Methods

CR1

// Sample program to show how to declare methods and how the Go

// compiler supports them.

package main

import (

"fmt"

)

// user defines a user in the program.

type user struct {

name string

email string

}

// notify implements a method with a value receiver.

func (u user) notify() {

fmt.Printf("Sending User Email To %s<%s>\n",

u.name,

u.email)

}

// changeEmail implements a method with a pointer receiver.

func (u \*user) changeEmail(email string) {

u.email = email

}

func main() {

// Values of type user can be used to call methods

// declared with both value and pointer receivers.

bill := user{"Bill", "bill@email.com"}

bill.changeEmail("bill@hotmail.com")

bill.notify()

// Pointers of type user can also be used to call methods

// declared with both value and pointer receiver.

joan := &user{"Joan", "joan@email.com"}

joan.changeEmail("joan@hotmail.com")

joan.notify()

// Create a slice of user values with two users.

users := []user{

{"ed", "ed@email.com"},

{"erick", "erick@email.com"},

}

// Iterate over the slice of users switching

// semantics. Not Good!

for \_, u := range users {

u.changeEmail("it@wontmatter.com")

}

// Exception example: Using pointer semantics

// for a collection of strings.

keys := make([]string, 10)

for i := range keys {

keys[i] = func() string { return "key-gen" }()

}

}

CR2

package main

import (

"sync/atomic"

"syscall"

)

// Sample code to show how it is important to use value or pointer semantics

// in a consistent way. Choose the semantic that is reasonable and practical

// for the given type and be consistent. One exception is an unmarshal

// operation since that always requires the address of a value.

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// These is a named type from the net package called IP and IPMask with a base

// type that is a slice of bytes. Since we use value semantics for reference

// types, the implementation is using value semantics for both.

type IP []byte

type IPMask []byte

// Mask is using a value receiver and returning a value of type IP. This

// method is using value semantics for type IP.

func (ip IP) Mask(mask IPMask) IP {

if len(mask) == IPv6len && len(ip) == IPv4len && allFF(mask[:12]) {

mask = mask[12:]

}

if len(mask) == IPv4len && len(ip) == IPv6len && bytesEqual(ip[:12], v4InV6Prefix) {

ip = ip[12:]

}

n := len(ip)

if n != len(mask) {

return nil

}

out := make(IP, n)

for i := 0; i < n; i++ {

out[i] = ip[i] & mask[i]

}

return out

}

// ipEmptyString accepts a value of type IP and returns a value of type string.

// The function is using value semantics for type IP.

func ipEmptyString(ip IP) string {

if len(ip) == 0 {

return ""

}

return ip.String()

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Should time use value or pointer semantics? If you need to modify a time

// value should you mutate the value or create a new one?

type Time struct {

sec int64

nsec int32

loc \*Location

}

// Factory functions dictate the semantics that will be used. The Now function

// returns a value of type Time. This means we should be using value

// semantics and copy Time values.

func Now() Time {

sec, nsec := now()

return Time{sec + unixToInternal, nsec, Local}

}

// Add is using a value receiver and returning a value of type Time. This

// method is using value semantics for Time.

func (t Time) Add(d Duration) Time {

t.sec += int64(d / 1e9)

nsec := int32(t.nsec) + int32(d%1e9)

if nsec >= 1e9 {

t.sec++

nsec -= 1e9

} else if nsec < 0 {

t.sec--

nsec += 1e9

}

t.nsec = nsec

return t

}

// div accepts a value of type Time and returns values of built-in types.

// The function is using value semantics for type Time.

func div(t Time, d Duration) (qmod2 int, r Duration) {

// Code here

}

// The only use pointer semantics for the `Time` api are these

// unmarshal related functions.

func (t \*Time) UnmarshalBinary(data []byte) error {

func (t \*Time) GobDecode(data []byte) error {

func (t \*Time) UnmarshalJSON(data []byte) error {

func (t \*Time) UnmarshalText(data []byte) error {

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Factory functions dictate the semantics that will be used. The Open function

// returns a pointer of type File. This means we should be using pointer

// semantics and share File values.

func Open(name string) (file \*File, err error) {

return OpenFile(name, O\_RDONLY, 0)

}

// Chdir is using a pointer receiver. This method is using pointer

// semantics for File.

func (f \*File) Chdir() error {

if f == nil {

return ErrInvalid

}

if e := syscall.Fchdir(f.fd); e != nil {

return &PathError{"chdir", f.name, e}

}

return nil

}

// epipecheck accepts a pointer of type File.

