**Methods**

// Declare a struct that represents a baseball player. Include name, atBats and hits.

// Declare a method that calculates a players batting average. The formula is hits / atBats.

// Declare a slice of this type and initialize the slice with several players. Iterate over

// the slice displaying the players name and batting average.

package main

import "fmt"

// player represents a person in the game.

type player struct {

name string

atBats int

hits int

}

// average calculates the batting average for a player.

func (p \*player) average() float64 {

if p.atBats == 0 {

return 0.0

}

return float64(p.hits) / float64(p.atBats)

}

func main() {

// Create a few players.

ps := []player{

{"bill", 10, 7},

{"jim", 12, 6},

{"ed", 6, 4},

}

// Display the batting average for each player.

for i := range ps {

fmt.Printf("%s: AVG[.%.f]\n", ps[i].name, ps[i].average()\*1000)

}

// Why did I not choose this form?

for \_, p := range ps {

fmt.Printf("%s: AVG[.%.f]\n", p.name, p.average()\*1000)

}

}

**Interfaces**

// Declare an interface named speaker with a method named speak. Declare a struct

// named english that represents a person who speaks english and declare a struct named

// chinese for someone who speaks chinese. Implement the speaker interface for each

// struct using a value receiver and these literal strings "Hello World" and "你好世界".

// Declare a variable of type speaker and assign the address of a value of type english

// and call the method. Do it again for a value of type chinese.

//

// Add a new function named sayHello that accepts a value of type speaker.

// Implement that function to call the speak method on the interface value. Then create

// new values of each type and use the function.

package main

import "fmt"

// speaker implements the voice of anyone.

type speaker interface {

speak()

}

// english represents an english speaking person.

type english struct{}

// speak implements the speaker interface using a

// value receiver.

func (english) speak() {

fmt.Println("Hello World")

}

// chinese represents a chinese speaking person.

type chinese struct{}

// speak implements the speaker interface using a

// pointer receiver.

func (\*chinese) speak() {

fmt.Println("你好世界")

}

func main() {

// Declare a variable of the interface speaker type

// set to its zero value.

var sp speaker

// Declare a variable of type english.

var e english

// Assign the english value to the speaker variable.

sp = e

// Call the speak method against the speaker variable.

sp.speak()

// Declare a variable of type chinese.

var c chinese

// Assign the chinese pointer to the speaker variable.

sp = &c

// Call the speak method against the speaker variable.

sp.speak()

// Call the sayHello function with new values and pointers

// of english and chinese.

sayHello(english{})

sayHello(&english{})

sayHello(&chinese{})

// Why does this not work?

// sayHello(chinese{})

}

// sayHello abstracts speaking functionality.

func sayHello(sp speaker) {

sp.speak()

}

**Embedding**

// Sample program to show how you can use embedding to reuse behavior from

// another type and override specific methods.

package main

import (

"fmt"

"log"

"time"

)

// Document is the core data model we are working with.

type Document struct {

Key string

Title string

}

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

// Feed is a type that knows how to fetch Documents.

type Feed struct{}

// Count tells how many documents are in the feed.

func (f \*Feed) Count() int {

return 42

}

// Fetch simulates looking up the document specified by key. It is slow.

func (f \*Feed) Fetch(key string) (Document, error) {

time.Sleep(time.Second)

doc := Document{

Key: key,

Title: "Title for " + key,

}

return doc, nil

}

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

// CachingFeed keeps a local copy of Documents that have already been

// retrieved. It embeds Feed to get the Fetch and Count behavior but

// "overrides" Fetch to have its cache.

type CachingFeed struct {

docs map[string]Document

\*Feed

}

// NewCachingFeed initializes a CachingFeed for use.

func NewCachingFeed(f \*Feed) \*CachingFeed {

return &CachingFeed{

docs: make(map[string]Document),

Feed: f,

}

}

// Fetch calls the embedded type's Fetch method if the key is not cached.

func (cf \*CachingFeed) Fetch(key string) (Document, error) {

if doc, ok := cf.docs[key]; ok {

return doc, nil

}

doc, err := cf.Feed.Fetch(key)

if err != nil {

return Document{}, err

}

cf.docs[key] = doc

return doc, nil

}

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

// FetchCounter is the behavior we depend on for our process function.

type FetchCounter interface {

Fetch(key string) (Document, error)

Count() int

}

func process(fc FetchCounter) {

fmt.Printf("There are %d documents\n", fc.Count())

keys := []string{"a", "a", "a", "b", "b", "b"}

for \_, key := range keys {

doc, err := fc.Fetch(key)

if err != nil {

log.Printf("Could not fetch %s : %v", key, err)

return

}

fmt.Printf("%s : %v\n", key, doc)

}

}

func main() {

fmt.Println("Using Feed directly")

process(&Feed{})

fmt.Println("Using CachingFeed")

c := NewCachingFeed(&Feed{})

process(c)

}

**Exporting**

// Package toy contains support for managing toy inventory.

package toy

// Toy represents a toy we sell.

type Toy struct {

Name string

Weight int

onHand int

sold int

}

// New creates values of type toy.

func New(name string, weight int) \*Toy {

return &Toy{

Name: name,

Weight: weight,

}

}

// OnHand returns the current number of this

// toy on hand.

func (t \*Toy) OnHand() int {

return t.onHand

}

// UpdateOnHand updates the on hand count and

// returns the current value.

func (t \*Toy) UpdateOnHand(count int) int {

t.onHand += count

return t.onHand

}

// Sold returns the current number of this

// toy sold.

func (t \*Toy) Sold() int {

return t.sold

}

// UpdateSold updates the sold count and

// returns the current value.

func (t \*Toy) UpdateSold(count int) int {

t.sold += count

return t.sold

}

// Create a package named toy with a single exported struct type named Toy. Add

// the exported fields Name and Weight. Then add two unexported fields named

// onHand and sold. Declare a factory function called New to create values of

// type toy and accept parameters for the exported fields. Then declare methods

// that return and update values for the unexported fields.

//

// Create a program that imports the toy package. Use the New function to create a

// value of type toy. Then use the methods to set the counts and display the

// field values of that toy value.

package main

import (

"fmt"

"github.com/ardanlabs/gotraining/topics/go/language/exporting/exercises/exercise1/toy"

)

func main() {

// Create a value of type toy.

t := toy.New("Bat", 28)

// Update the counts.

t.UpdateOnHand(100)

t.UpdateSold(2)

// Display each field separately.

fmt.Println("Name", t.Name)

fmt.Println("Weight", t.Weight)

fmt.Println("OnHand", t.OnHand())

fmt.Println("Sold", t.Sold())

}