() {print("Moving Fast");}
  stop()  {print("Stopping...");}
  // other properties and methods ...
}

class Tesla implements Car {
  start() {print("Moving Super Fast");}
  stop()  {print("Hard to stop...");}
  // other properties and methods ...
}

main() {
  Car c;
  BMW b = new BMW();
  Tesla t = new Tesla();

  c = b;  c.start();
  c = t;   c.start();
}
```

- In Go, an **interface** defines a set of methods that do not contain code (abstract methods); an interface **cannot contain variables**;

- Interface format:
```
type Name interface {
  method1(param_list) return_type      // no body
  method2(param_list) return_type      // …
}
```

- Interfaces in Go normally contain **less than four methods**.
- Real world example of interface: A remote control is a type of interface, you use the buttons, without knowing what's happening under the hood.

```
type rectangle struct {
    w, l int //width & length
}
func (c *rectangle) area() int {
    return c.w * c.l
}
```

```
type square struct {
    s int //side
}

func (c *square) area() int {
    return c.s * c.s
}
```

```
type shape interface {
    area() int
}
func info(s shape) {
    fmt.Printf("%d - plus other info", s.area())
}
```

- We have a new type called '**shape**'.
- Any other type in the file with a function '**area() int**' (no parameter, and return type of int) is also a member of type 'shape'.
- Since you're now a member of this interface, you can also call the **info()** function.

```
r1 := rectangle{2, 3}
fmt.Println(r1.area())
info(&r1)

s1 := square{3}
fmt.Println(s1.area())
info(&s1)
```

```
var shapes [2]shape

shapes[0] = &r1
shapes[1] = &s1

info(shapes[0])
info(shapes[1])
```

- ****code: ...\s10\ review examples 1 to 2***

Conversion / Type Casting

Conversion

- **Narrowing**: converting a larger type to a smaller (like float64 to int);
- **Widening**: converting a smaller type to a larger (like int to float64);

- Conversion using '**strconv**' package: The most common numeric conversions are **Atoi** (string to int) and **Itoa** (int to string).
  ref: https://golang.org/pkg/strconv/

- ****code: ...\s10\03_interface_conversion_1\main.go***

Conversion using **'strconv'** package

- Converting to string and int:
  func **Atoi**(s string) (int, error) (string to int)
  func **Itoa**(i int) string (int to string)

- Converting **strings to values**:
  func **ParseBool**(str string) (bool, error)
  func **ParseFloat**(s string, bitSize int) (float64, error)
  func **ParseInt**(s string, base int, bitSize int) (i int64, err error)
  func **ParseUint**(s string, base int, bitSize int) (uint64, error)

- Converting **values to strings**
  func **FormatBool**(b bool) string
  func **FormatFloat**(f float64, fmt byte, prec, bitSize int) string
  func **FormatInt**(i int64, base int) string
  func **FormatUint**(i uint64, base int) string

- ****code: ...\s10\04_interface_conversion_2\main.go***

- **Type assertion**: detecting and converting the type of an **interface variable**.

  The following interface type variable 'message' can contain a value of any type; we need **a way to detect this dynamic type**; we'll use **variable.(TYPE)**

  ```
  var message interface{} = 10                    or
  var message interface{} = "Hello"
  s, ok := message.(string)                       if ok { ... }
  ```

- ****code: ...\s10\05_interface_assertion\main.go***

Package 'sort': Package sort provides primitives for sorting slices and user-defined collections.
Ref: https://golang.org/pkg/sort/

- func **Ints**(a []int) { Sort(IntSlice(a)) }
  sorts a slice of ints in increasing order.
- func **Strings**(a []string) { Sort(StringSlice(a)) }
  sorts a slice of strings in increasing order
- …

- ***code: ...\s10\ ...\s10\06_interface_sort1\main.go***

type Interface: a type, typically a collection, that satisfies sort.Interface can be sorted by the routines in the 'sort' package. The methods require that the elements of the collection be enumerated by an integer index.
Ref: https://golang.org/pkg/sort/#Interface

```
type Interface interface {
        Len() int              // number of elements in the collection.
        Less(i, j int) bool    // reports whether the element with index i
                               // should sort before the element with index j.
        Swap(i, j int)         // swaps the elements with indexes i and j.
}
```

- func **Sort**(data *Interface*)
  *// sorts data. It makes one call to data.Len to determine n, and O(n*log(n)) calls to data.Less and data.Swap. The sort is not guaranteed to be stable.*

- func **Reverse**(data *Interface*) Interface *// returns the reverse order for data.*

- func **IsSorted**(data *Interface*) bool *// reports whether data is sorted.*

- Any type (such as 'IntSlice', 'StringSlice', 'Float64Slice', …) that contains Len(), Less(), and Swap(), implements the "Interface" interface.

type IntSlice: attaches the methods of Interface to []int, sorting in increasing order.
Ref: https://golang.org/pkg/sort/#IntSlice

- **type IntSlice []int**
  - func (p IntSlice) **Len**() int { return len(p) }
  - func (p IntSlice) **Less**(i, j int) bool { return p[i] < p[j] }
  - func (p IntSlice) **Swap**(i, j int) { p[i], p[j] = p[j], p[i] }
  - func (p IntSlice) **Sort**() { Sort(p) }
    // Sort() is a convenience method.
  - func (p IntSlice) **Search**(x int) int { return SearchInts(p, x) }
    // Search returns the result of applying SearchInts to the receiver and x.

type StringSlice: attaches the methods of Interface to []string, sorting in increasing order.
Ref: https://golang.org/pkg/sort/#StringSlice

- **type StringSlice []string**
  - func (p StringSlice) **Len**() int { return len(p) }
  - func (p StringSlice) **Less**(i, j int) bool { return p[i] < p[j] }
  - func (p StringSlice) **Swap**(i, j int) { p[i], p[j] = p[j], p[i] }
  - func (p StringSlice) **Sort**() { Sort(p) }
    // Sort is a convenience method.
  - func (p StringSlice) **Search**(x string) int { return SearchStrings(p, x) }
    // Search returns the result of applying SearchStrings to the receiver and x.

- By using StringSlice() in "sort.StringSlice(s).Sort()", we have access to its methods. We've used Sort() to sort our slice of string.

- Also check **type Float64Slice**
  Ref: https://golang.org/pkg/sort/#Float64Slice

- ****code: ...\s10\ ...\s10\07_interface_sort2\main.go***
- ****code: ...\s10\ ...\s10\08_interface_sort_struct\main.go***

String() method

Ref: https://golang.org/doc/effective_go.html#printing - then search for "String()"

If you want **to control the default format for a custom type**, all that's required is to define a method with the signature **String() string** on the type.

For our simple type T, that might look like this:

```
func (t *T) String() string {
	return fmt.Sprintf("%d/%g/%q", t.a, t.b, t.c)
}
fmt.Printf("%v\n", t)
```

- ***code: ...\s10\09_interface_sort_tostring\main.go**

SECTION 11 – CONCURRENCY

Goroutines

- A **goroutine** is a **lightweight, independently executing** function (managed by the Go runtime) that **can run concurrently with other activities/functions**.
- When the program starts, the **main routine** gets created by default.
- The main routine is the place that we can create **child routines**.
- Routines (including main) are independent.
- Goroutines cannot stop one another, but can communicate and send requests (including stop requests) to one another.
- A goroutine has its own stack which grows and shrinks.
- Goroutines are lightweight and many of them can be easily created and run at the same time.

- Use the '**go**' statement to **create new goroutines**.
  authenticateUser() // **blocking**: program waits for the function to return
  go authenticateUser() // **non-blocking**: program doesn't wait for the function to return (moves to the next line)

- ****code: ...\s11\01_concurrency_goroutine\main.go***

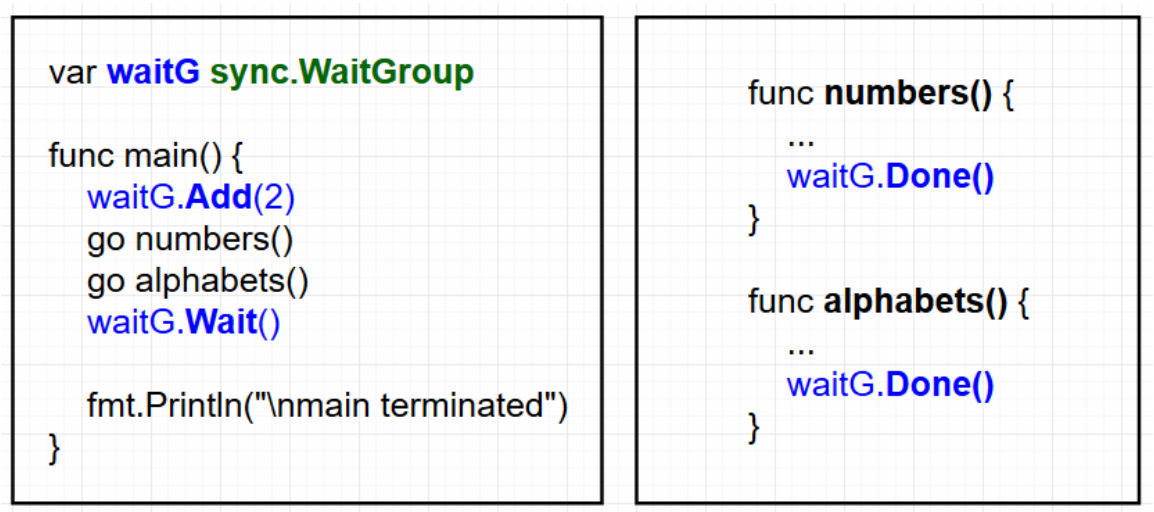
type WaitGroup

ref: https://golang.org/pkg/sync/#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.
- Then each of the goroutines runs and calls Done when finished.
- At the same time, Wait can be used to block until all goroutines have finished.
- A WaitGroup must not be copied after first use.

- func (wg *WaitGroup) **Add**(delta int)
  Add adds delta, which may be negative, to the WaitGroup counter. If the counter becomes zero, all goroutines blocked on Wait are released. If the counter goes negative, Add panics.
- func (wg *WaitGroup) **Done**()
  Done decrements the WaitGroup counter by one.
- func (wg *WaitGroup) **Wait**()
  Wait blocks until the WaitGroup counter is zero.

- Example:



```
var waitG sync.WaitGroup

func main() {
    waitG.Add(2)
    go numbers()
    go alphabets()
    waitG.Wait()

    fmt.Println("\nmain terminated")
}
```

```
func numbers() {
    ...
    waitG.Done()
}

func alphabets() {
    ...
    waitG.Done()
}
```

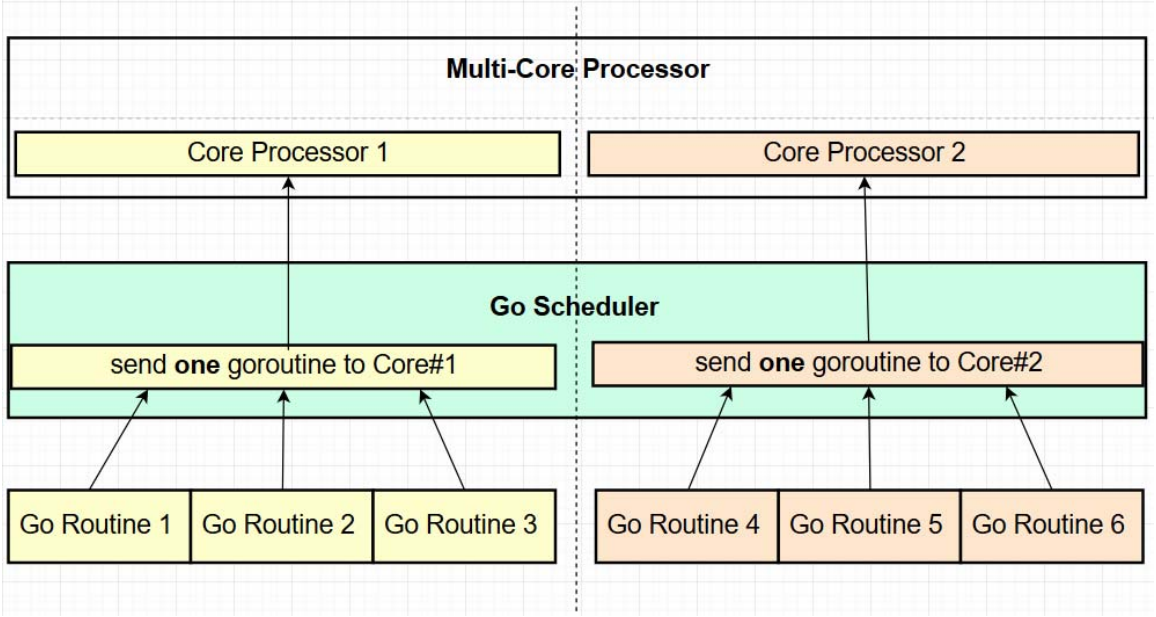
- ****code: ...\s11\02_concurrency_waitgroup\main.go***

Concurrency & Parallelism are different

- **Concurrency** is like **"inhaling"** and **"talking"**. You cannot do both at the same time. You take turns, but it's that fast that it's not noticeable.
- **Parallelism** is like **"breading"**, **"sleeping"**, and **"dreaming"**. All can be done at the same time with no short interruptions.

- A **blocking call**: any statement (normally call to another routine) that its execution takes long, and the rest of the code waits for it to complete is called a blocking call. For instance a call to a busy server that the server response may be immediate or may take several minutes.
- With **one CPU**, the scheduler simulates concurrency by executing a slice of a goroutine one at a time. If one thread/goroutine is blocked, another one is executed.
- With **multiple CPUs**, multiple goroutines are executed at the same time at different CPUs.

- Rob Pike-Concurrency Is Not Parallelism: https://www.youtube.com/watch?v=cN_DpYBzKso



- ****code: ...\s11\03_concurrency_parallelism\main.go***

Race Condition

- **Race condition** happens when more than one process try to access the same resource.
- > go run **-race** main.go *(to find possible race conditions)*
  It prints the results, and gives some information about the race, including the following:
  counter: 10Found 2 data race(s)
  exit status 66

- ****code: ...\s11\04_concurrency_race_condition\main.go***

mutex (mutual exclusion)

Ref: https://en.wikipedia.org/wiki/Mutual_exclusion

- **Mutex (mutual exclusion)** is a property of concurrency control, which is instituted for the purpose of preventing race conditions. It is the requirement that one thread of execution never enter its critical section at the same time that another concurrent thread of execution enters its own critical section.

- ****code: ...\s11\05_concurrency_mutex\main.go***
  *> go run **-race** main.go (to make sure there are no race conditions)*

Package Atomic

Ref: https://golang.org/pkg/sync/atomic/

- **Package atomic** provides low-level atomic memory primitives useful for implementing synchronization algorithms.
- These functions require great care to be used correctly.
- Except for special, low-level applications, synchronization is better done with channels or the facilities of the sync package.

- Another way of **encapsulating an operation**:
  - func **AddUint64**(addr *uint64, delta uint64) (new uint64)
    Ref: https://golang.org/pkg/sync/atomic/#AddUint64
    AddUint64() atomically adds delta to *addr and returns the new value.

  - func **LoadUint64**(addr *uint64) (val uint64)
    Ref: https://golang.org/pkg/sync/atomic/#LoadUint64
    LoadUint64() atomically loads *addr.

- ***code: ...\s11\06_concurrency_atomic\main.go***
  *> go run **-race** main.go (to make sure there are no race conditions)*

SECTION 12 – CHANNELS

- **Channels** connect goroutines and enable them to communicate and synchronize (their execution).

- Channels are **typed**, means that the data that is passed into a channel (shared between multiple goroutines) must be of the same type.

- To create a channel:
  c = make(**chan** int) // unbuffered channel - c has type 'chan int'
  c = make(chan int, 4) // buffered channel with capacity 4

- Channels are **reference types**. The zero value of a channel is nil.

- You can compare two channels (of the same type) using ==.
  true means they're referring to the same channel variable.
  A channel may also be compared to nil.

- **Two operations of a channel**: **send** and **receive** (collectively known as communications).
  c <- x // send - sends value x to channel c
  x = <- c // receive - reads from the channel c and saves in variable x
  <- c // receive - result is discarded

- Channels support a **third operation**, close().
  close() sets a flag indicating that no more values will be sent on this channel;
  To close a channel: **close(ch)**

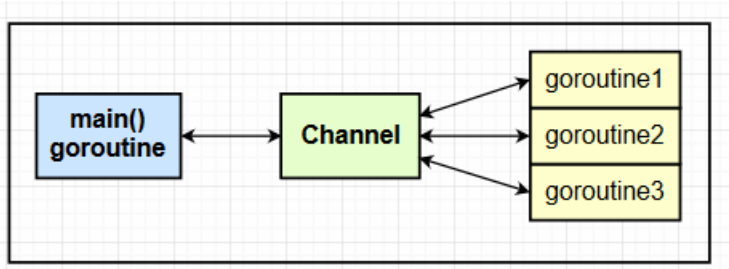
- **On a closed channel**:
  - Further '**send**' operations will panic.
  - Further '**receive**' operations receive the already sent values, until no more values are left.
    After that completion point, any receive operations immediately concludes and yields the zero value of the channel's element type.

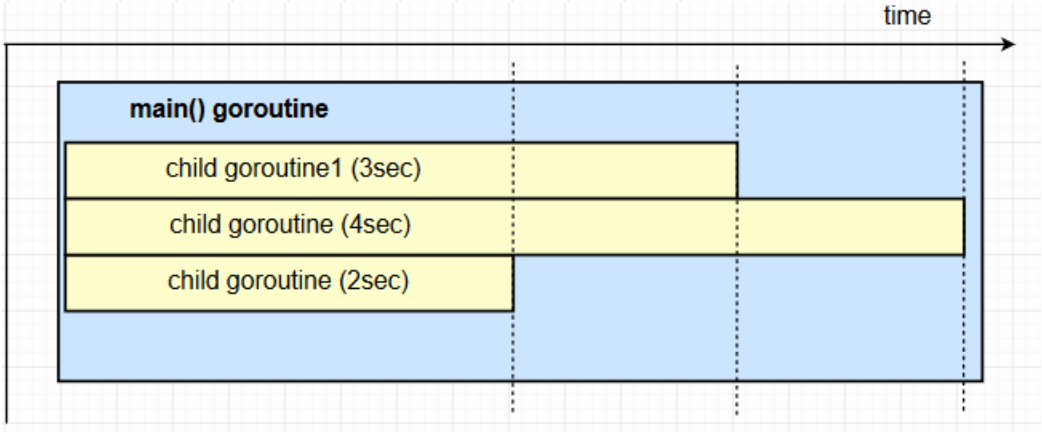
```
func message1(c chan string) { time.Sleep(3* time.Second);   c <-"msg1, delay 3sec" }
func message2(c chan string) { time.Sleep(4* time.Second);   c <-"msg2, delay 4sec" }
func message3(c chan string) { time.Sleep(2* time.Second);   c <-"msg3, delay 2sec" }
```