// The function is using pointer semantics for type File.

func epipecheck(file \*File, e error) {

if e == syscall.EPIPE {

if atomic.AddInt32(&file.nepipe, 1) >= 10 {

sigpipe()

}

} else {

atomic.StoreInt32(&file.nepipe, 0)

}

}

CR3

// Sample program to show how to declare methods against

// a named type.

package main

import "fmt"

// duration is a named type that represents a duration

// of time in Nanosecond.

type duration int64

const (

nanosecond duration = 1

microsecond = 1000 \* nanosecond

millisecond = 1000 \* microsecond

second = 1000 \* millisecond

minute = 60 \* second

hour = 60 \* minute

)

// setHours sets the specified number of hours.

func (d \*duration) setHours(h float64) {

\*d = duration(h) \* hour

}

// hours returns the duration as a floating point number of hours.

func (d duration) hours() float64 {

hour := d / hour

nsec := d % hour

return float64(hour) + float64(nsec)\*(1e-9/60/60)

}

func main() {

// Declare a variable of type duration set to

// its zero value.

var dur duration

// Change the value of dur to equal

// five hours.

dur.setHours(5)

// Display the new value of dur.

fmt.Println("Hours:", dur.hours())

}

CR4

// Sample program to show how to declare function variables.

package main

import "fmt"

// data is a struct to bind methods to.

type data struct {

name string

age int

}

// displayName provides a pretty print view of the name.

func (d data) displayName() {

fmt.Println("My Name Is", d.name)

}

// setAge sets the age and displays the value.

func (d \*data) setAge(age int) {

d.age = age

fmt.Println(d.name, "Is Age", d.age)

}

func main() {

// Declare a variable of type data.

d := data{

name: "Bill",

}

fmt.Println("Proper Calls to Methods:")

// How we actually call methods in Go.

d.displayName()

d.setAge(45)

fmt.Println("\nWhat the Compiler is Doing:")

// This is what Go is doing underneath.

data.displayName(d)

(\*data).setAge(&d, 45)

// =========================================================================

fmt.Println("\nCall Value Receiver Methods with Variable:")

// Declare a function variable for the method bound to the d variable.

// The function variable will get its own copy of d because the method

// is using a value receiver.

f1 := d.displayName

// Call the method via the variable.

f1()

// Change the value of d.

d.name = "Joan"

// Call the method via the variable. We don't see the change.

f1()

// =========================================================================

fmt.Println("\nCall Pointer Receiver Method with Variable:")

// Declare a function variable for the method bound to the d variable.

// The function variable will get the address of d because the method

// is using a pointer receiver.

f2 := d.setAge

// Call the method via the variable.

f2(45)

// Change the value of d.

d.name = "Sammy"

// Call the method via the variable. We see the change.

f2(45)

}

CR5

// Sample program to show how to declare and use function types.

package main

import "fmt"

// event displays global events.

func event(message string) {

fmt.Println(message)

}

// data is a struct to bind methods to.

type data struct {

name string

age int

}

// event displays events for this data.

func (d \*data) event(message string) {

fmt.Println(d.name, message)

}

// =============================================================================

// fireEvent1 uses an anonymous function type.

func fireEvent1(f func(string)) {

f("anonymous")

}

// handler represents a function for handling events.

type handler func(string)

// fireEvent2 uses a function type.

func fireEvent2(h handler) {

h("handler")

}

// =============================================================================

func main() {

// Declare a variable of type data.

d := data{

name: "Bill",

}

// Use the fireEvent1 handler that accepts any

// function or method with the right signature.

fireEvent1(event)

fireEvent1(d.event)

// Use the fireEvent2 handler that accepts any

// function or method of type `handler` or any

// literal function or method with the right signature.

fireEvent2(event)

fireEvent2(d.event)

// Declare a variable of type handler for the

// global and method based event functions.

h1 := handler(event)

h2 := handler(d.event)

// User the fireEvent2 handler that accepts

// values of type handler.

fireEvent2(h1)

fireEvent2(h2)

