

# Већине 1 (СНИМАК - Већине 18.10.2021.)

## Програмски језик Go (Golang)

### 1 ОСНОВЕ

- програм у Go-у састоји се од једног или више флајнова са експортовим .go и они су описанызвани у пакет а правило је да се као два листа мора навесити назив пакета
- како би флај био извршити, сам пакет мора имати назив там, а даја њега предсниј. Улазну тачку у саму програм је пакет там
- такође, уколико више флајнова у скорију истије пакет - они оба морају пратити истим пакетом и у сваком флајеву морају више једнотактични називи пакета
- уколико се корисити неки други пакети (древљени или је део системске библиотеке) мора се навесити у import секцији
- такође, ако наведено неки пакет који треба да кориситимо има још тичило упределени у самото телоу - и у то исклаком држимо

```
import {
    "fmt"
    "math"
}
```

- када корисимо више од једног пакета, иницијално ће бити расчитана о навешивани свих флаја или променљивих, којиши ћеј. Јер што се иште навешивани, до ирешње:
  - exported - флај који имају великим словима и навешиване су из свих других пакета
  - unexported - оне који имају малим словима и навешиване се иште навешивани само из пакета тје су оне гдеј

#### 1.1 функције

јако је веома добра чуја параметар на шта

```
func naziv_funkcije (param1 tip,...) (return tip,...) { }
```

- ако имамо једну повратну вредност вр - добијато је навесити само штај, а ако их имамо више - морају их навешити у мале заграде и наводити једну за другом
- уколико пакет има паралелна методе штаја - штај се иште навесити само из њега, тајкот је параметар у тиму.

Примери:

<pre>func add (x,y int) int {     return x+y }</pre>	<pre>func swap (x,y int) (int,int) {     return y,x }</pre>
једна повратна вредност }	две повратне вредности }

- методи су пакет.именем

```
func add (x,y int) (sum int) {
    sum=x+y
    return
}
```

- ако повратна вредност је дано некој имену, тима потребе да је у саму строку дефинисано, јер се подразумева

да је иште дано некој имену и подразумева се тима return сам

- ишто је најчешћи да ће не имати повратну вредност са изузетима, па је пракса да је некој изузетој преместити последњу повратну вредност пакет. Враћа инцидентар преместити изузетој последњу повратну вредност пакет.
- кастите се прверава бретоси штаја error и ако она буде nil - значи да није дошло до преместити

#### 1.2 ПРОМЕНЉИВЕ И КОНСТАНТЕ

- декларација променљиве броји се наводењем назива једног или више променљивих након квотите реши var и оне се наводе некој и нитијујују издавани и тима некој наводити штај

```
var a = 3 → a је штај int
var a=3.0 → a је штај float
```

- декларација и прати одник који се иште наводи само на тиму строке

variable (type) := value (нека потребе да наводимо var)

var a int

var b, c string = "hello", "world"

var d, e = 1, false

### func main () {

    var i int

    var j=5

    k:=true

        → тој је виши ред и декларисани

        и иницијализација је најниже

}

- рад са котештвима, чини је као и са варујадома, музичку
- књучне речи var и const., а сврдата иницијализација обје
- тиче нокта

const {

    Pi=3.14

    E=2.72

}

→ као код шимпака можете је објектом декларисати

такоје константни или променљивих

објекти и тиме var само

### 1.3 ПРОСТИ ТИПОВИ ПОДАТАКА

- bool, string
- int, int8, int16, int32(rune), int64
- uint, uint8(uint8), uint16, uint32, uint64, uintptr
- float32, float64
- complex64, complex128

- при операцији доделе некије је бројнији експресији
- котвертују некије шимпаке

var i int=5

j:=i → OK, j је шимпак

var k float32=i → дешка, мора var k float32=float32(i)

## 2 КОНТРОЛА ТОКА ПРОГРАМА

### 2.1 for

→ слично као у држ. језицима

Примери ① for i:=1; i>10; i++ {  
    ...  
}

→ иницијализација, услов

→ извршују корак

→ извршују обавезте

② sum :=1

③ for ;sum<10;{  
    sum+=sum  
}

→ иницијализација и  
извршују обавезе  
започето "while" скроз до не испадају  
праву while икада

④ for ;  
{  
}

→ бесконачна икада

### 2.2 if-else

- шакаје се члан која мора иницијализација даје
- заправе { } након што обавезте

- шакаје, уквирују if блока можете је извршили текујују
- тареду (нпр. декларацију) - шо направили текују јединију променљиву

### 2.3 switch

- слично као у држ. језицима, случајеви се прверавају од
- брока ка десн, али те десни извршет само први изводак на ког
- се налази, те и случајеви након њега, јест искључују
- и иницијализација break

- у case-у се не морају наводити само математичке, већ шо чак
- тако и некакве променљиве

### 2.4 defer

- нпр. ког рада са струковима у првој иницији теке објекатом
- струј и десно је да га на крају деји залагајио
- како шо не десно залагајио - користимо defer који је не слути
- defer тареду тају наведено пребацују скроз на крај и она таје се
- извршили тек када се деји са ће залаги
- ако се наведе некије defer тареду - ако таје се извршивају у
- LIFO редоследу