```
func main() {

    var c = make(chan string)

    start := time.Now()

    go message1(c);    go message2(c);    go message3(c)

    fmt.Println(<-c);     fmt.Println(<-c)      fmt.Println(<-c)

    close(c)

    elapsed := time.Since(start)
    fmt.Printf("\ntime elapsed: %s \n", elapsed)
}
```

```
Output:

msg3, delay 2sec
msg1, delay 3sec
msg2, delay 4sec

time elapsed: 4.0002102s
```

```
c <- "write msg to channel"

// reading a msg from channel
msg := <- c
Println (msg)
```





- ***code: ...\s12\01_channel_demo\main.go***
- ***code: ...\s12\02_channel_nil\main.go***
- ***code: ...\s12\03_channel_deadlock\main.go***

Unbuffered (synchronous) Channels

- By default, **communication is unbuffered (synchronous)**;
- It means that a ‘send’ does not complete until a ‘receiver’ accepts the value;
- This **blocking behaviour of unbuffered channels** tells us that there is **no space in the channel for data** (because for any ‘send’ there must be a ‘receive’ and vice versa);
- In short, channel operations (i.e. send/receive) block until the other side is ready;
- **Communication** can be seen as **a form of synchronization** when goroutines share data through a channel that synchronizes (takes turn) to communicate with those goroutines. Clearly unbuffered channels are the perfect candidate for synchronizing communication of multiple goroutines.
- So far we’ve seen several examples of unbuffered channels, but here are some more.
- ***code: ...\s12\04_channel_unbuffered\main.go***
  *> go run **-race** main.go (to make sure there are no race conditions)*
- ***code: ...\s12\05_channel_range\main.go***

Semaphore

Ref: https://en.wikipedia.org/wiki/Semaphore_(programming)

- A **semaphore** is **a variable** (or abstract data type) used to **control access to a common resource shared by multiple processes** in a concurrent system such as a multitasking operating system.

- Semaphores are **a useful tool in the prevention of race conditions**; however, their use is **by no means a guarantee** that a program is free from these problems.

- Semaphores which allow an arbitrary resource count are called **counting semaphores**, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called **binary semaphores** and are used to implement locks.

- ***code: ...\s12\06_channel_semaphore\main.go***

- ***code: ...\s12\07_channel_multipe_receivers\main.go***

- ***assignment: ...\s12\08_channel_uppercase\main.go***
  Write a program that contains two channels that accept texts an convert them to uppercase and lowercase.

Channel Direction

- So far we've seen bidirectional channels. We're going to see two more types of channels (send only and receive only):
  All channels are created bidirectional, but we can assign them to directional channel variables.

  o
```
func bi(c chan int) {          //a "bidirectional" channel
    c <- 10                    //ok
    fmt.Println(<-c)           //ok
}
```

  o
```
func snd(out chan<- int) {     //a "send-only" channel
    out <- 10                  //ok
    // fmt.Println(<-out)      //error - receive from send-only type
}
```

  o
```
func rcv(in <-chan int) {      //a "receive-only" channel
    // in <- 10                //error - send to receive-only type
    fmt.Println(<-in)          //ok
}
```

- **Receive-only channels cannot be closed**, because closing a channel is a way for the sender to say that no more values will be sent to the channel, that doesn't make sense for receive-only channels.

- ****assignment: ...\s12\09_channel_direction\main.go***
  Assuming your current bank balance is 200, write a program containing these functions:
  1) credit() function with a 'bi-directional channel' parameter that sends random numbers (between 1 and 10) to the channel that will be added to the balance.
  2) debit() function with a 'send-only channel' parameter that sends random numbers (between -10 and -1) to the channel that will be subtracted from the balance.
  3) debit() function with a 'receive-only channel' parameter that adds/subtracts the random values (generated in steps 1 & 2) to the balance.
  Here's a sample output that indicates that the balance is randomly changing:
  Press Enter to stop the program ...
  => 200 + (6) = 206
  => 206 + (-10) = 196
  => 196 + (3) = 199
  => 199 + (-2) = 197

Multiplex

- **Multiplexing**: In telecommunications and computer networks, multiplexing is a method by which multiple analog or digital signals are combined into one signal over a shared medium. The aim is to **share a scarce resource**. For example, in telecommunications, several telephone calls may be carried using one wire.
  Ref: https://en.wikipedia.org/wiki/Multiplexing

- To extend this concept to a typical client-server application, many clients may send requests to a server (that is the shared scarce resource), requesting the server to do something for them. The server then processes the received requests and sends responses back to the clients.

- To multiplex operations we need a select statement:

```
select {
  case <- ch1:
        // ...
  case x := <- ch2:
        // ...use x...
  case ch3 <- y:
        // ...
  default:
        // ...
}
```

- The select statement is used to choose from multiple send/receive channel operations.
- The "select" statement blocks until one of the send/receive operations is ready.
- If multiple operations are ready, one of them will randomly be picked.

- ***code: ...\s12\10_channel_multiplex\main.go***

Buffered channels

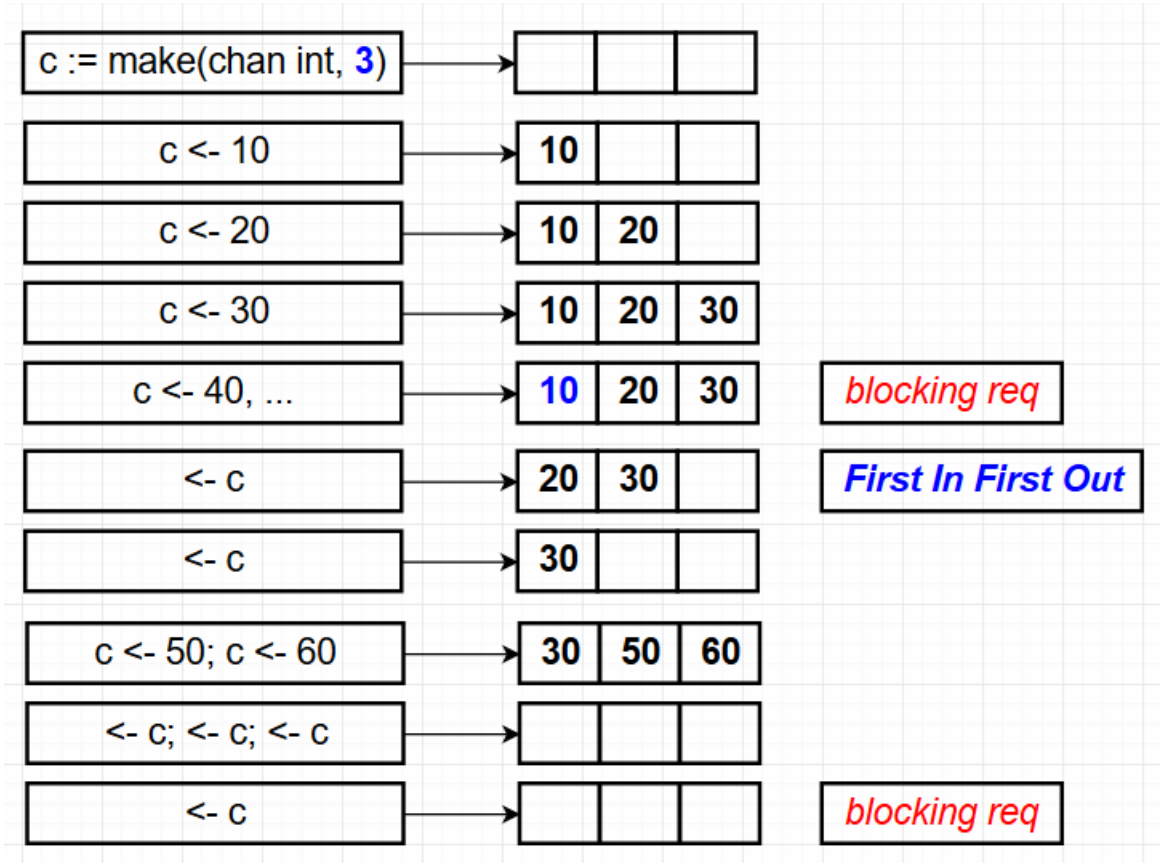
- **Unbuffered Channels** (synchronous / blocking): So far all the channels we discussed were unbuffered.
  This means that both sides of the channel will wait until the other side is ready.
  Capacity of an unbuffered channel is 0 by default. c := make(chan int)

- **Buffered Channels** (asynchronous / non-blocking):
  - the capacity argument of make is bigger than zero. c := make(chan int, 3)
    c := make(chan channelType, channelCapacity)

  - a **send** operation: adds an element to the **end of the queue**;
  - a **receive** operation: removes an element from the **front**;

  - 'sends' to a buffered channel are blocked only when the buffer is full.
    If the channel is **'full'** ->
    'send' operation blocks until a 'receive' operation makes a space available.

  - 'receives' from a buffered channel are blocked only when the buffer is empty.
    If the channel is **'empty'** ->
    'receive' operation blocks until a 'send' operation send a value.



- ****code: ...\s12\11_channel_buffered_demo\main.go***
- ****code: ...\s12\12_channel_buffered_capacity\main.go***

- ****assignment: ...\s12\13_channel_gcd\main.go***
  Write two functions to compute gcd (greatest common divisor) of two integers using ***recursive*** *and* ***channel****.*
  Ref: https://en.wikipedia.org/wiki/Greatest_common_divisor

- ****assignment: ...\s12\14_channel_pipeline_gen\main.go***
  A pipeline connects two (or more) channels operations, so that the result/output of the first channel operations becomes the input of the second channel.
  Write a program containing two functions to simulate a pipeline of two channels that the first function, gen() receives a list of integers and send them to a channel and then the second function, sq() that uses the return type of the first function to calculate the square of each received integer and then send all the values downstream.
  In short: receive integers -> send them to a channel ->
  return the channel -> calculate the square of integers
  Ref: https://blog.golang.org/pipelines

- ****assignment: ...\s12\15_channel_pipeline_words1\main.go***
  Write a pipeline including two goroutines:
  1) The 'newWords' goroutine that produces random words and send them to a channel;
  2) The 'uWords' (uppercase words) goroutine that converts the randomly-generated words to uppercase;
  Use anonymous functions to write your goroutines.

- ****assignment: ...\s12\16_channel_pipeline_words2\main.go***
  Rewrite the previous pipeline using functions and unidirectional Channels.

SECTION 13 – PACKAGES & DOCUMENTATION

Packages

- Ways to promote reusability and readability:
  functions, **structs**, **interfaces**, and **packages** (subject of this section)

- **Benefits of packages**: organizing code; reducing the chance of having overlapping names; optimizing the compiler by requiring only those libraries that are absolutely needed.

- We already know that **every "executable" go application must contain a "main()"** function that is the entry point for execution and resides in the "main package".

- We also know that **exported functions/names** start with a capital letter so that they can be accessed from other packages.

- As we've seen, import "fmt" imports the fmt package that contains methods such as Println, Scanln, ...

- **Importing custom packages** : 'import' can also be used to import custom packages. The only difference is that the path of the package needs to be specified.

- It's illegal in Go to import a package without using it anywhere in the code.
  For instance if you have an **import "fmt"** in your code, you must also use this package in your code by a statement such as **fmt.Println("a message")**.

- During the development phase, if you need packages to stay in your file, but not used, use the _ blank identifier. (example: _ "math")

- Each package acts as a separate 'name space'.

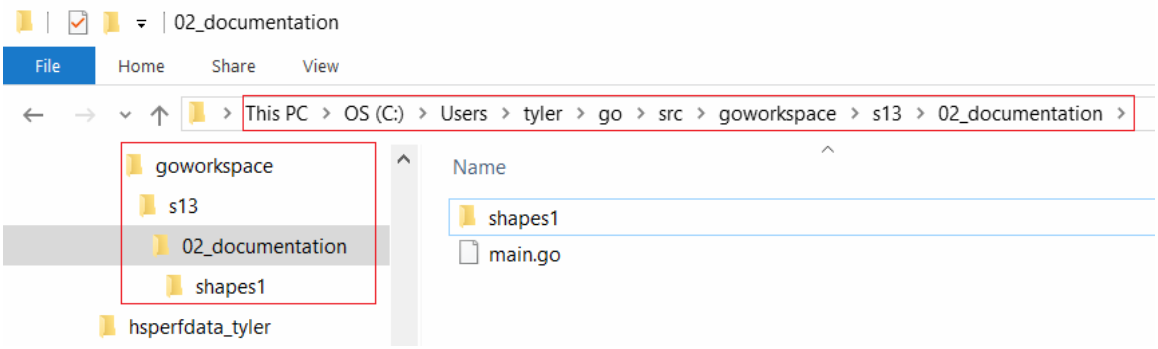
- ***code: ...\s13\01_packages\main.go**

Documentation

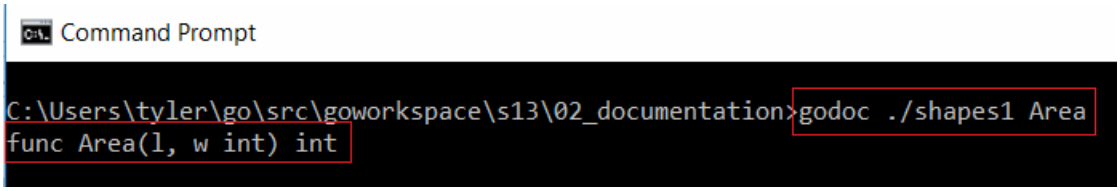
- Make sure the following is in place:
  GO_HOME=C:\Go
  GOPATH=C:\Users\tyler\go
  Add the following two to the "**path**" environment variable:
  %GO_HOME%\bin\
  %GOPATH%\src\; %GOPATH%\src\goworkspace\;

  > **go help gopath** *(In case you need information about gopath)*

- > cd C:\Users\tyler\go\src\goworkspace\s13\02_documentation
  Assuming that this folder contains your code.



- 1) Make sure that the Area() function has no comments
  > **godoc ./shapes1 Area**



- C:\Users\tyler\go\src\goworkspace\s13\02_documentation\shapes1**rectangle.go**

  package shapes1

  // Area calculates and returns area of a rectangle
  func **Area**(l, w int) int { return l * w }

  // Perim calculates and returns perimeter of a rectangle
  func **Perim**(l, w int) int { return 2 * (l + w) }

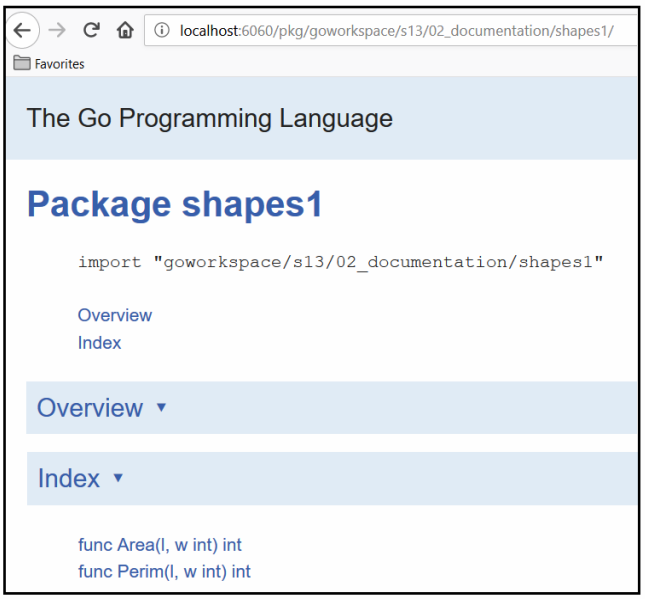
  > godoc ./shapes1 Area
  > godoc ./shapes1 Perim

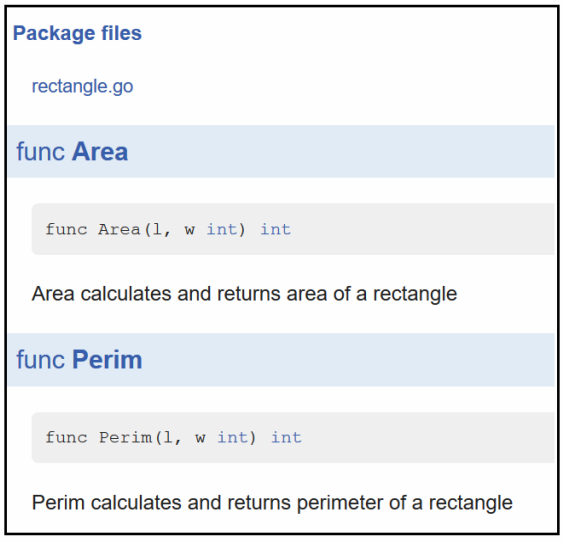
```
Command Prompt

C:\Users\tyler\go\src\goworkspace\s13\02_documentation>godoc ./shapes1 Area
func Area(l, w int) int
    Area calculates and returns area of a rectangle


C:\Users\tyler\go\src\goworkspace\s13\02_documentation>godoc ./shapes1 Perim
func Perim(l, w int) int
    Perim calculates and returns perimeter of a rectangle
```

- > cd C:\Users\tyler\go\src\goworkspace\s13\02_documentation
  > **go install**
  > **godoc -http=":6060"**

```
C:\Users\tyler\go\src\goworkspace\s13\02_documentation>go install
main.go:9:2: local import "./shapes1" in non-local package

C:\Users\tyler\go\src\goworkspace\s13\02_documentation>godoc -http=":6060"
2018/07/04 19:19:17 cannot find package "." in:
        \src\goworkspace\documentation\shapes1
```

In a browser:
http://localhost:6060/pkg/goworkspace/s13/02_documentation/shapes1/





- ***code: ...\s13\02_documentation\main.go***

SECTION 14 – ERROR HANDLING / UNIT TESTING

Error Handling

- **In Go, error values are like any other values**. They are not special. *(Rob Pike)*

- Go does not offer exception handling mechanism such as the try/catch in Java or other languages.
  Why does Go not have exceptions? https://golang.org/doc/faq#exceptions

- Go has a **defer-panic-and-recover** mechanism.
  Ref: https://blog.golang.org/defer-panic-and-recover

- To handle errors in Go, functions/methods can return an error object containing a nil value (indicating no error) or an error.
  The calling function is to check the error status.

- ****code: ...\s14\01_error_handling1\main.go***
  This program showcases three items (see the code for more details):

  - Package log implements a simple logging package. That logger writes to standard error and prints the date and time of each logged message.
    Example: ***log.Println****("Error: ", err) //err is of type* ***error***
    Control will return to the caller after this error.
    Review testCase1()
    ref: Package log: https://golang.org/pkg/log/

  - **func Exit(code int)**
    Ref: https://golang.org/pkg/os/#Exit
    Exit causes the current program to exit with the given status code. Conventionally, code zero indicates success, non-zero an error. The program terminates immediately; deferred functions are not run.

    Example: **log.Fatalln**(err) // prints the message and then calls os.Exit(1)
    Control will NOT return to the caller after this error.

  - **func panic(v interface{})**
    panic provide with **error stack info**. It's part of the package "builtin ""
    ref: https://golang.org/pkg/builtin/#panic
    Example: **panic**(err) // prints the message and then calls panic.
    Control will NOT return to the caller after this error.

- **How to signal errors?**

  ***code: ...\s14\02_error_handling2\main.go**

  func New(text string) error
  New returns an error that formats as the given text.
  Ref: https://golang.org/pkg/errors/

  Here's an example of "**errors.New**" copied from Go standard library:
  Ref: https://golang.org/src/archive/tar/common.go?h=errors.New