// User the fireEvent1 handler that accepts

// any function or method with the right signature.

fireEvent1(h1)

fireEvent1(h2)

}

Interfaces

CR1

// Sample program that could benefit from polymorphic behavior with interfaces.

package main

import "fmt"

// file defines a system file.

type file struct {

name string

}

// read implements the reader interface for a file.

func (file) read(b []byte) (int, error) {

s := "<rss><channel><title>Going Go Programming</title></channel></rss>"

copy(b, s)

return len(s), nil

}

// pipe defines a named pipe network connection.

type pipe struct {

name string

}

// read implements the reader interface for a network connection.

func (pipe) read(b []byte) (int, error) {

s := `{name: "bill", title: "developer"}`

copy(b, s)

return len(s), nil

}

func main() {

// Create two values one of type file and one of type pipe.

f := file{"data.json"}

p := pipe{"cfg\_service"}

// Call each retrieve function for each concrete type.

retrieveFile(f)

retrievePipe(p)

}

// retrieveFile can read from a file and process the data.

func retrieveFile(f file) error {

data := make([]byte, 100)

len, err := f.read(data)

if err != nil {

return err

}

fmt.Println(string(data[:len]))

return nil

}

// retrievePipe can read from a pipe and process the data.

func retrievePipe(p pipe) error {

data := make([]byte, 100)

len, err := p.read(data)

if err != nil {

return err

}

fmt.Println(string(data[:len]))

return nil

}

CR2

// Sample program to show how polymorphic behavior with interfaces.

package main

import "fmt"

// reader is an interface that defines the act of reading data.

type reader interface {

read(b []byte) (int, error)

}

// file defines a system file.

type file struct {

name string

}

// read implements the reader interface for a file.

func (file) read(b []byte) (int, error) {

s := "<rss><channel><title>Going Go Programming</title></channel></rss>"

copy(b, s)

return len(s), nil

}

// pipe defines a named pipe network connection.

type pipe struct {

name string

}

// read implements the reader interface for a network connection.

func (pipe) read(b []byte) (int, error) {

s := `{name: "bill", title: "developer"}`

copy(b, s)

return len(s), nil

}

func main() {

// Create two values one of type file and one of type pipe.

f := file{"data.json"}

p := pipe{"cfg\_service"}

// Call the retrieve function for each concrete type.

retrieve(f)

retrieve(p)

}

// retrieve can read any device and process the data.

func retrieve(r reader) error {

data := make([]byte, 100)

len, err := r.read(data)

if err != nil {

return err

}

fmt.Println(string(data[:len]))

return nil

}

CR3

// Sample program to show how to understand method sets.

package main

import "fmt"

// notifier is an interface that defines notification

// type behavior.

type notifier interface {

notify()

}

// user defines a user in the program.

type user struct {

name string

email string

}

// notify implements the notifier interface with a pointer receiver.

func (u \*user) notify() {

fmt.Printf("Sending User Email To %s<%s>\n",

u.name,

u.email)

}

func main() {

// Create a value of type User and send a notification.

u := user{"Bill", "bill@email.com"}

// Values of type user do not implement the interface because pointer

// receivers don't belong to the method set of a value.

sendNotification(u)

// ./example1.go:36: cannot use u (type user) as type notifier in argument to sendNotification:

// user does not implement notifier (notify method has pointer receiver)

}

// sendNotification accepts values that implement the notifier

// interface and sends notifications.

func sendNotification(n notifier) {

n.notify()

}

CR4

// Sample program to show how you can't always get the address of a value.

package main

import "fmt"

// duration is a named type with a base type of int.

type duration int

// notify implements the notifier interface.

func (d \*duration) notify() {

fmt.Println("Sending Notification in", \*d)

}

func main() {

duration(42).notify()

// ./example3.go:18: cannot call pointer method on duration(42)

// ./example3.go:18: cannot take the address of duration(42)

}

CR5

// Sample program to show how the concrete value assigned to

// the interface is what is stored inside the interface.

package main

import "fmt"

// printer displays information.

type printer interface {

print()

}

// cannon defines a cannon printer.

type cannon struct {

name string

}

// print displays the printer's name.

func (c cannon) print() {

fmt.Printf("Printer Name: %s\n", c.name)

}

// epson defines a epson printer.

type epson struct {

name string

}

// print displays the printer's name.

func (e \*epson) print() {

fmt.Printf("Printer Name: %s\n", e.name)

}

func main() {

// Create a cannon and epson printer.

c := cannon{"PIXMA TR4520"}

e := epson{"WorkForce Pro WF-3720"}

// Add the printers to the collection using both

// value and pointer semantics.

printers := []printer{

// Store a copy of the cannon printer value.

c,

// Store a copy of the epson printer value's address.