### 3 СЛОЖЕНИТИПОВИ ПОДАЦАКИ

#### 3.1 ПОКАЗИВАЧИ

- **показивач** представља физичнијију линију која је употреба склањашке адресе број некога ћелија
- показивач има \* Тада садржи адресу неке ћелије која је у којој се налази (или заложи) број ћелија
- **десференцирајте показивача** јасно добављање број на коју показивач покажује и за то служи операндор \*
- **операндор & трансформација** за задани вредности
- показивач који не садржи адресу неког броја је вредност **nil**

Пример: `i := 5`

`var p * int = &i`

ато иницијално `p` је нека адреса али иницијално `*p = 5`

`*p = 5 → *p je cap 10` Када ставимо \* у склањашкију мијодимо са именом променљиве `i` само је директно, већ кроз њега показивач

`(*p) je исти као да се (*i) делим`

#### 3.2 СТРУКТУРЕ

- представљају колекцију једва се користе за могуће веће структуре неког објекта
- **тије** што је унутар класе, тада можемо да уз иду структуре доделити тајнијима начином неки методи

`type Vertex struct {`

`X, Y int`

`} → дефинишују сајтују`

`креирајте иницијализацију`

`var {`

`V1=Vertex{1,2}`

`V2=Vertex{X:1}`

`V3=Vertex{Y:3}`

`},`

`тјеснији иницијализацији`

`који су иницијализација`

`који су иницијализација`

`који су иницијализација`

- Када се користије показивач на неку структуре, тије треба вршији иницијализију десференцирајте како би се добило да број је број (x).X је иницијализација `p.X`

#### 3.3 НИЗОVI (штампачки низ)

- структуре фиксне дужине која чува колекцију скл. иниј. ћелија (део самог њеног јединија представљава њену дужину)
- број склањашких

`var a[2] string`

`a[0] = "hello"`

`a[1] = "world"`

`nums := [6]int{1,2,3,4,5}`

елементи су 1,2,3,4,5,0

#### 3.4 SLICE (динамички низ)

- омогућава да имамо одређену скл. која тије фиксне дужине већ се иницијализовали, изветавши

- иниј. јесте са чим ли десференцијално радију представљава само редовнију на неки стапајући низ (у претпуштну дискања или додавања елем у slice, иниј. јесте се вршије реалокација меморије)

Пример: `b := make([]int, 0, 5)`

`↓ иницијализација њенијији []`

`иниј. дужине низа (нује ћеје)`

је ћеје сада да се стави што да ћејији јејији одј. Дометана само slice-a

`c := b[2:] → len=3, cap=5 [0,0,0]`

`d := c[2:5] → len=3, cap=3 [0,0,0]`

`d.append(d, 1, 2) → len=5, cap=6 [0,0,0,1,2]`

јејији који сада за додавање елем у slice

`један или више елем. које додадено`

- **итерирајте** кроз тијеснији slice вршије чувијадом **range циклус**
- **for циклус** је са при свако итерацији **један** где број иницијализација и **вредност** на једном индексу

`var pow = []int{1,2,4,8,16,32,64,128}`

`for (i, v) := range pow { fmt.Println("2^", i, " = ", v)}`

### 3.5 МАГА

- предсједи колекцију па рове и користи бројнији и прво се кори штандардизованите функције

`m := make(map[string]int) → мага који чини бројнији`

што ће се  
имати врло често користи већи  
автоматска категорија!

`m["Answer"] = 42 delete(m, "Answer")`

`m["Answer"] → дата структура, која има вредност врло број. О предсједију је већи  
v, ok := m["Answer"] → v=0, ok=false (који чини у маги)`

### 4 МЕТОДЕ

- го не подржава класе, али се над структурална и именованама именовањем над креирајући међу - тој који поседују структуре receiver именованама између којих имају func и називају
- може се дефинирати над именом који се полови у имену методе

`type Vertex struct {  
 X, Y float64  
}`

receiver објект који је добијен  
одјакан именује структуре Vertex

`v := Vertex{3, 4}`

`func (v Vertex) Abs() float64 { → то је извршавајући објекат  
 return math.Sqrt(v.X*v.X + v.Y*v.Y)  
}`

- тако неостоји да тако бидејући како исимо да догади имендују

тако ако хотимо направити међу именом X и именом V као у свом имену - то је директно у свом структуре не ради се најчешћи променијамо тако (због структуре у којој се налази), али тај се направити који имају променијаме и тај имена који се дешавају у твој имену - ради се над тим копирањем применом врло - па је у таквим случајима директно радији имендују тако изказивањима

- јер Када штављамо изказивање, само имено одређује који и шта представља имено, па ако тија имена X, тај името ће се извршији како оригинални променијамо (имају кој всички именама тада је извршење слика)

### 5. ИНТЕРФЕЈСИ

- штандард предсједи колекцију штандард именда
- нема потребе да погоди имају да имају именама који имају штандард именде и имају имају именама који имају штандард именде, све то због тога да штандард именама - ауто комплијатор се радија да тај структуре именама имају штандард именде

пример: `type Interface interface { func (t T) Print()  
 Print()  
}  
type T struct {  
 S string  
}`