```
var (
  ErrHeader = errors.New("archive/tar: invalid tar header")
  ErrWriteTooLong = errors.New("archive/tar: write too long")
  ErrFieldTooLong = errors.New("archive/tar: header field too long")
  ErrWriteAfterClose = errors.New("archive/tar: write after close")
  errMissData = errors.New("archive/tar: sparse file references non-existent data")
  errUnrefData = errors.New("archive/tar: sparse file contains unreferenced data")
  errWriteHole       = errors.New("archive/tar: write non-NUL byte in sparse hole")
)
```

  func Errorf(format string, a ...interface{}) error
  Errorf formats according to a format specifier and returns the string as a value that satisfies error.
  Ref: https://golang.org/pkg/fmt/#Errorf

- **lint (software)**: A linter or lint refers to tools that analyze source code to flag programming errors, bugs, stylistic errors, and suspicious constructs. The term originates from a Unix utility that examined C language source code.
  Ref: https://en.wikipedia.org/wiki/Lint_(software)

- **golang lint**: golint is concerned with coding style and prints out **style mistakes**.
  Ref: https://github.com/golang/lint

  Installation: **go get -u -v golang.org/x/lint/golint**
  Installation usually copies "**golint.exe**" to "USER\go\bin". You may need to copy this file to your path.

- **Errors**: Library routines must often return some sort of error indication to the caller. As mentioned earlier, Go's multivalue return makes it easy to return a detailed error description alongside the normal return value. It is good style to use this feature to provide detailed error information. For example, as we'll see, os.Open doesn't just return a nil pointer on failure, it also returns an error value that describes what went wrong.
  Ref: https://golang.org/doc/effective_go.html?#errors

  By convention, errors have type error, a simple built-in interface.

  ```
  type error interface {
      Error() string
  }
  ```

  For custom-error-handling, you need to implement the 'error' interface.

  See the following example:
  code: ...\s14\03_error_handling_custom\main.go

Simple Unit Testing

- In software testing, test automation is the use of special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes with predicted outcomes. Test automation can automate some repetitive but necessary tasks in a formalized testing process already in place, or perform additional testing that would be difficult to do manually. Test automation is critical for continuous delivery and continuous testing.
  Ref: https://en.wikipedia.org/wiki/Test_automation

- The field of software testing is enormous and requires its own course.

- Go's lightweight approach to testing relying on one command, **go test** is effective for pure testing.
  Writing test code is fairly similar to writing the original program, utilizing short test cases/functions to ensure of program health.

- **Three types of functions for testing**:
  - A **test function**: whose name begins with Test; it's to test the correct program logic and behaviour; the final report by go test is either PASS or FAIL.
  - A **benchmark function**: whose name begins with Benchmark; measures the performance of some operations; the mean execution time of the operation will be reported; ***(out of scope of this course)***
  - An **example function**: whose name starts with Example; creates machine-checked documentation; ***(out of scope of this course)***

- **Package 'testing'** provides support for automated testing of Go packages. It is intended to be used in concert with the **"go test"** command.
  Ref: https://golang.org/pkg/testing/

  The "go test" command will run any function that starts with "Test", such as **TestRect(t *testing.T) {...}**, in any file in the current folder.

  The **t parameter** provides ways for reporting/logging test failures and additional information.

- ***code: ...\s14\04_testing\main.go***
  code: ...\s14\04_testing\myutil\utilprops_test.go

  How to run the test cases?

  > cd C:\KamProgramming\Go\GoByExample\code\s14\04_testing\myutil
  > **go test**

```
C:\KamProgramming\Go\GoByExample\code\s14\04_testing\myutil>go test
** TestRect - ALL PASSED (Number of test cases: 3) **
** TestAverage - ALL PASSED (Number of test cases: 4) **
PASS
ok      _/C_/KamProgramming/Go/GoByExample/code/s14/04_testing/myutil   3.288s
```

  -v to *report name and execution time of test cases.*
  > go test **-v**

```
C:\KamProgramming\Go\GoByExample\code\s14\04_testing\myutil>go test -v
=== RUN   TestRect
** TestRect - ALL PASSED (Number of test cases: 3) **
--- PASS: TestRect (0.00s)
=== RUN   TestAverage
** TestAverage - ALL PASSED (Number of test cases: 4) **
--- PASS: TestAverage (0.00s)
PASS
ok      _/C_/KamProgramming/Go/GoByExample/code/s14/04_testing/myutil   0.625s
```

  -run to run only those tests whose function name matches the pattern.
  > go test **-run** "TestRect"

```
C:\KamProgramming\Go\GoByExample\code\s14\04_testing\myutil>go test -run "TestRect"
** TestRect - ALL PASSED (Number of test cases: 3) **
PASS
ok      _/C_/KamProgramming/Go/GoByExample/code/s14/04_testing/myutil   0.628s
```

  > go test **-v -run** "TestRect"

```
C:\KamProgramming\Go\GoByExample\code\s14\04_testing\myutil>go test -v -run "TestRect"
=== RUN   TestRect
** TestRect - ALL PASSED (Number of test cases: 3) **
--- PASS: TestRect (0.00s)
PASS
ok      _/C_/KamProgramming/Go/GoByExample/code/s14/04_testing/myutil   0.636s
```

> go test -v -run "TestRect|TestAverage"