&e,

}

// Change the name field for both printers.

c.name = "PROGRAF PRO-1000"

e.name = "Home XP-4100"

// Iterate over the slice of printers and call

// print against the copied interface value.

for \_, p := range printers {

p.print()

}

// When we store a value, the interface value has its own

// copy of the value. Changes to the original value will

// not be seen.

// When we store a pointer, the interface value has its own

// copy of the address. Changes to the original value will

// be seen.

}

CR6

// Sample program to show the syntax of type assertions.

package main

import (

"fmt"

"log"

)

// user defines a user in our application.

type user struct {

id int

name string

}

// finder represents the ability to find users.

type finder interface {

find(id int) (\*user, error)

}

// userSVC is a service for dealing with users.

type userSVC struct {

host string

}

// find implements the finder interface using pointer semantics.

func (\*userSVC) find(id int) (\*user, error) {

return &user{id: id, name: "Anna Walker"}, nil

}

func main() {

svc := userSVC{

host: "localhost:3434",

}

if err := run(&svc); err != nil {

log.Fatal(err)

}

}

// run performs the find operation against the concrete data that

// is passed into the call.

func run(f finder) error {

u, err := f.find(1234)

if err != nil {

return err

}

fmt.Printf("Found user %+v\n", u)

// Ideally the finder abstraction would encompass all of

// the behavior you care about. But what if, for some reason,

// you really need to get to the concrete value stored inside

// the interface?

// Can you access the "host" field from the concrete userSVC type pointer

// that is stored inside this interface variable? No, not directly.

// All you know is the data has a method named "find".

// ./example5.go:61:26: f.host undefined (type finder has no field or method host)

log.Println("queried", f.host)

// You can use a type assertion to get a copy of the userSVC pointer

// that is stored inside the interface.

svc := f.(\*userSVC)

log.Println("queried", svc.host)

return nil

}

CR7

// Sample program to show type assertions using the comma-ok idiom.

package main

import (

"fmt"

"log"

)

// user defines a user in our application.

type user struct {

id int

name string

}

// finder represents the ability to find users.

type finder interface {

find(id int) (\*user, error)

}

// userSVC is a service for dealing with users.

type userSVC struct {

host string

}

// find implements the finder interface using pointer semantics.

func (\*userSVC) find(id int) (\*user, error) {

return &user{id: id, name: "Anna Walker"}, nil

}

// mockSVC defines a mock service we will access.

type mockSVC struct{}

// find implements the finder interface using pointer semantics.

func (\*mockSVC) find(id int) (\*user, error) {

return &user{id: id, name: "Jacob Walker"}, nil

}

func main() {

var svc mockSVC

if err := run(&svc); err != nil {

log.Fatal(err)

}

}

func run(f finder) error {

u, err := f.find(1234)

if err != nil {

return err

}

fmt.Printf("Found user %+v\n", u)

// If the concrete type value stored inside the interface value is of the

// type \*userSVC, then "ok" will be true and "svc" will be a copy of the

// pointer stored inside the interface.

if svc, ok := f.(\*userSVC); ok {

log.Println("queried", svc.host)

}

return nil

}

CR8

// Sample program to show the syntax and mechanics of type

// switches and the empty interface.

package main

import "fmt"

func main() {

// fmt.Println can be called with values of any type.

fmt.Println("Hello, world")

fmt.Println(12345)

fmt.Println(3.14159)

fmt.Println(true)

// How can we do the same?

myPrintln("Hello, world")

myPrintln(12345)

myPrintln(3.14159)

myPrintln(true)

// - An interface is satisfied by any piece of data when the data exhibits

// the full method set of behavior defined by the interface.

// - The empty interface defines no method set of behavior and therefore

// requires no method by the data being stored.

// - The empty interface says nothing about the data stored inside

// the interface.

// - Checks would need to be performed at runtime to know anything about

// the data stored in the empty interface.

