Turn-Based Game With Go

Bora Kaşmer Follow
Dec 4, 2020 · 18 min read

Image Source: indivisle.jpg

Today we will talk about how to write a simple turn-based game engine with Go. While writing our character's abilities and fights with each other, we will use Interfaces, channels, and concurrency with Golang.

> "Reality is broken. Game designers can fix it."
> — Jane McGonigal

If you don't have any idea about the turn-based game, it is about the different types of characters with different strengths, and different abilities fight and kill each other one

by one. The winner is the survived last person.

Image Source: https://i.redd.it/y1h6d509uac31.jpg

Turn Game Rules

1. Fights mean every Warrier hit only the one Warrier in Turn. Every attacker loses stamina or mana after hit to an opponent.

2. Every Hit damage is changed by Warrier's Level and his Weapon, Magic, or Animal kind.

3. When a warrior is damaged by someone else, his or her blood will decreases. If the blood value is zero or less, he will die.

4. If a warrior's stamina or mana less than zero, he can't hit or make magic until the next Turn. Next Turn, his mana or stamina will be full.

Image Source:-south-park-scontri-di-retti-quando-la-lotta-si-fa-intestina-le-9252.jpg

Firstly let's create base traits struct of these types of characters.

heroCharacter/heroCharacter.go: All the characters have Name, Level (experience), Attacks(Damage List) and Blood(Energy)

```
package heroCharacter

type Hero struct {
   Name    string
   Level   int
   Attacks map[string]int
   Blood   int
}
```

Let's Create all Characters Interface. We will list all character's common Actions in this interface.

fightHero/fightHero.go: Every hero Hits the other characters, takes Damage after attacked by someone else. We can get info about him or her healthy, stamina or mana. And finally, we can check he or she died or not. These are the common functions of all Characters.

```
package fightHero

type FightHero interface {
   Hit() int
   TakeDamage(damage int)
   GetInfo() (string, int)
   IsDeath() bool
}
```

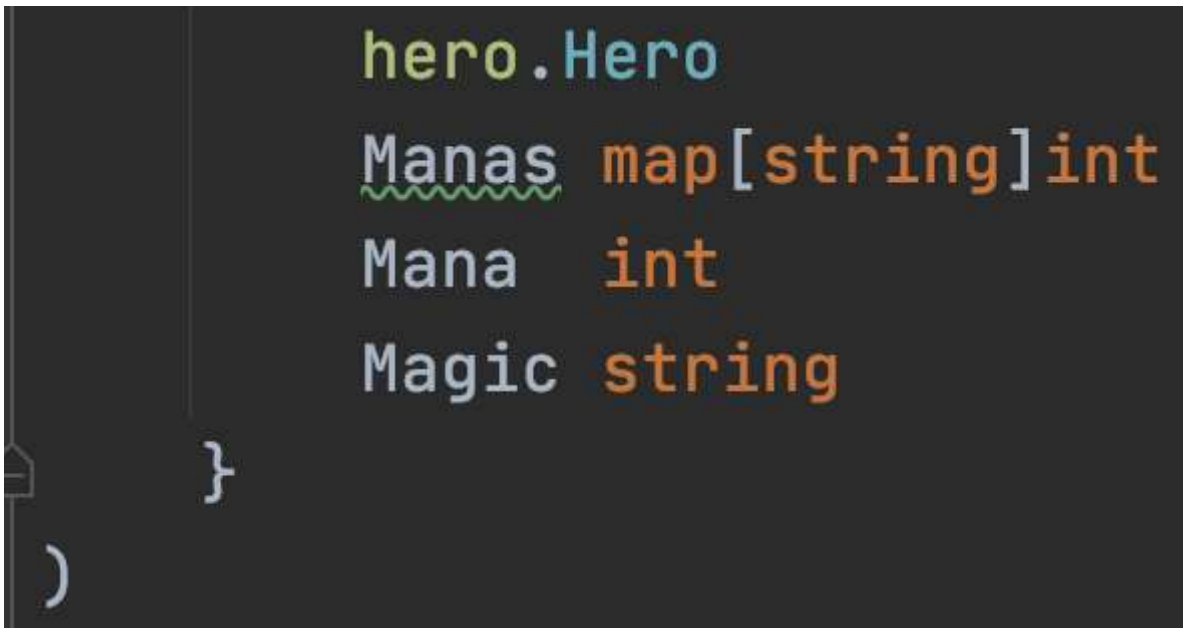
Image Source: il_1588xN.1656567706_93wq.jpg

Let’s create first character Wizard.

wizard/wizard.go:

Properties of Wizard:

```
type (
    Wizard struct {
```



- Hero: Common character's properties.
- Manas: This is a collection of "string" magic names with their "int" mana costs. **Example**: "Manas: map[string]int{"FireBall": 5, "Thunder": 10, "Ghost Attack": 30}}"
- Mana: This is integer Wizard's mana value. Wizard needs mana, for making Magic. Without mana Wizard is nothing :)
- Magic: This is Wizard's weapon. He or she can hit an enemy with the Magic. **Example**: "map[string]int{"FireBall": 25, "Thunder": 18, "Ghost Attack": 30}"

Actions of Wizard:

FightHero interface Methods / Hit, TakeDamage, GetInfo and IsDeath:

Hit():

This is "fightHero" interface method. And this is one of the common Actions of all heroes.

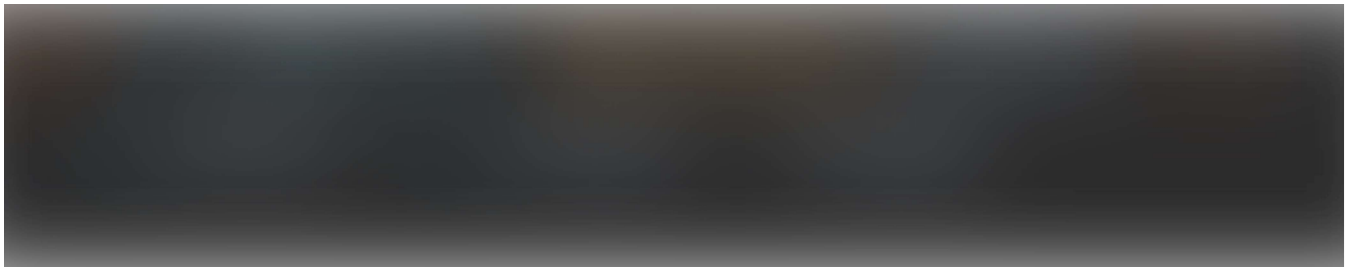
```
func (w Wizard) Hit() int {
    if w.Mana >= w.Manas[w.Magic] {
        levelEffect := int(math.Ceil(float64(w.Level) * 0.1))
        return w.Attacks[w.Magic] + levelEffect
    } else : 0
}
```

> ***In Go, all structs must declare all the methods of Interface.***

- ”*if w.Mana >= w.Manas[w.Magic] {*“: A wizard should have more or equal mana than the magic’s mana, which he hit the enemy with it.
- “*levelEffect := int(math.Ceil(float64(w.Level) * 0.1))*” : This is the level effect of the damage to the enemy. We will talk about the level later. But more experience means more damage.
- “*return w.Attacks[w.Magic] + levelEffect*”: This is the total Magic and Level damage of enemy.
- “*else return 0*": If the Wizard doesn’t have enough mana for the magic, he hits zero damage to the enemy.

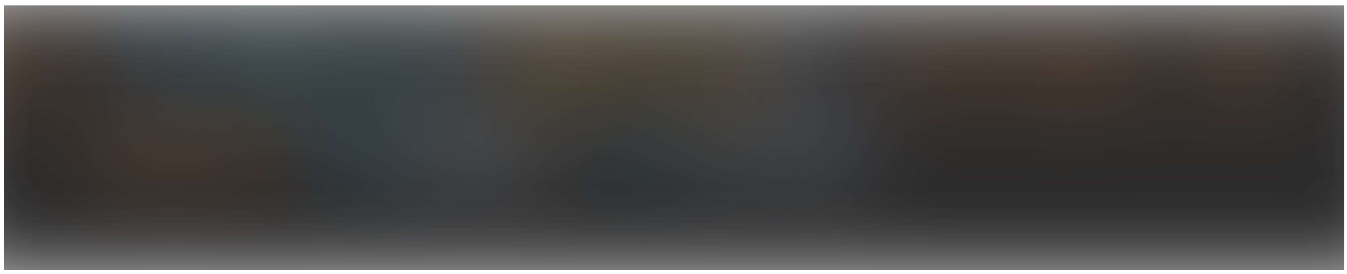
TakeDamage():

If another character hits you, your blood will be reduced. If the blood’s value becomes zero or less, you will die. If you look carefully, we will use **the** * pointer value of the wizard. Because when he takes damage, the referanced wizard’s blood must be reduced.