```
C:\KamProgramming\Go\GoByExample\code\s14\04_testing\myutil>go test -v -run "TestRect|TestAverage"
=== RUN   TestRect
** TestRect - ALL PASSED (Number of test cases: 3) **
--- PASS: TestRect (0.00s)
=== RUN   TestAverage
** TestAverage - ALL PASSED (Number of test cases: 4) **
--- PASS: TestAverage (0.00s)
PASS
ok      _/C_/KamProgramming/Go/GoByExample/code/s14/04_testing/myutil   0.676s
```

SECTION 15 – MISCELLANEOUS TOPICS

This section introduces several new topics in Go: ‘reference types’, ‘json’, ‘working with files’, ‘manipulating strings’, and ‘reflection’.
Detailed review of some of these topics would meaningful only in the context of other courses such as web development or networking.
Full and detailed explanation of those topics would be lengthy which at this time is out of scope of this course.
However, this section will provide students with a solid background of these topics so that further learning would be easy.

Reference Types - more examples

- So far we’ve seen several examples of call by reference / value but due to topics interdependencies, we had to leave the final discussion to this section.
- **Reference types**: pointers, slices, maps, functions, and channels.

- **Non-reference types**: the rest (int, float, bool, string, arrays, ...)

- ***code: ...\s15\01_call_by_ref\main.go***

Introduction to json

- **JavaScript Object Notation (JSON)** is a standard notation for sending and receiving structured information.

- Go offers support for encoding/decoding such formats (packages like encoding/json, encoding/xml)

- **Marshalling** is the process of transforming the memory representation of an object to a data format suitable for storage or transmission, and it is typically used when data must be moved between different parts of a computer program or from one program to another.

  Marshalling is similar to serialization and is used to communicate to remote objects with an object, in this case a serialized object. It simplifies complex communication (using composite objects, instead of primitives).

  The inverse, of marshalling is called unmarshalling (or demarshalling, similar to deserialization).

  Ref: https://en.wikipedia.org/wiki/Marshalling_(computer_science)

- **package json**:
  Ref: https://godoc.org/encoding/json

  type Decoder: A Decoder reads and decodes JSON values from an input stream.
  Ref: https://godoc.org/encoding/json#Decoder

  type Encoder: An Encoder writes JSON values to an output stream.
  Ref: https://godoc.org/encoding/json#Encoder

  func **Marshal**(v interface{}) ([]byte, error)
  Marshal returns the JSON encoding of v.
  Ref: https://godoc.org/encoding/json#Marshal

  func **Unmarshal**(data []byte, v interface{}) error
  Unmarshal parses the JSON-encoded data and stores the result in the value pointed to by v. If v is nil or not a pointer, Unmarshal returns an InvalidUnmarshalError.
  Ref: https://godoc.org/encoding/json#Unmarshal

- json.**Marshal()** / json.**MarshalIndent()**: a Go data structure → JSON object
  json.**Unmarshal()**: JSON object → a Go data structure

- ****code: ...\s15\02_json\main.go***

Introduction to Working with Files

- func **WriteFile**(filename string, data []byte, perm os.FileMode) error
  *(package **ioutil**)*
  writes data to a file named by filename. If the file does not exist, WriteFile creates it with permissions perm; otherwise WriteFile truncates it before writing.
  Ref: https://golang.org/pkg/io/ioutil/#WriteFile

- func **ReadFile**(filename string) ([]byte, error) *(package **ioutil**)*
  reads the file named by filename and returns the contents. A successful call returns err == nil, not err == EOF. Because ReadFile reads the whole file, it does not treat an EOF from Read as an error to be reported.
  Ref: https://golang.org/pkg/io/ioutil/#ReadFile

- func **Rename**(oldpath, newpath string) error *(package **os**)*
  renames (moves) oldpath to newpath. If newpath already exists and is not a directory, Rename replaces it. OS-specific restrictions may apply when oldpath and newpath are in different directories. If there is an error, it will be of type *LinkError.
  Ref: https://golang.org/pkg/os/#Rename

- func **Remove**(name string) error *(package **os**)*
  removes the named file or directory. If there is an error, it will be of type *PathError.
  Ref: https://golang.org/pkg/os/#Remove

- func **Open**(name string) (*File, error) *(package **os**)*
  opens the named file for reading. If successful, methods on the returned file can be used for reading; the associated file descriptor has mode O_RDONLY. If there is an error, it will be of type *PathError.
  Ref: https://golang.org/pkg/os/#Open

- func **Exit**(code int) *(package **os**)*
  causes the current program to exit with the given status code. Conventionally, code zero indicates success, non-zero an error. The program terminates immediately; deferred functions are not run.
  Ref: https://golang.org/pkg/os/#Exit

- func **Getwd()** (dir string, err error) *(package **os**)*
  returns a rooted path name corresponding to the current directory. If the current directory can be reached via multiple paths (due to symbolic links), Getwd may return any one of them.
  Ref: https://golang.org/pkg/os/#Getwd

- func **NewScanner**(r io.Reader) *Scanner *(package **bufio**)*
  returns a new Scanner to read from r. The split function defaults to ScanLines.
  Ref: https://golang.org/pkg/bufio/#NewScanner

- func (s *Scanner) **Scan()** bool *(package **bufio**)*
  advances the Scanner to the next token, which will then be available through the Bytes or Text method. It returns false when the scan stops, either by reaching the end of the input or an error. After Scan returns false, the Err method will return any error that occurred during scanning, except that if it was io.EOF, Err will return nil. Scan panics if the split function returns too many empty tokens without advancing the input. This is a common error mode for scanners.
  Ref: https://golang.org/pkg/bufio/#Scanner.Scan

- func (s *Scanner) **Text()** string *(package **bufio**)*
  returns the most recent token generated by a call to Scan as a newly allocated string holding its bytes.
  Ref: https://golang.org/pkg/bufio/#Scanner.Text

- ***code: ...\s15\03_io1\main.go**
- ***code: ...\s15\04_io2\main.go**

Manipulating Strings

- We are going to review the some of the commonly used functions in **bytes**, and **strings** packages.

- func **Compare**(a, b string) int *(package **strings**)*
  result: a==b -> 0; a < b -> -1 ; a > b -> +1;
  Ref: https://golang.org/pkg/strings/#Compare

- func **Compare**(a, b []byte) int *(package **bytes**)*
  returns an integer comparing two byte slices lexicographically. The result will be 0 if a==b, -1 if a < b, and +1 if a > b. A nil argument is equivalent to an empty slice.
  Ref: https://golang.org/pkg/bytes/#Compare

- func **Contains**(s, substr string) bool *(package **strings**)*
  reports whether substr is within s.
  Ref: https://golang.org/pkg/strings/#Contains

- func **Contains**(b, subslice []byte) bool *(package **bytes**)*
  Contains reports whether subslice is within b.
  Ref: https://golang.org/pkg/bytes/#Contains

- func **Count**(s, substr string) int *(package **strings**)*
  counts the number of non-overlapping instances of substr in s. If substr is an empty string, Count returns 1 + the number of Unicode code points in s.
  Ref: https://golang.org/pkg/strings/#Count

- func **Fields**(s string) []string *(package **strings**)*
  splits the string s around each instance of one or more consecutive white space characters, as defined by unicode.IsSpace, returning a slice of substrings of s or an empty slice if s contains only white space.
  Ref: https://golang.org/pkg/strings/#Fields

- func **Fields**(s []byte) [][]byte *(package **bytes**)*
  interprets s as a sequence of UTF-8-encoded code points. It splits the slice s around each instance of one or more consecutive white space characters, as defined by unicode.IsSpace, returning a slice of subslices of s or an empty slice if s contains only white space.
  Ref: https://golang.org/pkg/bytes/#Fields

- func **HasPrefix**(s, prefix string) bool *(package **strings**)*
  tests whether the string s begins with prefix.
  Ref: https://golang.org/pkg/strings/#HasPrefix

- func **HasSuffix**(s, suffix string) bool *(package **strings**)*
  tests whether the string s ends with suffix.
  Ref: https://golang.org/pkg/strings/#HasSuffix

- func **Index(**s, substr string) int *(package **strings**)*
  returns the index of the first instance of substr in s, or -1 if substr is not present in s.
  Ref: https://golang.org/pkg/strings/#Index

- func **Join**(a []string, sep string) string *(package **strings**)*
  concatenates the elements of a to create a single string. The separator string sep is placed between elements in the resulting string.
  Ref: https://golang.org/pkg/strings/#Join

- func **LastIndex**(s, substr string) int *(package **strings**)*
  returns the index of the last instance of substr in s, or -1 if substr is not present in s.
  Ref: https://golang.org/pkg/strings/#LastIndex

- func **Map**(mapping func(rune) rune, s string) string *(package **strings**)*
  returns a copy of the string s with all its characters modified according to the mapping function. If mapping returns a negative value, the character is dropped from the string with no replacement.
  Ref: https://golang.org/pkg/strings/#Map

- func **Repeat**(s string, count int) string *(package **strings**)*
  returns a new string consisting of count copies of the string s. It panics if count is negative or if the result of (len(s) * count) overflows.
  Ref: https://golang.org/pkg/strings/#Repeat

- func **Replace**(s, old, new string, n int) string *(package **strings**)*
  returns a copy of the string s with the first n non-overlapping instances of old replaced by new. If old is empty, it matches at the beginning of the string and after each UTF-8 sequence, yielding up to k+1 replacements for a k-rune string. If n < 0, there is no limit on the number of replacements.
  Ref: https://golang.org/pkg/strings/#Replace

- func Split(s, sep string) []string *(package **strings**)*
  slices s into all substrings separated by sep and returns a slice of the substrings between those separators.
  If s does not contain sep and sep is not empty, Split returns a slice of length 1 whose only element is s.
  If sep is empty, Split splits after each UTF-8 sequence. If both s and sep are empty, Split returns an empty slice.
  It is equivalent to SplitN with a count of -1.
  Ref: https://golang.org/pkg/strings/#Split

- func **ToLower**(s string) string *(package **strings**)*
  returns a copy of the string s with all Unicode letters mapped to their lower case.
  Ref: https://golang.org/pkg/strings/#ToLower

- func **ToUpper**(s string) string *(package **strings**)*
  returns a copy of the string s with all Unicode letters mapped to their upper case.
  Ref: https://golang.org/pkg/strings/#ToUpper

- func **Trim**(s string, cutset string) string *(package **strings**)*
  returns a slice of the string s with all leading and trailing Unicode code points contained in cutset removed.
  Ref: https://golang.org/pkg/strings/#Trim

- func **TrimLeft**(s string, cutset string) string *(package **strings**)*
  returns a slice of the string s with all leading Unicode code points contained in cutset removed.
  Ref: https://golang.org/pkg/strings/#TrimLeft

- func **TrimRight**(s string, cutset string) string *(package **strings**)*
  returns a slice of the string s, with all trailing Unicode code points contained in cutset removed.
  Ref: https://golang.org/pkg/strings/#TrimRight

- *****code: ...\s15\05_string_manipulation\main.go***

Introduction to Reflection

- ***code: ...\s15\06_reflection1\main.go***

- **Reflection** is the ability of a program to study its own structure, particularly through the types.

- It can be used to dynamically examine types and variables at runtime.

- Although it's a powerful tool but it should be used with care and only when it's absolutely necessary.

- We can use reflection when we need to deal with values of types that don't satisfy a common interface, or they're absent at design time.
  fmt.Fprintf is a good example that prints different types and accepts a variable number of parameters.

- Package **reflect**: https://golang.org/pkg/reflect/
  Package reflect implements run-time reflection, allowing a program to manipulate objects with arbitrary types. The typical use is to take a value with static type interface{} and extract its dynamic type information by calling TypeOf, which returns a Type.

- Basic information of a variable (reflect.TypeOf, and reflect. ValueOf):
  - func **TypeOf**(i interface{}) **Type**
    Ref (TypeOf): https://golang.org/pkg/reflect/#TypeOf
    accepts any interface{} and returns its dynamic type as a reflect.Type

    Ref (type Type): https://golang.org/pkg/reflect/#Type

  - func **ValueOf**(i interface{}) **Value**
    Ref (ValueOf): https://golang.org/pkg/reflect/#ValueOf
    accepts any interface{} and returns a reflect.Value containing the interface's dynamic value.
    A reflect.Value can hold a value of any type.
    Ref (type Value): https://golang.org/pkg/reflect/#Value

  - func (v Value) **Interface()** (i interface{})
    returns v's current value as an interface{}.
    It is equivalent to: var i interface{} = (v's underlying value)
    Ref: https://golang.org/pkg/reflect/#Value.Interface

- func (v Value) **Float()** float64
  returns v's underlying value, as a float64. It panics if v's Kind is not Float32 or Float64
  Ref: https://golang.org/pkg/reflect/#Value.Float

- func (v Value) **String()** string
  returns the string v's underlying value, as a string. String is a special case because of Go's String method convention. Unlike the other getters, it does not panic if v's Kind is not String. Instead, it returns a string of the form "<T value>" where T is v's type. The fmt package treats Values specially. It does not call their String method implicitly but instead prints the concrete values they hold.
  Ref: https://golang.org/pkg/reflect/#Value.String

- type **Kind**: https://golang.org/pkg/reflect/#Kind
  A Kind represents the specific kind of type that a Type represents. The zero Kind is not a valid kind.