// - Decouple around well defined behavior and only use the empty

// interface as an exception when it is reasonable and practical to do so.

}

func myPrintln(a interface{}) {

switch v := a.(type) {

case string:

fmt.Printf("Is string : type(%T) : value(%s)\n", v, v)

case int:

fmt.Printf("Is int : type(%T) : value(%d)\n", v, v)

case float64:

fmt.Printf("Is float64 : type(%T) : value(%f)\n", v, v)

default:

fmt.Printf("Is unknown : type(%T) : value(%v)\n", v, v)

}

}

CRa

// Sample program that explores how interface assignments work when

// values are stored inside the interface.

package main

import (

"fmt"

"unsafe"

)

// notifier provides support for notifying events.

type notifier interface {

notify()

}

// user represents a user in the system.

type user struct {

name string

}

// notify implements the notifier interface.

func (u user) notify() {

fmt.Println("Alert", u.name)

}

func inspect(n \*notifier, u \*user) {

word := uintptr(unsafe.Pointer(n)) + uintptr(unsafe.Sizeof(&u))

value := (\*\*user)(unsafe.Pointer(word))

fmt.Printf("Addr User: %p Word Value: %p Ptr Value: %v\n", u, \*value, \*\*value)

}

func main() {

// Create a notifier interface and concrete type value.

var n1 notifier

u := user{"bill"}

// Store a copy of the user value inside the notifier

// interface value.

n1 = u

// We see the interface has its own copy.

// Addr User: 0x1040a120 Word Value: 0x10427f70 Ptr Value: {bill}

inspect(&n1, &u)

// Make a copy of the interface value.

n2 := n1

// We see the interface is sharing the same value stored in

// the n1 interface value.

// Addr User: 0x1040a120 Word Value: 0x10427f70 Ptr Value: {bill}

inspect(&n2, &u)

// Store a copy of the user address value inside the

// notifier interface value.

n1 = &u

// We see the interface is sharing the u variables value

// directly. There is no copy.

// Addr User: 0x1040a120 Word Value: 0x1040a120 Ptr Value: {bill}

inspect(&n1, &u)

}

Embedding

CR1

// Sample program to show how what we are doing is NOT embedding

// a type but just using a type as a field.

package main

import "fmt"

// user defines a user in the program.

type user struct {

name string

email string

}

// notify implements a method notifies users

// of different events.

func (u \*user) notify() {

fmt.Printf("Sending user email To %s<%s>\n",

u.name,

u.email)

}

// admin represents an admin user with privileges.

type admin struct {

person user // NOT Embedding

level string

}

func main() {

// Create an admin user.

ad := admin{

person: user{

name: "john smith",

email: "john@yahoo.com",

},

level: "super",

}

// We can access fields methods.

ad.person.notify()

}

CR2

// Sample program to show how to embed a type into another type and

// the relationship between the inner and outer type.

package main

import "fmt"

// user defines a user in the program.

type user struct {

name string

email string

}

// notify implements a method notifies users

// of different events.

func (u \*user) notify() {

fmt.Printf("Sending user email To %s<%s>\n",

u.name,

u.email)

}

// admin represents an admin user with privileges.

type admin struct {

user // Embedded Type

level string

}

func main() {

// Create an admin user.

ad := admin{

user: user{

name: "john smith",

email: "john@yahoo.com",

},

level: "super",

}

// We can access the inner type's method directly.

ad.user.notify()

// The inner type's method is promoted.

ad.notify()

}

CR3

// Sample program to show how embedded types work with interfaces.

package main

import "fmt"

// notifier is an interface that defined notification

// type behavior.

type notifier interface {

notify()

}

// user defines a user in the program.

type user struct {

name string

email string

}

// notify implements a method notifies users

// of different events.

func (u \*user) notify() {

fmt.Printf("Sending user email To %s<%s>\n",

u.name,

u.email)

}

// admin represents an admin user with privileges.

type admin struct {

user

level string

}

func main() {

// Create an admin user.

ad := admin{

user: user{

name: "john smith",

email: "john@yahoo.com",

},

level: "super",

}

// Send the admin user a notification.