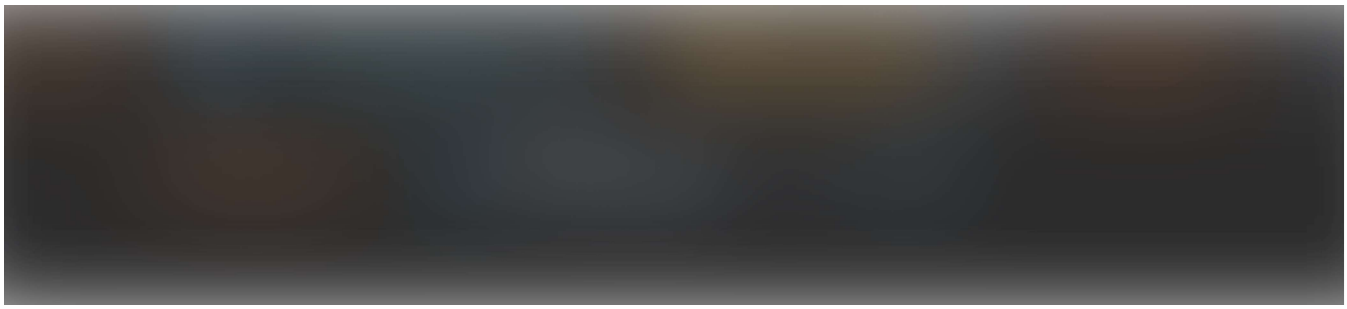
GetInfo():

This is one of the interface’s methods. We will return two values. It is used for getting the Name and Blood value of the Wizard.



IsDeath():

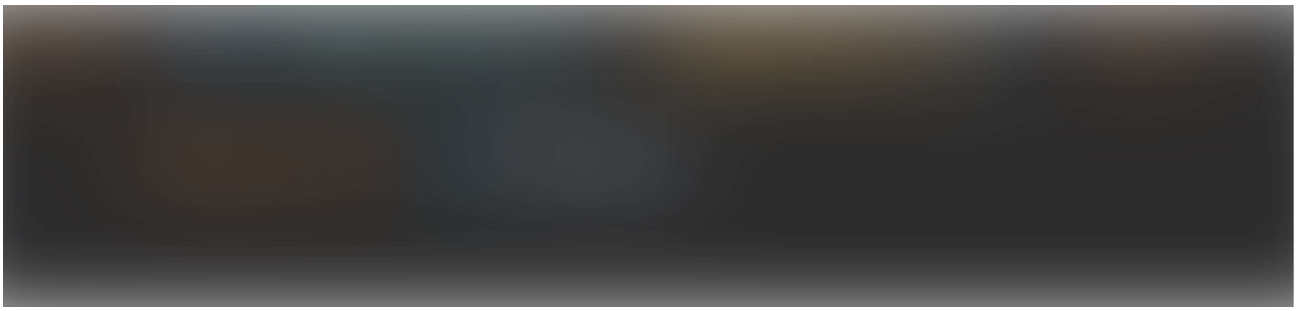
This is the last method of the interface. We will check the Blood value of the wizard. If it is more than zero, this means he or she lives. So we will return false. But if it is zero or less than zero, we will return true because he or she died.



Wizard Own Methods:

GetMana():

This method is special to the wizard. Because Wizards need mana for making magic, we will get the value of mana.



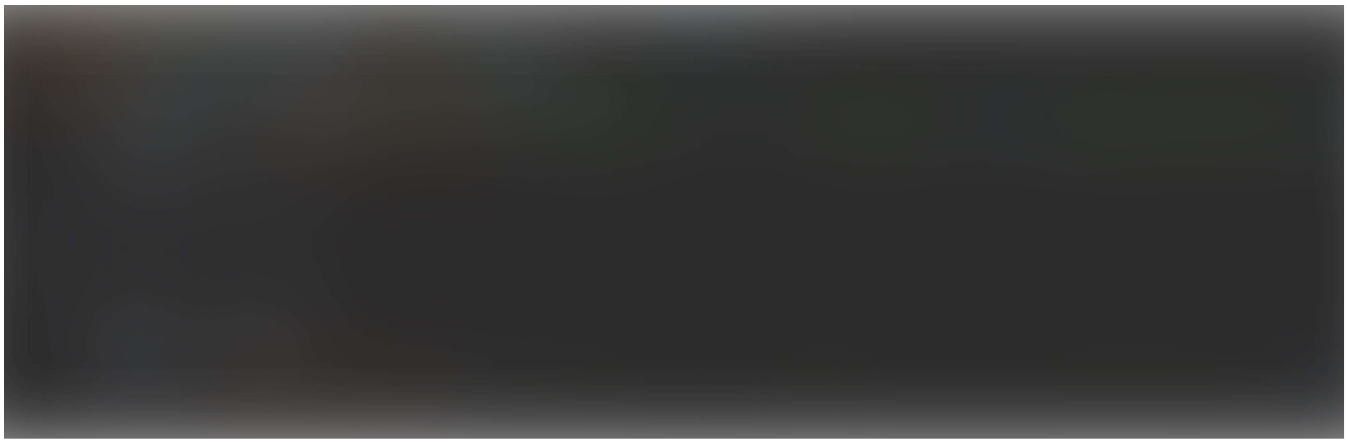
SpendMana()

This method is special to the wizard. We use the * **pointer** value of the wizard here too. Because when a wizard makes magic, the original wizard's mana must be reduced. Manas is "**map[string]int**" list. Every magic spends a different value of mana. We keep these values in this list.

CreateWizard():

This is the constructor of the wizard struct. We will create a default wizard character with default Attacks and Manas map. And we will set blood and mana values randomly. Finally, we will return the pointer address of this created wizard struct.

- "Attacks" is a "**map[string]int**" list. Every magics' attack is damaged different value to the enemy. We keep these values in this list.
- "Manas" is a "**map[string]int**" list too. Every magic spends different value mana from the wizard. We keep these values in this list too.



wizard/wizard.go:

```
package wizard

import (
   "math"
   hero "turnBaseGame/heroCharacter"
)

type (
   Wizard struct {
      hero.Hero
      Manas map[string]int
      Mana  int
      Magic string
   }
)

func (w Wizard) Hit() int {
   if w.Mana >= w.Manas[w.Magic] {
      levelEffect := int(math.Ceil(float64(w.Level) * 0.1))
      return w.Attacks[w.Magic] + levelEffect
   } else {
      return 0
   }
```

```
}

func (w *Wizard) TakeDamage(damage int) {
   w.Blood = w.Blood - damage
}

func (w Wizard) GetInfo() (string, int) {
   return w.Name, w.Blood
}

func (w Wizard) GetMana() int {
   return w.Mana
}

func (w *Wizard) SpendMana() {
   if w.Mana >= w.Manas[w.Magic] {
      w.Mana = w.Mana - w.Manas[w.Magic]
   }
}

func (w Wizard) IsDeath() bool {
   return w.Blood <= 0
}

func CreateWizard(blood int, mana int) *Wizard {
   return &Wizard{Hero: hero.Hero{
      Attacks: map[string]int{"FireBall": 25, "Thunder": 18, "Ghost
Attack": 30},
      Blood: blood,
   },
      Mana:  mana,
      Manas: map[string]int{"FireBall": 5, "Thunder": 10, "Ghost
Attack": 30}}
}
```

Image Source: b22a65a1de4e5f32d8f1346c73d25d19.png

fighter/fighter.go:

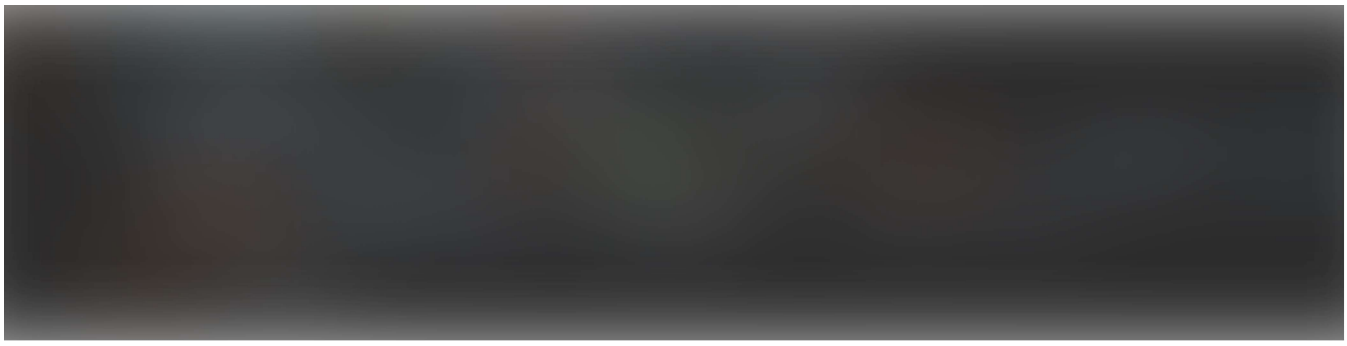
Properties of fighter:

Hero property is the same as the wizard.