```
const (
        Invalid Kind = iota
        Bool
        Int, Int8, Int16, Int32, Int64
        Uint, Uint8, Uint16, Uint32, Uint64
        Uintptr
        Float32, Float64
        Complex64, Complex128
        Array
        Chan
        Func
        Interface
        Map
        Ptr
        Slice
        String
        Struct
        UnsafePointer
)
```

  func (k Kind) **String()** string

  For our variable x if v:= reflect.ValueOf(x) then v.Kind() is float64, so the following is true: v.Kind() == reflect.Float64

- ***code: ...\s15\07_reflection2\main.go***
- ***code: ...\s15\08_reflection3\main.go***

SECTION 16 – MySQL / PostgreSQL / A Tour of SQL & Final Project

PostgreSQL & MySQL Installation

https://www.postgresql.org/
https://dev.mysql.com/downloads/

A Tour of SQL (MySQL & PostgreSQL)

Relational model:
Ref: https://en.wikipedia.org/wiki/Relational_database

- The relational model organizes data into one or more tables.
- A **table** consists of columns and rows, normally with a unique key identifying each row;
- Generally, each table represents one "**entity type**" (such as customer or product);
- A **row (record)** represents an instance of that type of entity;
- A **column (filed** or **attribute)** represents value attributed to that instance.

EMPLOYEE TABLE

field 1 (column 1)	*field 2*	*field 3*	*field 4*	*field 5*	*field 6*	
customer_id	**first_name**	**last_name**	**email**	**gender**	**employer_id**	
1	Jessica	Young	j@young.org	F	3	*row 1*
2	John	Jones	j@jones.org	M	2	*row 2*
3	Ray	Davis	r@davis.org	M	1	*row 3*
4	Sandra	Walker	s@walker.org	F	2	*row 4*
5	Tina	Simmons	t@simmons.org	F	3	*row 5*
6	Don	Jackson	d@jackson.org	M	1	*row 6*

EMPLOYER TABLE

employer_id	company_name
1	Amazon
2	Google
3	Microsoft

SQL Statements

Create a database:
CREATE DATABASE go_db1 *(PostgreSQL)*
CREATE SCHEMA go_db1; *(MySQL)* *right-click / Set As Default Schema*

Make sure the database is the **default DB**.

* *

```
CREATE TABLE employee (
	customer_id SERIAL PRIMARY KEY,
	first_name VARCHAR(50),
	last_name VARCHAR(50),
	email VARCHAR(50),
	gender VARCHAR(1),
	employer_id int
);

INSERT INTO employee (customer_id, first_name, last_name, email, gender,
employer_id)
VALUES (1, 'Jessica', 'Young', 'j@young.org', 'F', 3);

INSERT INTO employee VALUES (2, 'John', 'Jones', 'j@jones.org', 'M', 2);
INSERT INTO employee VALUES (3, 'Ray', 'Davis', 'r@davis.org', 'M', 1);
INSERT INTO employee VALUES (4, 'Sandra', 'Walker', 's@walker.org', 'F', 2);
INSERT INTO employee VALUES (5, 'Tina', 'Simmons', 't@simmons.org', 'F', 3);
INSERT INTO employee VALUES (6, 'Don', 'Jackson', 'd@jackson.org', 'M', 1);

SELECT * FROM employee;
```

* *

```
CREATE TABLE employer (
	employer_id SERIAL PRIMARY KEY,
	company_name VARCHAR(50)
);

INSERT INTO employer (employer_id, company_name) VALUES (1, 'Amazon');
INSERT INTO employer VALUES (2, 'Google');
INSERT INTO employer VALUES (3, 'Microsoft');

SELECT * FROM employer;
```

* *

```
SELECT first_name, gender FROM employee;

SELECT * FROM employee WHERE gender = 'F';
SELECT * FROM employee WHERE gender != 'F';

SELECT * FROM employee
WHERE first_name='John' OR last_name='Jackson';
Note: Double quote only works in MySQL.

SELECT * FROM employee WHERE employer_id <= 2;

SELECT * FROM employee
WHERE customer_id NOT BETWEEN 2 AND 4;

SELECT * FROM employee WHERE first_name LIKE 'J%';

SELECT * FROM employee WHERE last_name LIKE '%n%';

SELECT * FROM employee WHERE email LIKE '_@j%';

SELECT employer_id FROM employee;
SELECT DISTINCT employer_id FROM employee;

SELECT COUNT(*)  FROM employee;
SELECT COUNT(employer_id)  FROM employee;
SELECT COUNT(DISTINCT employer_id)  FROM employee;

SELECT * FROM employee
ORDER BY gender DESC, employer_id ASC;

SELECT * FROM employee
ORDER BY gender DESC, employer_id ASC LIMIT 4;
```

* *

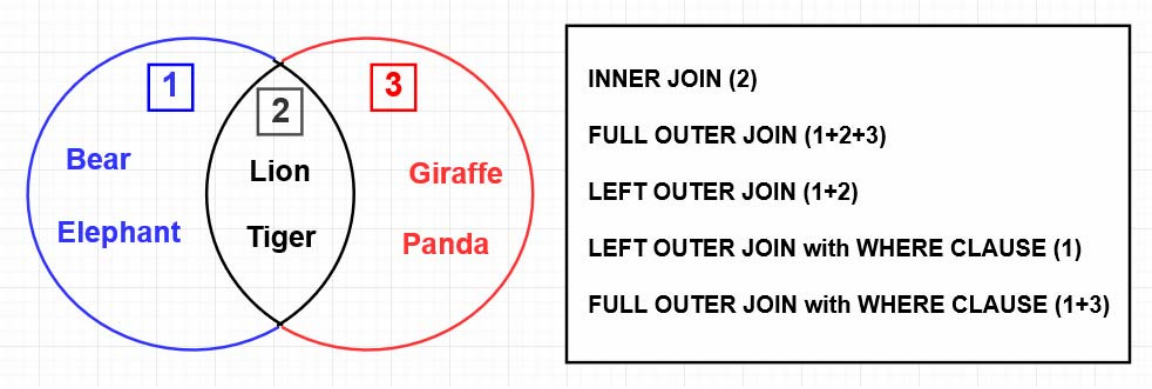
```
SELECT * FROM employee e, employer r
WHERE e.employer_id = r.employer_id;

SELECT e.first_name, e.employer_id, r.company_name
FROM employee e, employer r
WHERE e.employer_id = r.employer_id;
```

* *

```
UPDATE employee SET first_name='MIKE' WHERE customer_id=6;

DELETE FROM employee WHERE customer_id=6;
```

JOIN

```
CREATE TABLE t1 (
	id serial PRIMARY KEY,
	animal VARCHAR (20)
);

INSERT INTO t1 (animal)
VALUES ('Lion'), ('Bear'), ('Tiger'), ('Elephant');


CREATE TABLE t2 (
	id serial PRIMARY KEY,
	animal VARCHAR (20)
);

INSERT INTO t2 (animal)
VALUES ('Lion'), ('Giraffe'), ('Panda'), ('Tiger');
```



*** INNER JOIN** (2)
```
SELECT * FROM t1
INNER JOIN t2
ON t1.animal = t2.animal;
```

*** FULL OUTER JOIN** (1+2+3) (only works in Postgres)
```
SELECT * FROM t1
FULL OUTER JOIN t2
ON t1.animal = t2.animal;
```

*** FULL OUTER JOIN** (1+2+3) (works in both MySQL & Postgres)
```
SELECT * FROM t1
LEFT JOIN t2 ON t1.animal = t2.animal
UNION ALL
SELECT * FROM t1
RIGHT JOIN t2 ON t1.animal = t2.animal
WHERE t1.id IS NULL;
```

*** LEFT OUTER JOIN** (1+2)
```
SELECT * FROM t1
LEFT OUTER JOIN t2
ON t1.animal = t2.animal;
```

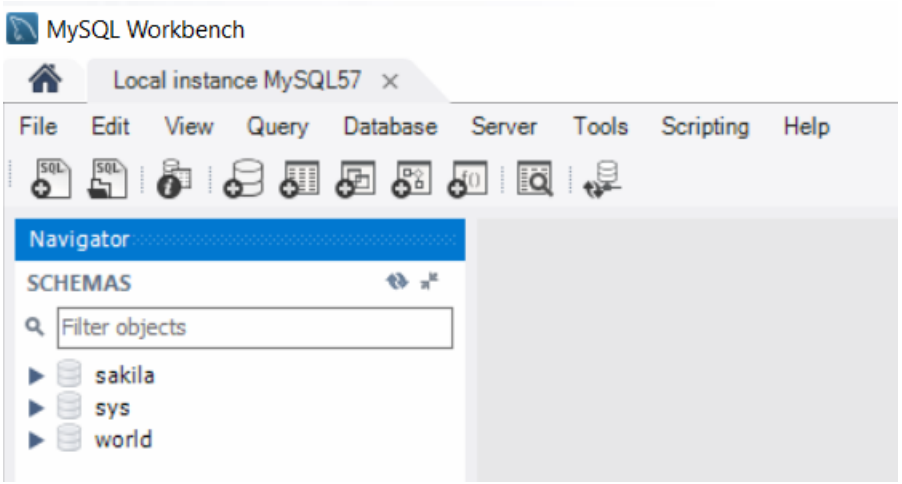
*** LEFT OUTER JOIN with WHERE CLAUSE** (1)
```
SELECT * FROM t1
LEFT OUTER JOIN t2
ON t1.animal = t2.animal
WHERE t2.id IS NULL;
```

*** FULL OUTER JOIN with WHERE CLAUSE** (1+3) (only works in Postgres)
```
SELECT * FROM t1
FULL OUTER JOIN t2
ON t1.animal = t2.animal
WHERE t1.id IS NULL OR
t2.id IS NULL;
```

*** FULL OUTER JOIN with WHERE CLAUSE** (1+3) (works in both**)**
```
SELECT * FROM t1
LEFT OUTER JOIN t2
ON t1.animal = t2.animal
WHERE t2.id IS NULL or t1.id IS NULL
UNION
SELECT * FROM t1
RIGHT OUTER JOIN t2
ON t1.animal = t2.animal
WHERE t2.id IS NULL or t1.id IS NULL;
```

Using Go to connect to MySQL

- Start the MySQL Workbench → If this is your first attempt after installation, you will see the Workbench with the following databases.



- In MySQL Workbench:
  CREATE SCHEMA `**go_db1**`;
  right-click / Set As Default Schema

- **DROP SCHEMA** `go_db1` ; *(if needed)*
  Will remove the `go_db1` database and all its contents (tables, …)

- Create a table
  CREATE TABLE person (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255)
  );

- Insert several rows to the table:

  INSERT INTO `go_db1`.`person` (`id`, `name`) VALUES ('1', 'Ava');
  INSERT INTO `go_db1`.`person` (`id`, `name`) VALUES ('2', 'John');
  INSERT INTO `go_db1`.`person` (`id`, `name`) VALUES ('3', 'June');
  INSERT INTO `go_db1`.`person` (`id`, `name`) VALUES ('4', 'Tim');

- **SELECT** * FROM `go_db1`.`person`;
- **UPDATE** `go_db1`.`person` SET `name`='Jack' WHERE `id`='2';
- **DELETE** FROM `go_db1`.`person` WHERE `id`='2';

- *Ref: https://golang.org/pkg/database/sql/#Open*
  func **Open**(driverName, dataSourceName string) (*DB, error)

  opens a database specified by its database driver name and a driver-specific data source name, usually consisting of at least a database name and connection information.

- *Ref: https://golang.org/pkg/database/sql/#DB.Close*
  func (db *DB) **Close()** error

  closes the database, releasing any open resources.

- *Ref: https://golang.org/pkg/database/sql/#DB*
  type **DB**

  DB is a database handle representing a pool of zero or more underlying connections. It's safe for concurrent use by multiple goroutines.

  The sql package creates and frees connections automatically; it also maintains a free pool of idle connections. If the database has a concept of per-connection state, such state can only be reliably observed within a transaction. Once DB.Begin is called, the returned Tx is bound to a single connection. Once Commit or Rollback is called on the transaction, that transaction's connection is returned to DB's idle connection pool. The pool size can be controlled with SetMaxIdleConns.