// The embedded inner type's implementation of the

// interface is "promoted" to the outer type.

sendNotification(&ad)

}

// sendNotification accepts values that implement the notifier

// interface and sends notifications.

func sendNotification(n notifier) {

n.notify()

}

CR4

// Sample program to show what happens when the outer and inner

// type implement the same interface.

package main

import "fmt"

// notifier is an interface that defined notification

// type behavior.

type notifier interface {

notify()

}

// user defines a user in the program.

type user struct {

name string

email string

}

// notify implements a method notifies users

// of different events.

func (u \*user) notify() {

fmt.Printf("Sending user email To %s<%s>\n",

u.name,

u.email)

}

// admin represents an admin user with privileges.

type admin struct {

user

level string

}

// notify implements a method notifies admins

// of different events.

func (a \*admin) notify() {

fmt.Printf("Sending admin Email To %s<%s>\n",

a.name,

a.email)

}

func main() {

// Create an admin user.

ad := admin{

user: user{

name: "john smith",

email: "john@yahoo.com",

},

level: "super",

}

// Send the admin user a notification.

// The embedded inner type's implementation of the

// interface is NOT "promoted" to the outer type.

sendNotification(&ad)

// We can access the inner type's method directly.

ad.user.notify()

// The inner type's method is NOT promoted.

ad.notify()

}

// sendNotification accepts values that implement the notifier

// interface and sends notifications.

func sendNotification(n notifier) {

n.notify()

}

Exporting

CR1

// Package counters provides alert counter support.

package counters

// AlertCounter is an exported named type that

// contains an integer counter for alerts.

type AlertCounter int

// Sample program to show how to access an exported identifier.

package main

import (

"fmt"

"github.com/ardanlabs/gotraining/topics/go/language/exporting/example1/counters"

)

func main() {

// Create a variable of the exported type and initialize the value to 10.

counter := counters.AlertCounter(10)

fmt.Printf("Counter: %d\n", counter)

}

CR2

// Package counters provides alert counter support.

package counters

// alertCounter is an unexported named type that

// contains an integer counter for alerts.

type alertCounter int

// Sample program to show how the program can't access an

// unexported identifier from another package.

package main

import (

"fmt"

"github.com/ardanlabs/gotraining/topics/go/language/exporting/example2/counters"

)

func main() {

// Create a variable of the unexported type and initialize the value to 10.

counter := counters.alertCounter(10)

// ./example2.go:17: cannot refer to unexported name counters.alertCounter

// ./example2.go:17: undefined: counters.alertCounter

fmt.Printf("Counter: %d\n", counter)

}

CR3

// Package counters provides alert counter support.

package counters

// alertCounter is an unexported named type that

// contains an integer counter for alerts.

type alertCounter int

// New creates and returns values of the unexported type alertCounter.

func New(value int) alertCounter {

return alertCounter(value)

}

// Sample program to show how the program can access a value

// of an unexported identifier from another package.

package main

import (

"fmt"

"github.com/ardanlabs/gotraining/topics/go/language/exporting/example3/counters"

)

func main() {

// Create a variable of the unexported type using the exported

// New function from the package counters.

counter := counters.New(10)

fmt.Printf("Counter: %d\n", counter)

}

CR4

// Package users provides support for user management.

package users

// User represents information about a user.

type User struct {

Name string

ID int

password string

}

// Sample program to show how unexported fields from an exported struct

// type can't be accessed directly.

package main

import (

"fmt"

"github.com/ardanlabs/gotraining/topics/go/language/exporting/example4/users"

)

func main() {

// Create a value of type User from the users package.

u := users.User{

Name: "Chole",

ID: 10,

password: "xxxx",

}

// ./example4.go:21: unknown users.User field 'password' in struct literal

fmt.Printf("User: %#v\n", u)

}

CR5

// Package users provides support for user management.

package users

// user represents information about a user.

type user struct {

Name string

ID int

}

// Manager represents information about a manager.

type Manager struct {

Title string

user

}

// Sample program to show how to create values from exported types with

// embedded unexported types.

package main

import (

"fmt"

"github.com/ardanlabs/gotraining/topics/go/language/exporting/example5/users"

)

func main() {

// Create a value of type Manager from the users package.

u := users.Manager{

Title: "Dev Manager",

}

// Set the exported fields from the unexported user inner type.

u.Name = "Chole"

u.ID = 10

fmt.Printf("User: %#v\n", u)

}