Hit():

This is “fightHero” interface method. And this is one of the common Actions of all heroes.

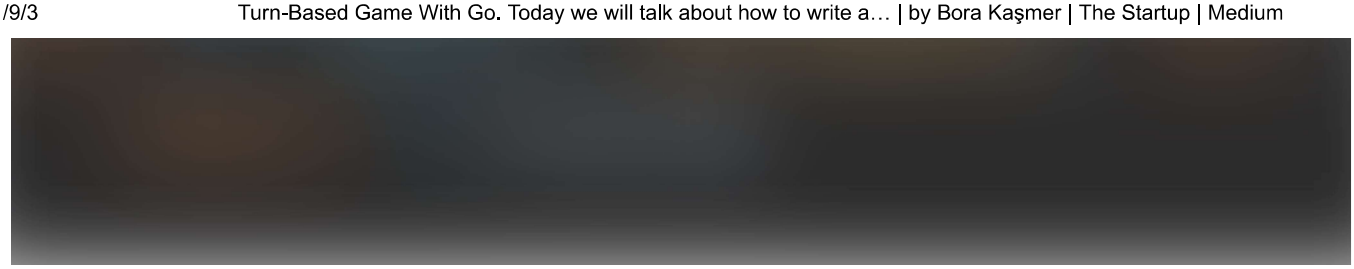
Like Wizard’s magic, a Fighter should have enough stamina to hit the enemy with a weapon. If he doesn’t have, he can not hit the enemy for one turn. After one turn, his stamina fills with randomly. And he can hit this time. We have to calculate the damage value by his level and selected weapon type.



TakeDamage(), **GetInfo()** and **IsDeath()** functions are the same as at Wizard struct.

GetStamina():

This method is special to the fighter. It is like GetMana() method at Wizard struct. But this time, the fighter needs stamina for using weapons, not for making magic. We will get the value of stamina.



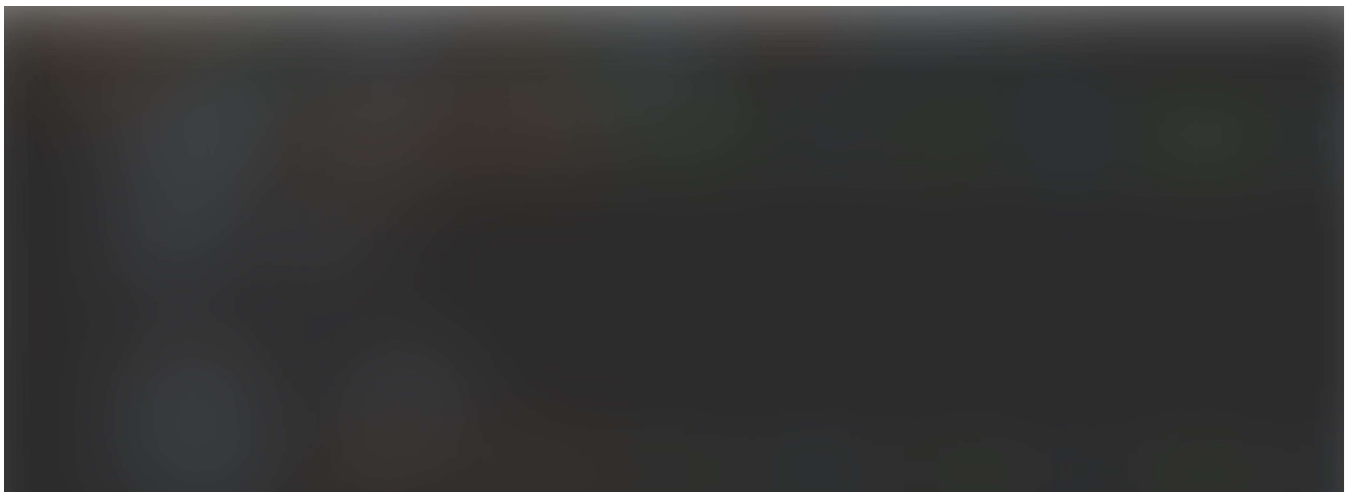
SpendStamina():

This method is special to the Fighter. We use the * **pointer** value of the Fighter here too. Because when a Fighter hit the enemy with the weapon, the original Fighter's stamina must be reduced. Stamina is the "**map[string]int**" list. Every weapon spends a different value of stamina. We keep these values in this list.

CreateFighter():

This is the constructor of the fighter struct. We will create a default fighter character with default Attacks and Stamina map. And we will set blood and stamina values randomly. Finally, we will return the pointer address of this created fighter struct.

- "Attacks" is a "**map[string]int**" list. Every weapons' attack is damaged different value to the enemy. We keep these values in this list.
- "Staminas" is a "**map[string]int**" list too. Every weapon spend's different value stamina from the fighter. We keep these values in this list too.



fighter/fighter.go:

```
package fighter

import (
   "math"
   hero "turnBaseGame/heroCharacter"
)

type Fighter struct {
   hero.Hero
   Staminas map[string]int
   Stamina  int
   Weapon   string
}

func (f Fighter) Hit() int {
   if f.Stamina >= f.Staminas[f.Weapon] {
      levelEffect := int(math.Ceil(float64(f.Level) * 0.1))
      return f.Attacks[f.Weapon] + levelEffect
   } else {
      return 0
   }
}

func (f *Fighter) TakeDamage(damage int) {
   f.Blood = f.Blood - damage
}

func (f Fighter) GetInfo() (string, int) {
   return f.Name, f.Blood
}

func (f Fighter) GetStamina() int {
   return f.Stamina
}

func (f *Fighter) SpendStamina() {
   if f.Stamina >= f.Staminas[f.Weapon] {
      f.Stamina = f.Stamina - f.Staminas[f.Weapon]
   }
}

func (f Fighter) IsDeath() bool {
   return f.Blood <= 0
}

func CreateFighter(blood int, stamina int) *Fighter {
   return &Fighter{Hero: hero.Hero{
      Attacks: map[string]int{"Stick": 15, "Axe": 30, "Sword": 40},
      Blood: blood,
   },
```

```
        Stamina:  stamina,
        Staminas: map[string]int{"Axe": 10, "Stick": 5, "Sword": 30},
    }
}
```

Image Source: vignetta.wikia

druid/druid.go:

Properties of druid:

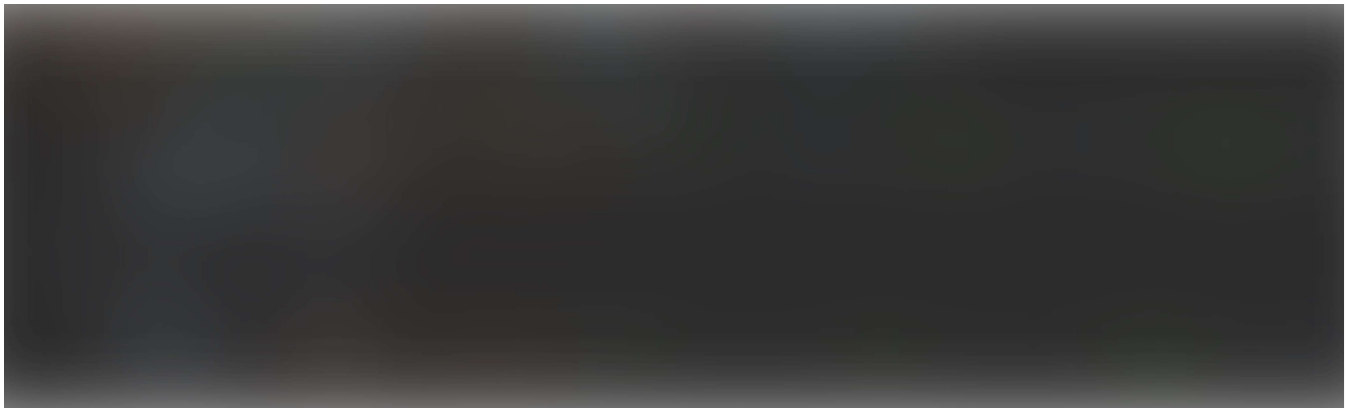
Hero property is the same as the wizard.