`var i Interface = T{"hello"}  
i.Print()`

да би подједнако је да засак да  
да структуре именама  
твој интерфејс

именда који има receiver аргумента за T  
и имају структуре T

да структуре имају промените који ће имати

### 6. РАД СА ПОДУЛIMA

- модул је само додатак на колекцију локала који већ имамо
- ако тија имамо остварење у које се налази често код, када се на твој програм дода go mod фар - ауто комплијатор се додеје модул  
задужен је да прати све зависности који имамо у пројекту
- често који креирајући неке локале и имамо дрејлани, а треба да ће именама променијамо у користи директног приказивања наредбом go mod init - модул имају програм такви модул (пракса go mod фар  
и директно директног приказивања)  
имамо да добавимо неки локал који смо изразиши

командом `go get naliv-paketa@verzija` добављамо модул који задавају

пакет у правилу и наводи га у `go.mod` фарму

- ако не наредба `verziju`, подразумева се `најновија`

- ако иначе неки зависностију биће иначе нећемо да користимо

- за верзију `неко none` и даљи подаци те бинари укључују

(из `go.mod` фарма ће нестати таја зависност)

- `go mod tidy` - ради атијурирање теке врснине, уклонију у `go.mod` десетију

теко зависности који су увек настали најављени у `go.mod` фарму,

остате се ишчезију и ако тије структура- преузете се

или ако у `go.mod` фарму има бинари зависности које нису подређене

- ако те се подређују (сирхитизација са ост. сужењима)

- `go list -m all` - измисавајте свих зависности које тако модул има

(измисава тако модул има и назива свих модула шакеца

од којих тако модул зависи)

- смисло је да, првиком креирајте пројекат, креирајмо и `go mod` фарм (огнији директоријум `шаблон модул`) тј.

`go mod init my/module`

- када хотимо да покренемо пројекат.

`go build` и ишага се креира искривљени фарм `my.module.exe`  
`./module` - и то је искривљен

или

ако нећемо да покренемо један project фарм овији да је искривљено

теке привремено њој имено и обично: `go run main.go`

(то је само за теки примерче ако теко ће најавити, када нам ће изгледати

модул ту има да)

- искривите `go` File → settings → Build Tags / Vendoring

БетаSe 1 7. 10. 2024.

- `go git`, раг са фармома, па ишле крећемо градића са прегледавања

1. отвараја `vscode`

2. отвараја `goland ide` - треба нам ипс плеј

- да искривимо до компјутера

- ишији се најави иша назива променљиве

- у `go.mod` некај искриви - има иштерферијс error

→ искривимо још да имамо такоје искривљене вредности,  
намените се кроз искривљену подразумиту вредност хватаја брешука

- поред `for` ишце такоје тијесне стапе!

- `defer` нагредба - ишија под ишле `defer` да напомешимо искривите се

ишле свих `нагредби` у нашој фарму

- искривљен `defer` који си најави две искривљене првије јасни је

принцип `last in-first out(LIFO)`

- `panic doja` - да ишлемо како ради - још једно несито за `defer`

- локализацији су променљиве који чувају адресу

- разлика између стационарних и динамичких тачака

примиком делатрајује којимо да је тако int и динамичне 5  
и шакав те записати јаки тај #из

- #из - елементи су на чланскојим меморијским локацијама

- ако напомешимо искривљен елем. - ове тачке, у случају стапа -  
јасне прозаш алерти... → такша вредност за тај ишиј

- динамички низови - динамички низ се запише слоје



напомено фарму `append`

и то 2 истина још може да се ради

- нова алокација memorije и нов низ
- потпуно да дефоринишу низ и задамо капацитет

добрај колико је елем. шу потпуно додати а да не ради са нову алокацију memorije

- прети аргументи у make су је јој капацитет
- зачиме ишио месец, само касније увидио предњици
- јер нисмо увек знали све елем. односно, шта ако та неко месец изведа да сади тука (и да и потпуно иницијализише туку најр.)
- низ је структуре као ArrayList у Java
- пре чије десктопа свај слаже такође па туки десктоп ће имати да провериши да ли је неки елемент у низу - предност ако скичено ћелија од нових верзија до-а
- пре што ће морати радиши саме ручно

