12/01/2022 Go Lesson Instruction

132. What are pointers.  
This rule arises because pointer methods can modify the receiver; invoking them on a value would cause the method to receive a copy of the value, so any modifications would be discarded. The language therefore disallows this mistake. There is a handy exception, though. When the value is addressable, the language takes care of the common case of invoking a pointer method on a value by inserting the address operator automatically. In our example, the variable b is addressable, so we can call its Write method with just b.Write. The compiler will rewrite that to (&b).Write for us.  
fmt.Println(a)  
fmt.Println(&a)  
  
b := &a // & gives you the address  
fmt.Println(\*b) // \* gives you the value stored an address  
fmt.Println(\*&a) // \*& gives you value of addres  
  
133. When to use pointers in Golang  
Pointers are used for efficiency because everything in Go is passed by value so they let us pass an address where data is held instead of passing the data's value, to avoid unintentionally changing data, and so we can access an actual value in another function and not just a copy of it when we want to mutate it  
func main() {  
x := 34  
foo(&x) //sending addres of data  
fmt.Println(x)  
fmt.Println(&x)  
}  
  
func foo(y \*int) { //\*int is type of addres because y is address  
fmt.Println(y)  
fmt.Println(\*y)  
\*y = 43 //set value of address  
fmt.Println(y)  
fmt.Println(\*y)  
}  
  
134. JSON documentation  
<https://pkg.go.dev/encoding/json>  
  
135. JSON marshal  
func Marshal(v any) ([]byte, error)  
  
Marshal returns the JSON encoding of v. Marshal traverses the value v recursively. If an encountered value implements the Marshaler interface and is not a nil pointer, Marshal calls its MarshalJSON method to produce JSON. If no MarshalJSON method is present but the value implements encoding.TextMarshaler instead, Marshal calls its MarshalText method and encodes the result as a JSON string. The nil pointer exception is not strictly necessary but mimics a similar, necessary exception in the behavior of UnmarshalJSON.  
  
people := []person{p1, p2} // people is is slice, person is slice  
fmt.Println(people) // return only people slice  
bs, err := json.Marshal(people) // bs is json, Marshal is stored people to json  
if err != nil { // This is error catch  
fmt.Println(err) // Error message  
}  
fmt.Println(string(bs)) //This is Output of json decode.  
  
136. JSON unmarshal  
func Unmarshal(data []byte, v any) 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.  
Unmarshal uses the inverse of the encodings that Marshal uses, allocating maps, slices, and pointers as necessary, with the following additional rules:  
  
To unmarshal JSON into a pointer, Unmarshal first handles the case of the JSON being the JSON literal null. In that case, Unmarshal sets the pointer to nil. Otherwise, Unmarshal unmarshals the JSON into the value pointed at by the pointer. If the pointer is nil, Unmarshal allocates a new value for it to point to.  
  
To unmarshal JSON into a value implementing the Unmarshaler interface, Unmarshal calls that value's UnmarshalJSON method, including when the input is a JSON null. Otherwise, if the value implements encoding.TextUnmarshaler and the input is a JSON quoted string, Unmarshal calls that value's UnmarshalText method with the unquoted form of the string.  
  
To unmarshal JSON into a struct, Unmarshal matches incoming object keys to the keys used by Marshal (either the struct field name or its tag), preferring an exact match but also accepting a case-insensitive match. By default, object keys which don't have a corresponding struct field are ignored (see Decoder.DisallowUnknownFields for an alternative).  
  
s := `[  
{"First":"Javokhir","Last":"Nematov","Age":20},  
{"First":"Ravi","Last":"Seyed-Mahmoud","Age":35}  
]`  
bs := []byte(s)  
var people []person  
err := json.Unmarshal(bs, &people)  
if err != nil {  
fmt.Println(err)  
}  
fmt.Println(people)  
  
137. Writer Interface  
  
Formatted printing in Go uses a style similar to C's printf family but is richer and more general. The functions live in the fmt package and have capitalized names: `fmt.Printf`, `fmt.Fprintf`, `fmt.Sprintf` and so on. The string functions (Sprintf etc.) return a string rather than filling in a provided buffer.  
  
fmt.Printf("Hello %d\n", 23)  
fmt.Fprint(os.Stdout, "Hello ", 23, "\n")  
fmt.Println("Hello", 23)  
fmt.Println(fmt.Sprint("Hello ", 23))  
  
138. Sort  
fmt.Println(ab)  
sort.Ints(ab) //Sort for Integer  
fmt.Println(ab)  
  
fmt.Println(bc)  
sort.Strings(bc) //Sort for String  
fmt.Println(bc)  
  
139. bcrypt  
Cost returns the hashing cost used to create the given hashed password. When, in the future, the hashing cost of a password system needs to be increased in order to adjust for greater computational power, this function allows one to establish which passwords need to be updated.  
  
const (  
MinCost int = 4 // the minimum allowable cost as passed in to   
GenerateFromPassword  
  
MaxCost int = 31 // the maximum allowable cost as passed in to GenerateFromPassword  
  
DefaultCost int = 10 // the cost that will actually be set if a cost below MinCost is passed into GenerateFromPassword  
)