Actually Hit(), TakeDamage(), GetInfo(), GetMana(), SpendMana() and IsDeath() functions are the same at Wizard struct.

CreateDruid():

This is the constructor of the druid struct. We will create a default druid character with default Attacks and Manas map. And we will set blood and mana values randomly. Finally, we will return the pointer address of this created druid struct.

- “Attacks” is a “**map[string]int**” list. Every animal’ attack is damaged different value to the enemy. We keep these values in this list.
- “Manas” is a “**map[string]int**” list too. Every animal creation spends different value mana from the druid. We keep these values in this list too.



druid/druid.go:

```
package druid

import (
   "math"
   hero "turnBaseGame/heroCharacter"
)

type (
   Druid struct {
      hero.Hero
      Manas  map[string]int
      Mana   int
      Animal string
   }
)

func (d Druid) Hit() int {
   if d.Mana >= d.Manas[d.Animal] {
      levelEffect := int(math.Ceil(float64(d.Level) * 0.1))
      return d.Attacks[d.Animal] + levelEffect
   } else {
      return 0
   }

}

func (d *Druid) TakeDamage(damage int) {
   d.Blood = d.Blood - damage
}

func (d Druid) GetInfo() (string, int) {
   return d.Name, d.Blood
```

```
}

func (d Druid) GetMana() int {
   return d.Mana
}

func (d *Druid) SpendMana() {
   if d.Mana >= d.Manas[d.Animal] {
      d.Mana = d.Mana - d.Manas[d.Animal]
   }
}

func (d Druid) IsDeath() bool {
   return d.Blood <= 0
}

func CreateDruid(blood int, mana int) *Druid {
   return &Druid{Hero: hero.Hero{
      Attacks: map[string]int{"Wolf": 15, "Bear": 35, "Sheap": 5},
      Blood: blood,
   },
      Mana:  mana,
      Manas: map[string]int{"Wolf": 5, "Bear": 15, "Sheap": 1}}
}
```

> Life is a game, where either you lose or you learn.
> —Robert Kiyosaki

hero/hero.go:

We have defined all the characters. Now it's time to get all the heroes to fight each other. Only one of them will survive and win.

Firstly import all these packages

```
package main

import (
   "fmt"
   "math/rand"
   "reflect"
   "runtime"
   "strconv"
   "strings"
   "time"
   "turnBaseGame/druid"
   IHero "turnBaseGame/fightHero"
   "turnBaseGame/fighter"
```

```
    "turnBaseGame/wizard"
)
```

Declare global variables and Random functions:

```
var fighterList map[string]IHero.FightHero
var fighterNumberList map[int]IHero.FightHero

func GetRandomID(limit int) int {
   rand.Seed(time.Now().UnixNano())
   rndVictim := rand.Intn(limit) + 1
   //fmt.Printf("Random : %d \n", rndVictim)
   return rndVictim
}
func GetRandomBetweenID(minLimit int, maxlimit int) int {
   rand.Seed(time.Now().UnixNano())
   rndVictim := rand.Intn(maxlimit-minLimit) + minLimit + 1
   //fmt.Printf("Random : %d \n", rndVictim)
   return rndVictim
}

var isFighterDead bool = false
var isWizardDead bool = false
var isDruidDead bool = false

var TotalLive int = 3

func MapRandomKeyGet(mapI interface{}) interface{} {
   keys := reflect.ValueOf(mapI).MapKeys()

   return keys[rand.Intn(len(keys))].Interface()
}
```

- fighterList[] is used for storing the fighters by name. It is the map[string] of Hero.FightHero interface.
- fighterNumberList[] is used for storing the string name of fighters by their ID. It is the map[string] of Hero.FightHero interface.
- GetRandomID is used for getting a random number until the limit parameter.
- GetRandomBetweenID is used for getting a random number between minLimit and maxLimit.
- isFighterDead, isWizardDead, isDruidDead: These variables are kept the living state of every Fighter.

- TotalLive: We have three kinds of fighters, so in the beginning, we have three lives.
- MapRandomKeyGet is used for getting random magic for Wizzard or random weapon for Fighter or random animal for Druid.

hero/main-1():

Firstly we will create a default wizard, fighter, and druid characters. Every fight waits for each other. So we will use channels for sequential battle. After the fight finish, it will call the next battle.

If you pay attention to the picture above, every hero's blood, mana, and stamina are set randomly between 50 and 100.

GOMAXPROCS(1) means, at the same time, a maximum of only one OS thread that is executing code simultaneously. We will create three channels as above. Firstly we will set "fighterChan". Next, we will set "wizardChan", and finally, we will set "druidChan". And all the fights will follow this channel order. Channels are used for communicating between goroutines.

Image Source:golang-map-4.png

- We will keep all fighters by name in the "**fighterList**" map.
- We will use "**fighterNumberList**" map like an enum. We will select fighters randomly from this list. A **map** in **Golang** is a collection of unordered pairs of key-value.

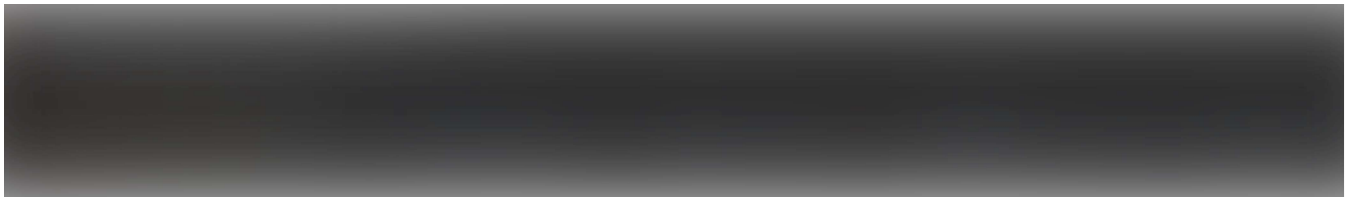
Main-1():

```
func main() {

   merlin := wizard.CreateWizard(GetRandomBetweenID(50, 100),
GetRandomBetweenID(50, 100))
   merlin.Name = "Bora"
   merlin.Level = GetRandomID(60)

   barbar := fighter.CreateFighter(GetRandomBetweenID(50, 100),
GetRandomBetweenID(50, 100))
   barbar.Name = "Barbar"
   barbar.Level = GetRandomID(60)

   forest := druid.CreateDruid(GetRandomBetweenID(50, 100),
GetRandomBetweenID(50, 100))
   forest.Name = "Forest"
   forest.Level = GetRandomID(60)

   runtime.GOMAXPROCS(1)
   fighterChan := make(chan *fighter.Fighter)
   wizardChan := make(chan *wizard.Wizard)
   druidChan := make(chan *druid.Druid)

   fighterList = map[string]IHero.FightHero{"fighter": barbar,
"wizard": merlin, "druid": forest}
   fighterNumberList = map[int]IHero.FightHero{1: barbar, 2: merlin,
3: forest}
```

hero/main-2():

In the second part, We will make the characters fight in an infinite loop until only one hero can stand. "**for TotalLive > 1 { //Fight to the death :)**"

- "***wg := &sync.WaitGroup{}***": We will use the pointer of "WaitGroup" object to wait for multiple goroutines to finish.
- "***lockObject := &sync.Mutex{}***": We will use pointer Mutex for avoiding from Race condition satiations.
- ***"for TotalLive > 1***": Infinite loop, until only one hero can stand.