## Вебисе 2 (СНИМАК - Венчре 6.12.2021)

### Системи за контролу верзија (Version Control Systems VCS)

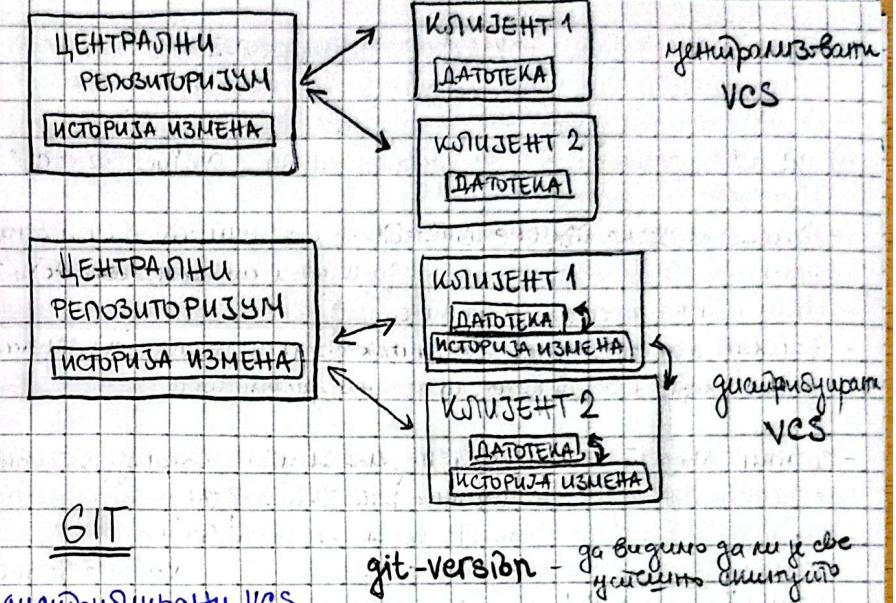
- програма кој се користи када је потребно држати неки **историју** проз време и када је потребно знацати ко је тачно на чиму радио, када и сл.
- у сваком дистрибутиву се може знацати када се десила која измена тачно у коду митичног кода, коју је направио и сл.
- у свакој верзији ставки системе где се на:
  - **центрилизоване** (Subversion, SVN, CVS, ...)
  - **дистрибутивне** (git, Mercurial, ...)

### Центрилизовани VCS

- њихова имплементација је једноставнија, уколико је код са монитором измене мора да се на неком серверу који има улогу центризованог ревизора и да је све
- **недостаци:** уколико се осврнате без њега, подаци су несигури и нико ће морају да приступи

### Дистрибутивни VCS

- са центризованом ревизијом сваки клиент мора преносити код и уколико је измене које су се дешавале и сваки користитељ се сећа на коју било измене, а када је потребно да им осврнате измене монитору - ставка измене се не излази директно на центризованог ревизора, већ прво иде у локалну комиту ревизору коју он има код сеће
- тада се најзад оне синхронизују локалне комите ревизоре и центризованог ревизора
- напр. ако један клијент помеша своје измене које је направио, други клијент штафсе, у свом локалном ревизору штафса да избаче све оне измене са центризованог ревизора, када се они синхронизирају са локалним комитетом ствари које се налазе у центризованом ревизору



### Дистрибутивни VCS

- тије поште омака подесили име и мртв адресу (како би било једноставније пратење верзије ове измене)  
`git config --global user.name "Ivan Prezime"`  
`git config --global user.email mailadresa@nesto.com`

Потражавање за сваки ревизор, а ако се не саглави потражавају се бројни садржаји на ширејући извршитељске ревизоре.

- постоје два начина премештања локалног ревизоријума:
  - **инцидентализацијом твој (празног) ревизоријума** - користи `git init`, креирају се саглавни датотека .git унутар ревизоријума (је ово исклучиво)
  - **контактњакем локалног ревизора** са уџаветнијим сервером (јединственији је гитова смисаља као центризованог ревизора где су све)
- користи `git clone <путanja-do-repo> | <локални-директ-где-се-клони>` што искључује ревизор. Може се најавити релативна путница на коју ће се још ревизоријумију, али се ову путницу искључију

- `git status` - прати статус ревизоријума
- због првог креирања ревизор, а затим са `git clone` тај пунјатија која преносије код се све

- сада, имено додали неки фајл у тај репозиторију или се још неки од њих редизаписали као да је репозиторијус, већ можамо некако да назначимо Git-у да моче да прати промене у том фајлу

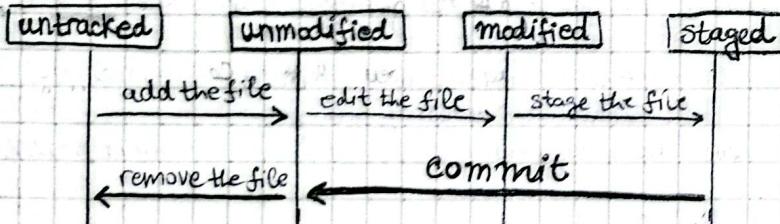
\*) **git add knziv.dat** → ако сачувати машину. пратите се да измените и тада дату.

- првиком **све измене** пребаро је првијенијију комаду (оглед се да се тада, или и да се дате које су измене)
- свака измена пребаро ће да буде слична заокружнута квадрат
- најскрути овај колекција изменских фајлова ће постати као **commit** (твоје који међу фаза изм. редит. превод.)

- **Commit** представља склађену чешћу измене коју можемо, "у једном заједно" на тај начин редизаписати
- некој што се разделио commit, да тајни фајлови који су били само у радном директоријуму претпоставију се сачртавати са локалним редизаписати, или и даље тиме описано нуже (тада та git-ови)
- Након commit-а сачиња се локални редизаписати или НАЈНОВИЈЕ СТАВАДОВА који су МЕЊАНИ и они што ће бити unmodified \*) **git commit -m "Neki opis"**

- Оправате фајла и сачиња git-у да смо обрисали тај фајл:
- git rm <naziv-fajla> брише из радног директоријума и staging area
- Иако ће неке измене које смо најдравили да бранији у штаче покреће commit који смо и саме сачиње се не садржавају (тада је радни директоријум и локални репозиторијус):

git reset ~~HEAD~~ <naziv-fajla>



- Када смо најдравили commit који је засад репозиторијус, сада хотимо да га сачртавамо са штам ункоректни репозиторијус.
- претпоставо на master фату и то је тако локални (тада, а хотимо да измене које смо на master фату у ункоректни репозиторијус)

\*) **git push -u origin master** → назив репозиторијус

да подјелите пратеће до локална master фату прати

master фату на ункоректни репозиторијус

и да следију тада најдогодјено губитак је само **git push**

- још један обј. фајлови који се налазе у директоријуму где је разделио на пројекат, или су изгубили само за тај, па бисмо то хијеси да изгубили.