  ```
  type DB struct {
    connector driver.Connector
    numClosed uint64
    mu           sync.Mutex
    freeConn     []*driverConn
    connRequests map[uint64]chan connRequest
    ...
  }
  ```

- *Ref: https://golang.org/pkg/database/sql/#DB.Query*
  func (db *DB) **Query**(query string, args ...interface{}) (*Rows, error)

  Query executes a query that returns rows, typically a SELECT. The args are for any placeholder parameters in the query.

- *Ref: https://golang.org/pkg/database/sql/#Rows.Next*
  func (rs *Rows) **Next()** bool

Next prepares the next result row for reading with the Scan method. It returns true on success, or false if there is no next result row or an error happened while preparing it. Err should be consulted to distinguish between the two cases.

Every call to Scan, even the first one, must be preceded by a call to Next.

- *Ref: https://golang.org/pkg/database/sql/#DB.QueryRow*
  func (db *DB) **QueryRow**(query string, args ...interface{}) *Row

  QueryRow executes a query that is expected to return at most one row. QueryRow always returns a non-nil value. Errors are deferred until Row's Scan method is called. If the query selects no rows, the *Row's Scan will return ErrNoRows. Otherwise, the *Row's Scan scans the first selected row and discards the rest.

- *Ref: https://golang.org/pkg/database/sql/#DB.Prepare*
  func (db *DB) **Prepare**(query string) (*Stmt, error)

  Prepare creates a prepared statement for later queries or executions. Multiple queries or executions may be run concurrently from the returned statement. The caller must call the statement's Close method when the statement is no longer needed.

- *Ref: https://golang.org/pkg/database/sql/#Stmt.Exec*
  func (s *Stmt) **Exec**(args ...interface{}) (Result, error)

  Exec executes a prepared statement with the given arguments and returns a Result summarizing the effect of the statement.

- *Ref: https://golang.org/pkg/database/sql/driver/#RowsAffected.LastInsertId*
  func (RowsAffected) **LastInsertId()** (int64, error)

- *Ref: https://golang.org/pkg/database/sql/driver/#RowsAffected*
  type **RowsAffected** int64
  RowsAffected implements Result for an INSERT or UPDATE operation which mutates a number of rows.

- *Ref: https://golang.org/pkg/database/sql/driver/#RowsAffected.RowsAffected*
  func (v RowsAffected) **RowsAffected()** (int64, error)

Project – Simplified Mortgage Application

D

id	first_name	last_name	credit score	salary	actual house downpayment	house_ id
1	Nick	White	670	$253,000	$320,000	3
2	Tom	Jones	770	$264,000	$800,000	1
3	Nicole	Freedman	750	$203,000	$850,000	2
4	June	Green	700	$148,000	$600,000	4

***CUSTOMER** TABLE*

P (2) (3)

id	house price	minimum required downpayment	property tax (yearly)	maintenance cost (yearly)
1	$2,000,000	$400,000	$9,200	$1,800
2	$1,750,000	$350,000	$8,050	$1,800
3	$1,500,000	$300,000	$6,900	$1,800
4	$1,250,000	$250,000	$5,750	$1,800

***HOUSE** TABLE*

- Nick wants to purchase house #3
- **(C1)** is his credit_score >= 650 ? Yes; it's 670 *(otherwise, application rejected)*
- **(C2)** is his actual house downpayment >= minimum required downpayment ?
  Yes; 320,000 >= 300,000 *(otherwise, application rejected)*

- P-D = 1,500,000 - 320,000 = 1,180,000 -> x **0.0612** = **72,216** (mortgage, yearly) **(1)**
- (1) + (2) + (3) = 72,216 + 6,900 + 1,800 = 80,916 (yearly expense) **(4)**
- salary * %32 = 253000 * %32 = **80,960** (32% of income) **(5)**
- **(C3)** if (4) <= (5) then application approved

Program Requirements:

- Create the required tables and data structures.

- Your program will have the following functionalities:
  - Matching customer records with house records and identify if application is approved or rejected. Start with functions and then use goroutines. Also test your program with 1,2,4, and 8 CPUs.

```
4  June       Green       700    148000.00    600000.00   4
4   1250000.00    250000.00      5750.00      1800.00
**Mortgage Application Approved! [process time=841ns]

7  Nancy      Freedman    750    200000.00    850000.00   2
2   1750000.00    350000.00      8050.00      1800.00
**Mortgage Application Rejected! [process time=525ns]
```

  - Log the approved and rejected applications to a channel and read them to console. The type of the channel is better to be of type of a struct (and not a primitive type).

```
[7 NANCY FREEDMAN 200000 850000 **REJECTED**]
[1 NICK WHITE 253000 320000 APPROVED]
```

  - In addition to logging the approved and rejected applications to a channel and console, send the summary to a file with the following format.

```
7,NANCY,FREEDMAN,200000,850000,**REJECTED**
1,NICK,WHITE,253000,320000,APPROVED
```

- Make sure to apply the techniques we've learned so far to your program. In particular use packages, goroutines, channels, structs, slices, files, …

- **CREATE TABLE house** (
  id SERIAL PRIMARY KEY,
  price FLOAT,
  min_down FLOAT,
  prop_tax FLOAT,
  m_cost FLOAT
  );

- Insert several rows to the table:
  INSERT INTO `go_db1`.`house` (`id`, `price`, `min_down`, `prop_tax`, `m_cost`) VALUES ('1', '2000000', '400000', '9200', '1800');
  INSERT INTO `go_db1`.`house` (`id`, `price`, `min_down`, `prop_tax`, `m_cost`) VALUES ('2', '1750000', '350000', '8050', '1800');
  INSERT INTO `go_db1`.`house` (`id`, `price`, `min_down`, `prop_tax`, `m_cost`) VALUES ('3', '1500000', '300000', '6900', '1800');
  INSERT INTO `go_db1`.`house` (`id`, `price`, `min_down`, `prop_tax`, `m_cost`) VALUES ('4', '1250000', '250000', '5750', '1800');

- **CREATE TABLE customer** (
  id SERIAL PRIMARY KEY,
  first_name VARCHAR(255),
  last_name VARCHAR(255),
  credit_score INT,
  salary FLOAT,
  downpayment FLOAT,
  house_id INT
  );

- Insert several rows to the table:
  INSERT INTO `go_db1`.`customer` (`id`, `first_name`, `last_name`, `credit_score`, `salary`, `downpayment`, `house_id`) VALUES ('1', 'Nick', 'White', '670', '253000', '320000', '3');
  INSERT INTO `go_db1`.`customer` (`id`, `first_name`, `last_name`, `credit_score`, `salary`, `downpayment`, `house_id`) VALUES ('2', 'Tom', 'Jones', '770', '264000', '800000', '1');
  INSERT INTO `go_db1`.`customer` (`id`, `first_name`, `last_name`, `credit_score`, `salary`, `downpayment`, `house_id`) VALUES ('3', 'Nicole', 'Freedman', '750', '203000', '850000', '2');
  INSERT INTO `go_db1`.`customer` (`id`, `first_name`, `last_name`, `credit_score`, `salary`, `downpayment`, `house_id`) VALUES ('4', 'June', 'Green', '700', '148000', '600000', '4');