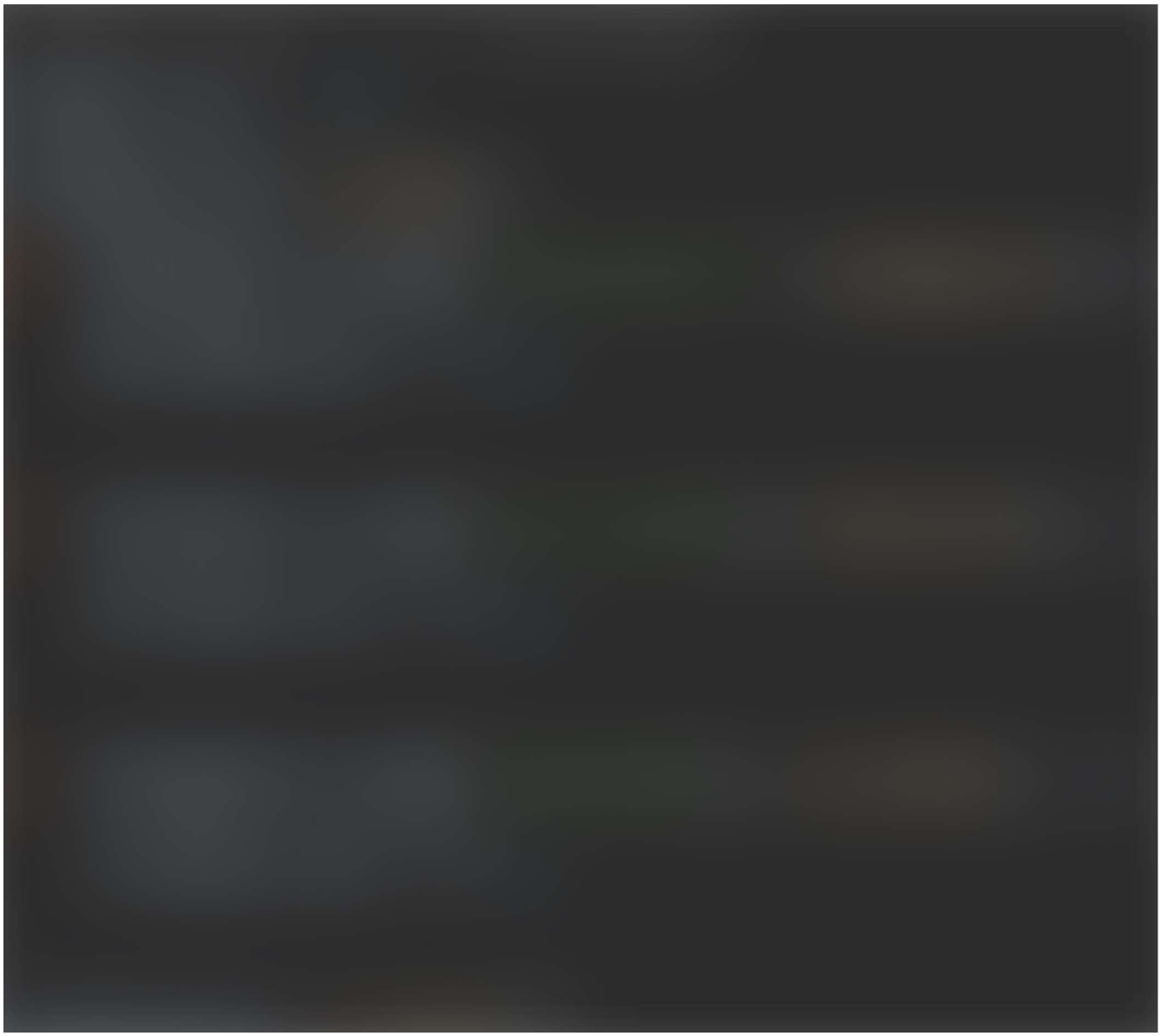


We will create a hero channel in each cycle, and two of them will fight in each round. In all four rounds, each hero must fight each other.

- "**wg.Add(1)**": We will use "WaitGroup" because we should wait to finish all the channels process before exiting the application.

Not: Every channel will set the next channel after the end of the turn. With this, every fight will happen synchronously.

- "***<-time.After(time.Millisecond * 100)***" : Every for loop, we will wait 100 miliseconds.



At every end of the 4 loops, we will check all heroes living state. If two of them die, we will stop the Infinite loop, and we will announce the champion. So we will check TotalLive variable. If it is one, the loop will finish. And we will never forget to call the "**Lock()**" method for not to be race condition and of course call Unlock() method at the end of the checking Character's "**IsDeath()**" reference from the "fighterList."

In every loop, we will call "**turnFights()**" function for fighting two of characters.

Main-2():

```
wg := &sync.WaitGroup{}
lockObject := &sync.Mutex{}

for TotalLive > 1 { //Fight to the death :)

   z := strings.Repeat("-", 100)
   fmt.Println(z)
   stage++
   fmt.Printf("STAGE %d \n", stage)
   fmt.Println(z)

   for i := 1; i < 5; i++ {
      wg.Add(1)
      go turnFights(i, wizardChan, fighterChan, druidChan,wg,
lockObject)
   }

   <-time.After(time.Millisecond * 100)

   //Check Who is Standing!
   TotalLive = 0

   lockObject.Lock()
   if !fighterList["fighter"].IsDeath() {
      TotalLive += 1
   }
   if !fighterList["wizard"].IsDeath() {
      TotalLive += 1
   }
   if !fighterList["druid"].IsDeath() {
      TotalLive += 1
   }
   lockObject.Unlock()
}

wg.Wait()
close(fighterChan)
close(wizardChan)
close(druidChan)

}
```

- *"**wg.Wait":*** *W*e should wait to finish all the goroutines finish before exiting the application.
- **close**(fighterChan), **close**(wizardChan), **close**(druidChan): We have to close all channels before the exit from the application.

hero/turnFights()-1:

We are waiting for three kinds of channels pointers and WaitGroup and Mutex as a parameter in this function. And which channel is sent to this function, its fighter will be fighting with one of the other two warriors. Every time only one of the channels will be sent to this function.

> ***channel <- data : Write data to channel,***
> ***data<-channel : Read data from channel***

```
func turnFights(turn int, wizardChan chan *wizard.Wizard,
fighterChan chan *fighter.Fighter, druidChan chan *druid.Druid, wg
*sync.WaitGroup, lockObject *sync.Mutex) {

   select {
   case wiz, ok := <-wizardChan:
    .
    .
   case fig, ok := <-fighterChan:
    .
    .
   case dru, ok := <-druidChan:
    .
    .
    default:
      var _fighterAttack, _ = fighterList["fighter"].
(*fighter.Fighter)
      fighterChan <- _fighterAttack
```

```
    }
    wg.Done()
}
```

Firstly we will start from the default case:

```
go turnFights(i, wizardChan, fighterChan, druidChan, wg,
lockObject)
```

When we first call the "**turnFights()**" function from the "**Main()**" function, all the channels are null. So none of the cases is accurate, and finally, we will come to the default case.

- Firstly, we will get a fighter character from the fighterList[] map[string] with his name "fighter".
- "**.(*fighter.Fighter)**": This is the parse action on Go. When we take an object from the map[string] list, its type is "**IHero.FightHero**" interface because all Characters come from the same interface. So when we take a fighter from this map, we have to parse it to the Fighter struct.
- "**var _fighterAttack, _**": We will set this Fighter struct to the _fighterAttack variable. "_" We will not consider the parser result for this situation.
- "**fighterChan <- _fighterAttack**": We set the pointer of "fighterChan" with this "_fighterAttack" Fighter character.