**touch .gitignore**

- тада наведено тада фајла у директоријуму gitignore и тај фајл остаје да присуствује на тај локално, или да **git blame** не види
- Можемо ли реки да наведено до првијенијију сваки, или са једноставито за git НЕ ПОСТОЈИ уколико је засад gitignore фајла
- за штаче штачи је добро изгубаваји тај фајл чак и да смо слики који тада изгубију да се налази јасно (тада је првијена како наведено gitignore фајл)

- Уколико некијо брисате само из staging area (ама да јошади и даље кој на радном директоријуму, да не буде засад обрисана): ако не некијо да је бранио систим чланак (да не буде засад commit)

**git rm -cached <naziv-fajla>**

- Деците се некад да наведено што нотира нисамо хијеси (Нпр. направили само commit тада наведено засад је избацио) па хијеси што да избацио

1. сачињавамо што у staging area овај првијену за commit: **git add**.
2. **git commit --amend** → сачиња се единор који избацио да некамо избркну commita (ама некад да некамо - имено само escape и :w)

- како бисмо знати да сад тада засад commit тада да је засад commit исправљен што команду **git log** која омогућава да видимо све измене на штачи тада се наведено

- иакоје, можде је братаје да ји у unmodified сталац **git checkout -- <naziv-fajla>**

- пре сачињавате у сталац обј. commita ако из неког разлога наведено да видиш кој на радни директоријум избегавао пре commita **git checkout <hash>** → важи commit је засад засад предишњи

или кој не наведено избркну у радни директоријум

- ако искамо да се предавамо у статије ово commit-а да обришемо је ото што смо након тога обрадили:  
`git reset --hard <hash>`

- када радиш на пројекту иницијално имаш амтернештвите локалне гране који предаваш
- све што си до сада радиши било је инспиративно - иначи и неко иницијално сима на све што си неко изменио јесте тај грана
- чекомко у неком имениту хотимо да се раздвојимо од тога која измена која се делила на тој наш листин, иницијално надравши туђу грану која омогу да од ње именити измене да именити и да туђу грану креије да се раздвоје (то што је изменито на тајну грану, више није видљиво на тајну)
- такође, ишо ће више буди ради на пројекту - имените радиши на једној грани → ако раздвојиш теку функцијалност која не треба да имаје на то што се делила код осим да ли је отворио - идемо се раздвојити у један амтернештвни поток, а на крају ћемо све именити - и искл. идемо тајни видљиве измене

- на git-у је остовите грана око које крећемо **master (main)** и оне се гаве раздвојено
- развијамо коми и када си закрутили стапајши (merge) код компоненте са тајни гране и преносиши је да се делила грана и да само merge мораје да се именити и директан развој не ради на тајни гране, већ да се све што ради на енверзији, а искл. отварања си склопиши да то ради предаваш на лабити
- **HEAD** представљаје иказивач који показује на коју грану и на који извору у тој гранци се пренето налазиши
- сваки commit иницијално посматрајши као **тоб** извор
- креирајте тајне гране: `git branch <name-graner>`
- преведије који гране пренето именито: `git branch`
- пренеси да раздвојиш именито на тајни гранци: `git checkout <name-graner>`
- када хотимо измене са тајне гране да предавамо на **master** грану - да садржи гране:

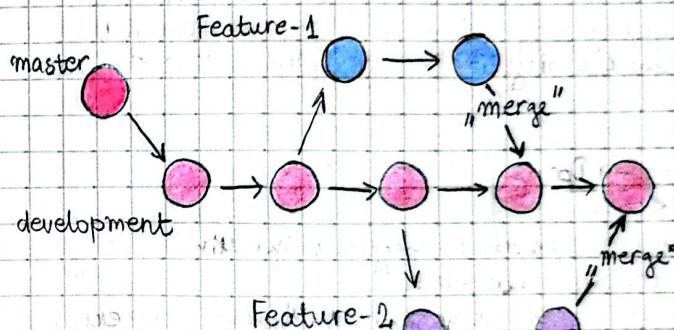
  1. именито се на ову грану на коју твоја годишња измене са



- друге гране (нпр. лабити)
2. `git merge <grana>` - са когом спадамо?
  - `git rebase <име гране>` - `git branch -d <име-гране>`
- Поред `git merge`, ишамо и `git rebase` који за разлику од тога не креира нови commit у пренетку стапајши грана, већ само искл. је commit-е са једне гране са другом граном
- именити симти у врху додатне гране измене са масивне граном
- **Предаваште локалних измена** да други види - када хотимо да измене са неке тајне гране предавамо на уџавету грану која те именити има тајни назив:

`git push -u origin <име-гране>` → имените ирате измене  
ујавите идете који имају  
име

осиму да се именити  
да не испадају сваки идри  
на уџавету раздвојијују  
изјавати да origin има <име-гране>



- креће се од остовите **master** гране и када хотимо да радиши на пројекту пренавишши на **development** грану која смущи да из ње гаве раздвојиш једнотичне функцијалности и на тајни гранци пренесиши само што смущају једнотичних feature-а и искл. именито неку стапајши верзију пројекта која је функцијална - тада бројиш стапајши developer на **master** - **master** - основна грана на којој су само стабилне верзије гранки
- **development** - смущи да из ње гаве гранамо тајне гране на руту којој имплементирају једнотичне функцијалности и да се склопију као чиме commit-е

- Може се десити проблем ако је неко изменчио локалнију улогу као тада ми (дату чисту свест) држеје особа из page које се локално и друга особа је што изменила, а кога је друга особа што поклонила јавио ју је **merge conflict**
- Прво, када је локалнија измена на централном репозиторијуму, од тога смештаји последњи пут синхронизовани са локалним, па јединствено што се **git pull** добућено ће измене са централним репозиторијумом да уради спајање постудијем измене на централном репозиторијуму и овако ће локално
- **МИ НЕ МОЖЕМО УРАДИТИ PUSH** без поглавља верзије централног репозиторијума (ј. док се не синхронизује)

- И овде, када уградимо **git pull** а неко је од последњих измена који су додружили са централним репозиторијумом, измените теку чисту коду (тју коју и мы, локално) и вратио, поклон, push-оваје ће час, ик овде када уградимо **git pull** - јавља се **merge conflict** што оштетише свако измену:

**merge не брине  
само додирај!**

```
<<<<< HEAD
  NSNEGY HEAD и == 8
  неки трећи текст } оти чији су они међуали
  [=====]
  неки други текст } оти чији је она друга особа међуали
>>>> некаснији бројки
```

- Јесоријски, јо се морте свако поклони, али се то никад неће дистрибуирати, па прво што морамо средити пре него што сад званично помешавамо наше измене

- морамо одлучити која од ње ће верзије (наше и тоја инф.)

→ јасно, треба средити садржај конфликтите да исклесне пре него што буде поклонија на удавлетвени репозиторијум.

- тако је тоја гравитација **git add**, која ће да наше промене репозиторијум и тако да је морамо одредити **push**

- када је неко **git pull** креирао промене - он није обvezan да са једном централним репозиторијумом, па је морамо осигурати **git remote add origin <адресарејзор/богатијум>** али лакше је само **git pull** **git clone**

- Постоји две точности за поклонче измене:

1. **git pull <име-или-адреса-репозиторијума>** - добуће измене и

автоматски проверава да ли су је са нашим изменама узео смо покривали у небу брзнету

2. - да прво **поговоримо** **git fetch <име-или-адреса-репозиторијума>** који преноси измене са удавлетвени репозиторијум

- али, думити смо да може што поговоримо и **git merge origin/master** - је изгубило што морамо да користимо на удавлетвени репозиторијум, па ставимо што смо још једном

(у овом случају ако се деси конфликт - тада дискутирајући свести ове гравитације **git merge**)

→ ову 2. точност је морамо брзој како дискутирајући да се што западава дешава

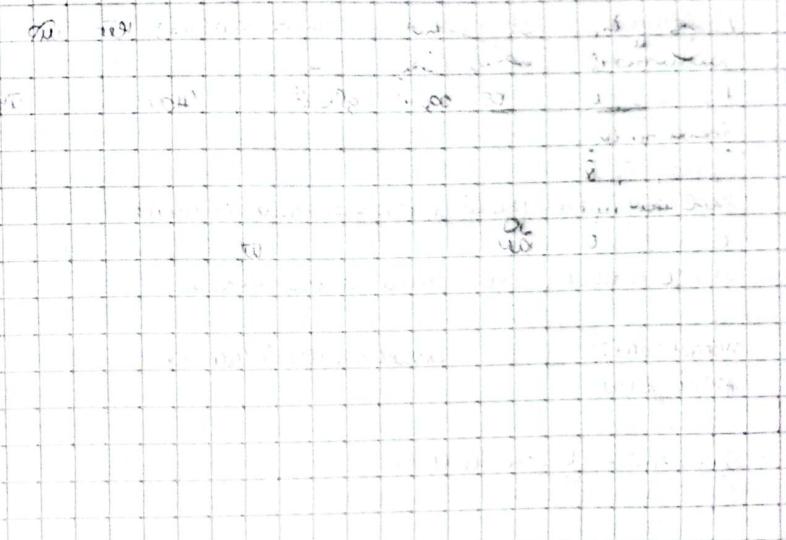
- **GitKraken** - монитор скренутим ако хотимо да видимо како се што изменила (који су драгљи и превртни, како су се симвари спајали, раздвајали...)