> ***Next time, when we call this turnFights() function in the loop, "fighterChannel" will not be null!***

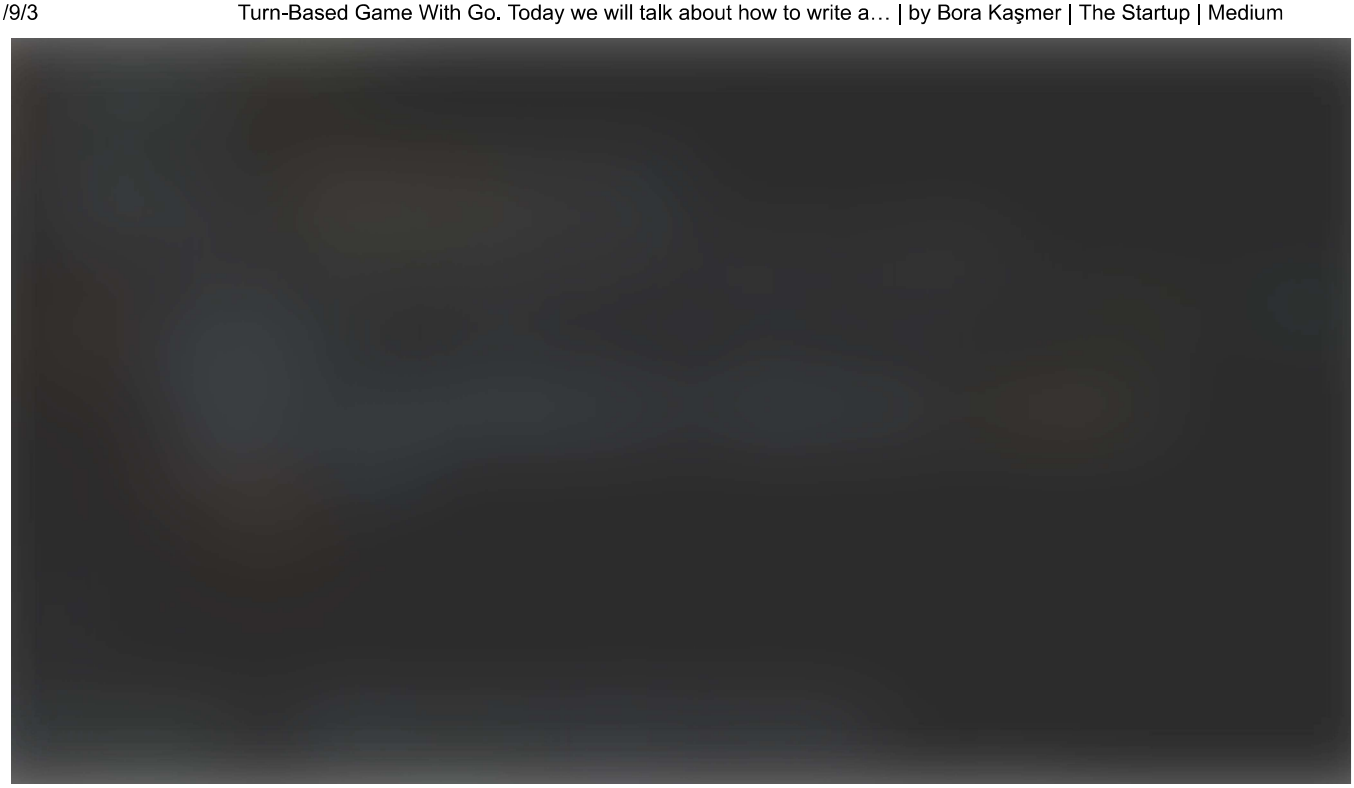
hero/turnFights()-2:

Fighter channel: After the default case, "***fighterChan"*** (fighter channel) is not null anymore. So we will have the fighter character fight another random character.

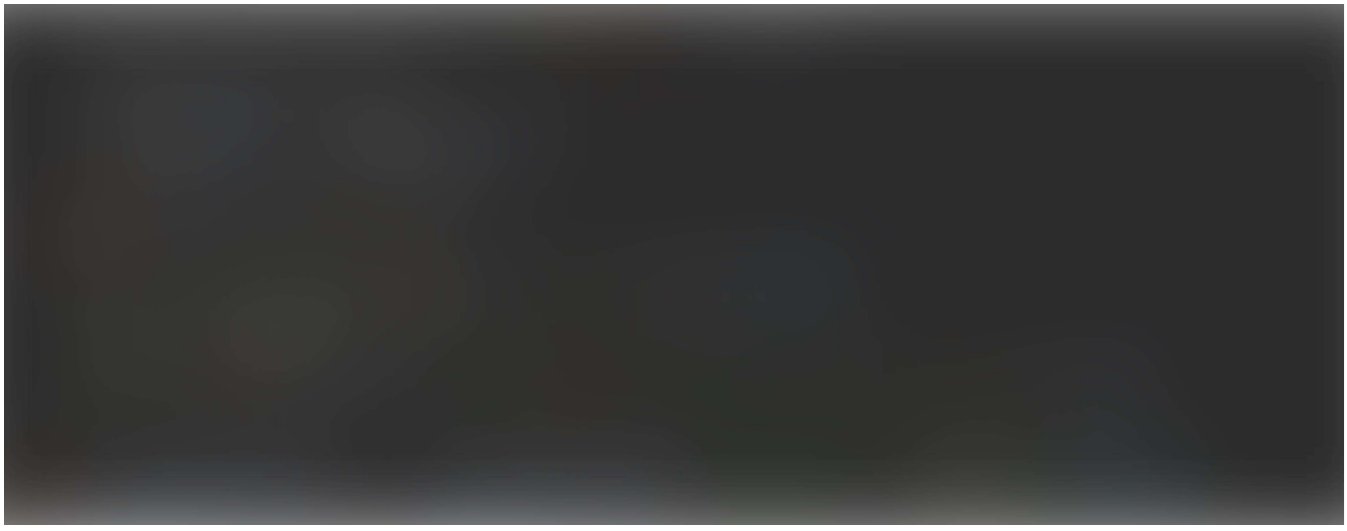
```
case fig, ok := <-fighterChan:
   if ok && fig.Blood > 0 { //If fighter is not dead
      fig.Weapon = MapRandomKeyGet(fig.Staminas).(string)
      //SelectRandom Victim
      var rndVictim int
      for {
         rndVictim = GetRandomID(3)
         //If selected Victim different from the Fighter
         var _, fighterOk = fighterNumberList[rndVictim].
(*fighter.Fighter)
         if !fighterOk {
            _, blood := fighterNumberList[rndVictim].GetInfo()
            if blood > 0 {
               break
            }
         }
      }
      randomVictim := fighterNumberList[rndVictim]
      //We Selected Random Victim Different From the Fighter

      fmt.Printf("TURN %d \n", turn)

      lockObject.Lock()
      fight(fig, randomVictim)
      lockObject.Unlock()
   } else {
      z := strings.Repeat("#", 100)
      fmt.Println(z)
      fmt.Printf("TURN %d Fighter is Dead..\n", turn)
   }
   var _wizardAttack, _ = fighterList["wizard"].(*wizard.Wizard)
   wizardChan <- _wizardAttack
```

- case fig, ok := <-fighterChan: If fighter channel is not null, we will set “***fighter.Fighter**” to the fig variable.

- “fig.Blood>0”: We will check fighter is live or not. If he is not live, he can not fight :)
- MapRandomKeyGet(fig.Staminas).(string): We will get Random Weapon from stamina “map[string]int” and cast it to the string.



- “for{:” We will try to find an opponent in a loop until we find it.
- “rndVictim = GetRandomID(3)” : We will get a random number between 1 to 3
- “var _, fighterOk = fighterNumberList[rndVictim].(*fighter.Fighter)”: We will get a character from the “fighterNumberList” map randomly and try to pars to Fighter. If he is not a fighter, we can select him or her as an opponent. If he is, we will select another random character.
- “_, blood := fighterNumberList[rndVictim].GetInfo()”: If this newly selected opponent is live, we will set this character to the “randomVictim” variable. If he or she is dead, we will select another random character for an opponent.



- “fmt.Printf(“TURN %d \n”, turn)”: We will write the Turn number to the console.

> *“**lockObject.Lock()**”: Mutex.Lock object locks before the called “fight ()” function. The region where the fight () function is located in the **Critical Section**. If goroutines write the same location on the memory while working, this area is called the “**critical section.**” This is an unwanted situation because the result can not be correct when the goroutines change the variable at the same time.*

```
fight(fig, randomVictim)
```

- We will talk about “fight()” function in the next section. It takes two parameters. “Fighter” and random opponent character.
- “else{“: This mean fighter is dead and can not fight.

```
var _wizardAttack, _ = fighterList["wizard"].(*wizard.Wizard)
```

- At the end of this case condition, we will get Wizard from the map[string]IHero.FighHero and cast it to the Wizard with =>.(*wizard. Wizard) and set it to the _wizardAttack

```
wizardChan <- _wizardAttack
```

- And finally, we will set the Wizzard Channel variable (“wizardChan”) with _wizardAttack character.

So next Turn, “wizardChan” will not null, and Wizard will fight against one of the characters.

> “Chaos was the law of nature; Order was the dream of man.” — Henry Adams

hero/turnFights()-3:

Wizard channel: Everything is almost the same as the Fighter channel. We will talk about only different ones.

- “case wiz, ok := <-wizardChan:” This time wizard Channel is not null.
- “wiz.Magic = MapRandomKeyGet(wiz.Manas).(string)” : We will select Wizard’s magic randomly.

```
var _druidAttack, _ = fighterList["druid"].(*druid.Druid)
druidChan <- _druidAttack
```

- At the end of the case condition, we will pick up the druid character and set it to the druidChan with this fighter.

- And finally, we will set the Durrid Channel variable (druidChan) with _druidAttack character.
- **So next Turn, "druidChan" will not null, and Druid will fight against one of the characters.**

hero/turnFights()-4:

Druid channel: Everything is almost the same as the Fighter channel. We will talk about only different ones.

- "case dru, ok := <-druidChan:" This time druid Channel is not null.

- “dru.Animal = MapRandomKeyGet(dru.Manas).(string)” : We will select Druid’s animal randomly.

> ***Finally, all three characters fight each other. Only one winner will be. If there is more than one warrior whose blood value is higher than 0, the cycle is restarted again.***

Image Source:dead-in-vinland-steam-review.jpg

fight()-1:

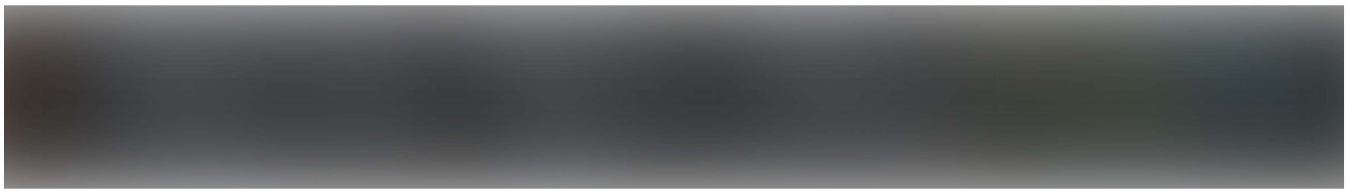
Two characters fight each other. fight() function get two parameters. One of the attacker and the other one is a defender.

```
func fight(attacker IHero.FightHero, target IHero.FightHero) {
   var _wizardAttacker, wizardOk = attacker.(*wizard.Wizard)
   var _fighterAttacker, fighterOk = attacker.(*fighter.Fighter)
   var _druidAttacker, druidOk = attacker.(*druid.Druid)

   damageValue := attacker.Hit()
   target.TakeDamage(damageValue)

   var attackerMana, attackerStamina int
   if wizardOk {
      _wizardAttacker.SpendMana()
      attackerMana = _wizardAttacker.GetMana()
   } else if fighterOk {
      _fighterAttacker.SpendStamina()
      attackerStamina = _fighterAttacker.GetStamina()
```

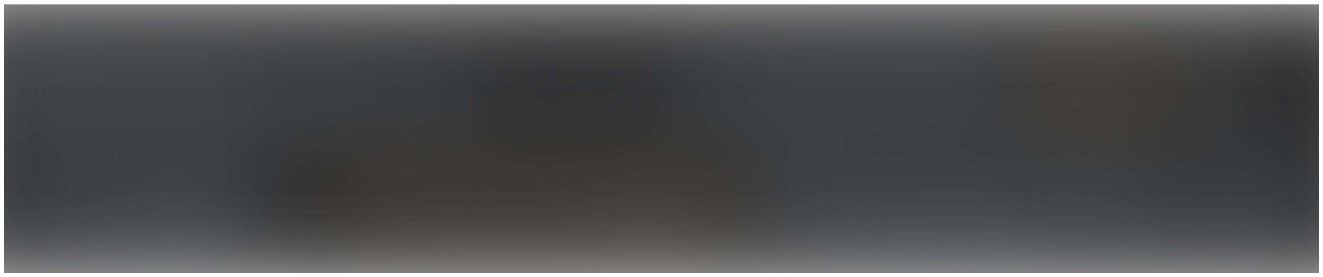
```
} else if druidOk {
   _druidAttacker.SpendMana()
   attackerMana = _druidAttacker.GetMana()
}

nameAttacker, _ := attacker.GetInfo()
nameTarget, bloodTarget := target.GetInfo()
.
.
```

In the below picture, we will check the attacker character type. Wizard, Fighter or Druid comes from the “IHero.FightHero” interface. It is the same as the attacker parameter type. So we try to parse this attacker struct to all three characters. Which one returns true; it is the type of attacker.

> ***Not: We will get both attacker and target parameter with their pointers.***



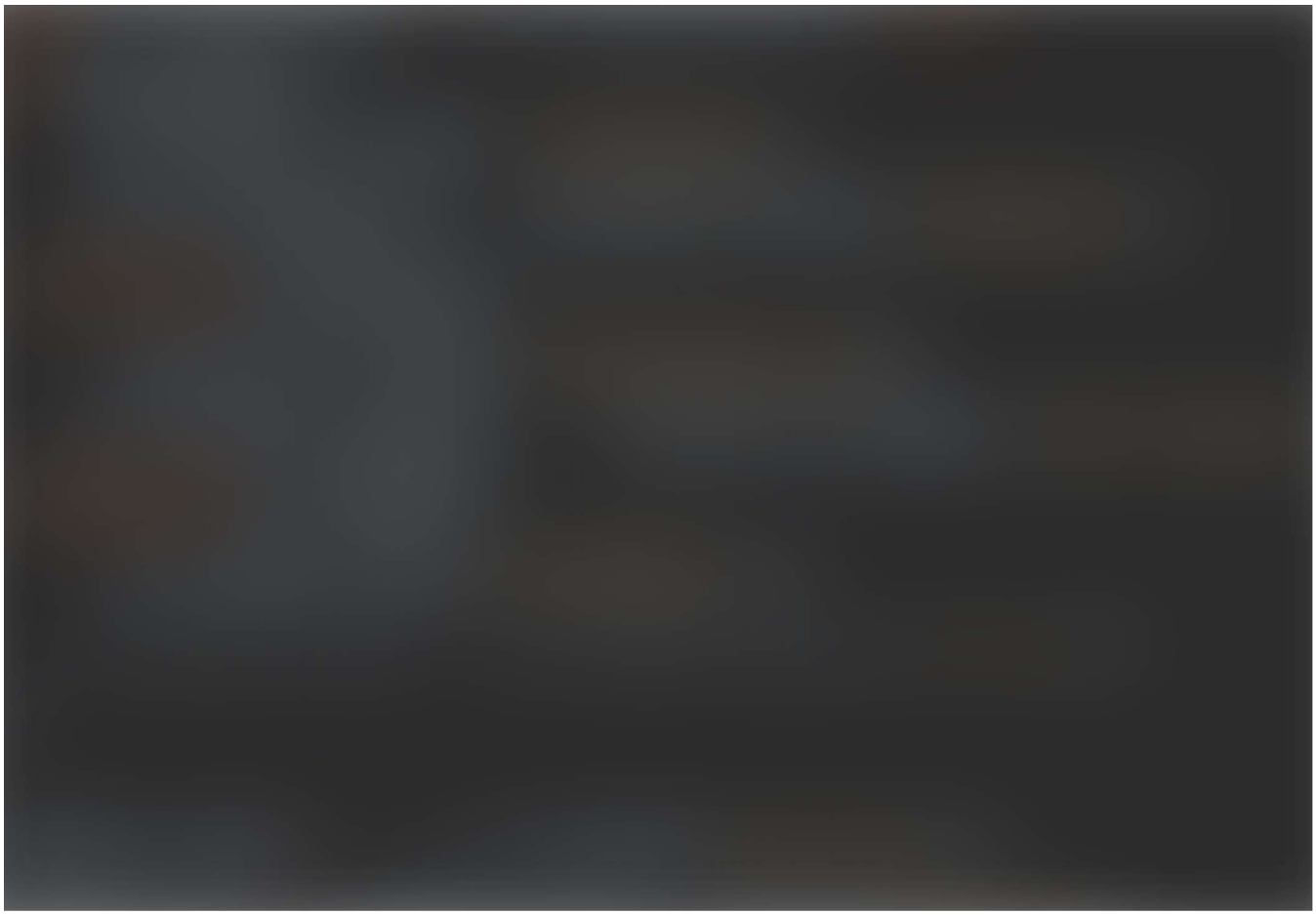
As seen in the picture below, we will get the attacker damaged value, and with TakeDamage() function, we will subtract this value from the target’s blood value. Don’t forget to add “levelEffect” to “damageValue”. Every character’s Hit point is changed by his level and magic, animal, or weapon kind. We send the target as a pointer. So when the amount of blood decreases, this means that the origin’s blood decreased.