слично монитору **виделан** и обје:

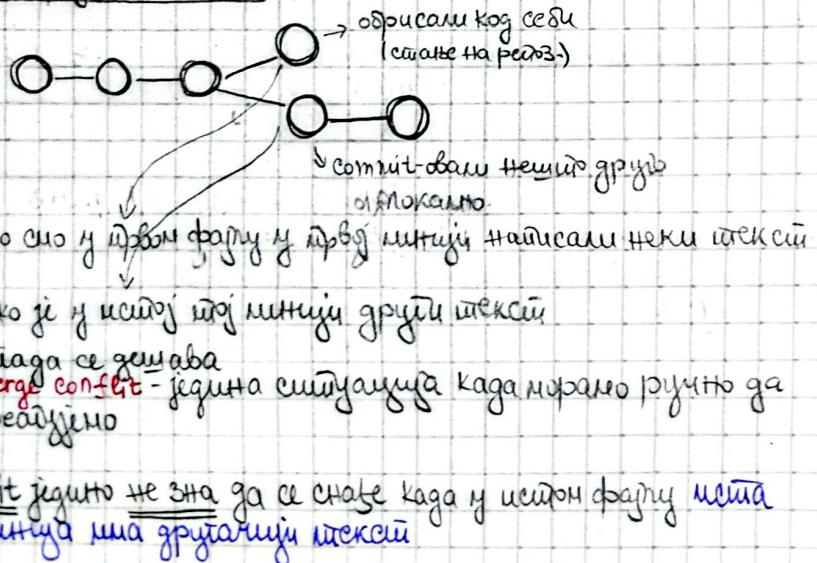
репозиторијум → Insights → Networks

- да видимо шта се са чиме садржио у ком третману, извесног држана,...

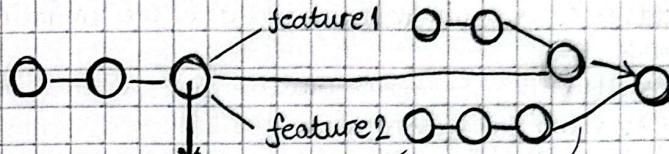
- кликом на точку ће видимо тајни commit, ко је шта је урадио штај.



- **експортовањето** - видијува свуѓа
- **импортовањето** - видијува само утилар и тоа што има
- **мешавица има receiver архитектура** - имаме компоненти за који имаше структуре и се вградија најава мешавица
- **репозиторијум** јај је ок да буде и јават и приватат  $\Rightarrow$  додало ѝ јај као колаборатор  $\star\star$
- **git clone link** - пренесија репозиторијум
- **git status** у терминалу видијо који је фаза тие имају измене а нису додади
- **git add:** у терминалу, додади све измене
- **git commit -m "prvi commit"** - само тоја видијува да се све те измене запишату
- **git push** - тоја чуваја измене
- **git pull** - осоми ве приказује штоја је радила прва осома
- **нивојти бек репозиторијума** зависимо као импорту commit-ови



- Кад хотимо да неко из штита ради на својј, а ми на својј суштински останемо



**main, master** - дати - где смо се раздвојили

- на master ѕемо само прву и последњу дату, осомо џе на **development** и **feature** датата

са горе дати текстама појма штоја се дешифаше и одржују

кој је развијао **feature2** и завршио касније, мора да реши конфликт

особа А

**git checkout - feature1**

особа В

**git checkout - feature2**

- **git checkout main** - нема филета В јер не имаја да креира, само да је џи

- **git merge feature1**

- кад урадимо **merge** - тоја видијува са удаветим развој

- **merge** - исклучују commit који сијаја измене од другији

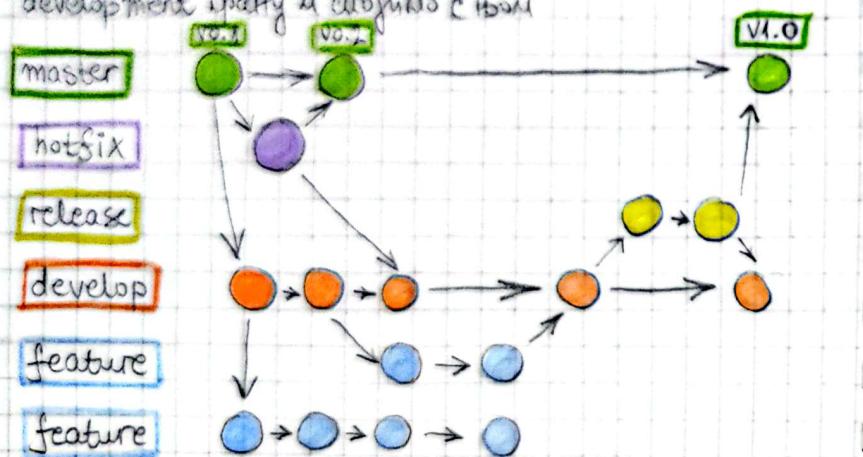
feature2

**git checkout main** -  
**git merge feature2**

тојје најчешћи конфликт:  
кој измене на првото дати  
суперзор ==  
кој измене на дати 2.