_wizardAttacker.SpendMana(): As seen in the picture below, if the attacker is Wizard, we should spend as much mana as the magic needs. Fighter and Druid are the same.
attackerMana = _wizardAttacker.GetMana(): We will get the final state of the Wizard’s mana. Fighter and Druid are the same.
“**nameAttacker, _ := attacker.GetInfo()**”: For writing to console, we will get the

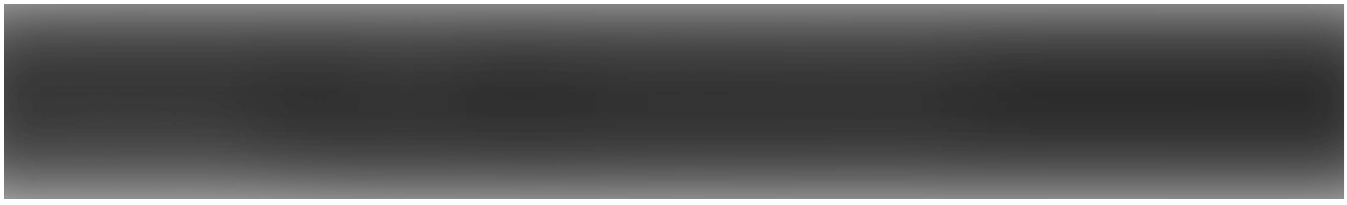
attacker's name.

"**nameTarget, bloodTarget := target.GetInfo()**": Again for writing to console, we will get the target's name and blood.



fight()-2:

We will write the result of the wizard fighting to the console.



- "***(lvl-"+strconv.Itoa(_wizardAttacker.Level)+")"+nameAttacker***"=> **(lvl-45)Bora**: Attacker Level and Name write the console.
- "***""+_wizardAttacker.Magic+""***"=> **'FireBall'**: Wizards's attack spell.
- "***nameTarget+"("+strconv.Itoa(bloodTarget+damageValue)+")'ya"***"=> **Forest(55) :** Target Name and his blood value before being attacked.
- "***damageValue***"=> **30 hasar** : We will write the hit point to the console.

- “ **if damageValue > 0 { fmt.Println(nameAttacker, attackerMana,**”=>If the attacker could hit the opponent, we will write his final stamina or mana value to the console with his name.

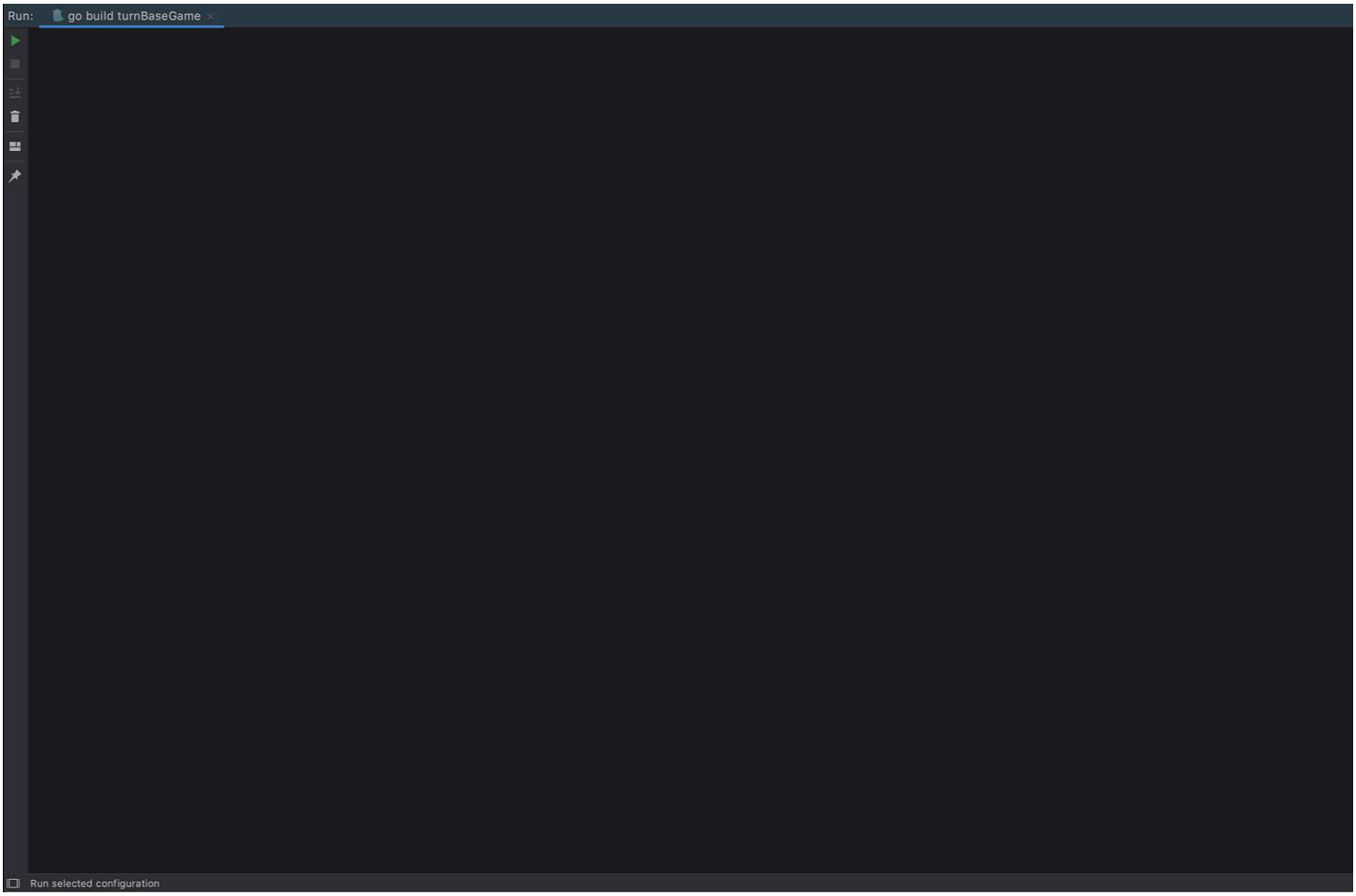
> *Not: If the attacker couldn’t hit the enemy, this means he is not enough mana or stamina for striking the opponent. So he must wait one turn to fill his mana or stamina.*

- “***fmt.Println(nameTarget+””nın”, “kalan canı “, bloodTarget,)***”: Target final blood value write console with his name.

If this turn Attacker could not hit the opponent, this means his mana or stamina is not enough for the weapon or magic. So we will fill the mana or stamina randomly between 50 to 100 for the next turn.

Other characters, Fighter and Druid, are the same of the above Notify Codes.

> ***There is no interface related to the course of the war in the game. As seen above, the status is monitored with the notifications sent to the console.***



Conclusion:

In this article, we tried the write turn-based game's core, basically with Go. Firstly we created our game's hero structs with the same interface. All of them have unique

abilities. We built new fight rules and fought every warrior with each other. They effected with their life, mana or stamina after that.

We met lots of technical problems and solved them with different techniques, like managing concurrency(WaitGroup), race condition(Mutex), critical section, pointer problems, goroutines communication(Channel) and some object-oriented implementation(Interface, Extension, Factory) problems. I shared some of them with you. I hope they will help you to save your time.

"If you have read so far, first of all, thank you for your patience and support. I welcome all of you to ***my blog*** *for more!"*

Source Code: https://github.com/borakasmer/TurnBaseGame

Video(Turkish): https://youtu.be/AjMW5Jrp9ok

Source:

- **https://golangbot.com/channels/**
- **https://medium.com/better-programming/common-go-pitfalls-a92197cd96d2**
- **https://www.pluralsight.com/courses/concurrent-programming-go**

- https://medium.com/swlh/go-a-tale-of-concurrency-a-beginners-guide-b8976b26feb
- https://medium.com/xebia-engineering/understanding-interfaces-in-go-978edc3eaaf7

Go Goroutines Concurrent Turn Based Strategy Games Channel