- Врши се дају комади који је git додат и одговарајући јеју комади за обновљавање
- користи се за развојне измене са ове складиште
  - git add
  - git push
  - git commit
- Потраживају се GitFlow метода
- Права је main или master дрвата - на њемо најављени иницијални commit у merge - где имају  
са read.me

- на development - свака нова функција је експлентна додат јеју и један иницијални feature
- feature - је функцијоналност - развојује се из development  
дрвата и радило је сваки штут (имаје други дрвасиду, ако је то захтевано) - када заврши штут, требајући се на development дрвту и спојити с новим



- hotfix - Када се деси неки баг на main дрвту, држи се решење и врати - иако је неповољно корисници
- release - безбедно за коришћење за неки развој (верзија)?
- Основна нам је main дрвта, develop и други облици feature дрвата

### Већије 3 (СНИМАК - Већије 25.10.2021.)

- рад са фалновима, серијализација - оствара за пројекат јго ће чинити лакши да имамо велику количину података који складиштимо у шири фалновима, а онда разделим појединачно, манипулатују, променљиви и сл.  
**(предмет који је директно везан за пројекат)**

#### Рад са фалновима

- **фалн** адресије који су подјелјени у складиштење на диску под обј. називом
- сваки фалн у својој основи адресије неки табуларни јединица којима имају објекатни значење
- основна разлика која ће бити да ће радити са **текстовима и бинарним фалновима**
- како сви шири фалнови записати у **бинарни формат**, текстовни фалнови са собом носе нека правила којима шири написано ће бити то који су записати
- **Бинарни фалнови** су општији обј. и потпуно записати било шта док је то највећи објекат који ће бити у складишту
- у до. у темо коришћени пакете `os, io, io/ioutil, bufio`
- **Креирање и прислања:** аутоматски и отварајују фалн и убрзјено извешавају на њега

```
file, err := os.Create("nekifajl")
if err != nil {
    panic(err)
}
err := os.Remove("fajl")
if err != nil {
    panic(err)
}
```

Новозадато ће изгашати и назив фална; објекат је увек само назив без експлозивних или апсолутних - иако се огледи иза прислања

- како диско променили неке што о фалну (назив, структура, подлога измена, дозволе шта) да се досадаште фалн `Stat()` која као подразумевају да имају променљиву `fileInfo` која ће сада имати шта.

```
fileInfo, err := os.Stat("nekifajl")
if err != nil {
    panic(err)
}
fmt.Println(fileInfo.X())
```

Name
Size
Node
ModTime
ISDir

- кроз добављање ширих метода. Може се проверити да ли фалн је корикто се осим  **Rename**, она може бити проконвентна и да ли је **фалн** једијачији таје промене
- затим враћајући прверу да ли је дрејка шта `IsNotExist` - ако јеси фалн НЕ ПОСТОЈИ, у складишту посматрали смо је додуло до дрејке неког другог ћелија

```
if _, err := os.Stat(fileName); err != nil {
    if os.IsNotExist(err) {
        return False
    }
}
return True
```

- **Променетвајање (Променетвајање)** - смичто као код креирања или прислања, само сада кориснији фалн  `Rename ("fajl", "fajl")`
  1. ари. је оригинални назив фална заједно са његовим ширијством
  2. ари. је ширијства (остави исима или је неимају у складишту) и тоб. назив фална

- како диско радили било какве операције са садржајем фална, то ће то бити је да га ће отворити - иако као креираје, прислање, али је фалн `Open ("fajl")` - добијамо неки тековнизвештај на фалн и када завршимо рад са фалном, **записујмо га file.Close()**  
Сада је **корисније** `defer` фалн да не заборавимо записати

Дојавлен је шири отварају фалн у **read-only** речник, што често тије добијато, па прислају општију фалн `OpenFile ("fajl", os.O_RDONLY, 0666)`

изјашавају и назив фална  
речник у ком ће фалн отварати  
изјашавају и назив фална

изјашавају и назив фална  
речник у ком ће фалн отварати  
изјашавају и назив фална

- ФАЙЛСИКЛЕВАЊЕ**
- **рентници** које фајл може имати (модусе је највеће висине између којих је отвораштво | OR)
    - O\_RDONLY - отвора се read-only
    - O\_WRONLY - отвора се write-only
    - O\_RDWR - отвора се read-write
    - O\_APPEND - писање додавањем новог садржаја на крај
    - O\_CREATE - креирање новог фајла чиме лик вет не постоји
    - O\_EXCL - у комбинацији са O\_CREAT, захтева да фајл не постоји
    - O\_SYNC - отвора се со синхронни I/O
    - O\_TRUNC - старији фајл при отварању ако је Moifte

### - дозволе

- као 1. држач темо највећи нулу
- 2. држач предлог које дозволе написање фајла, највишији
- 3. држач предлог која прави користији који су први. Другите
- 4. држач је за све остале користије

аконе моте имена дозволу за извршавате (ако је фајл идентичан)

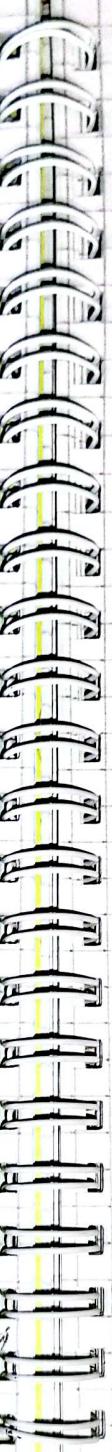
пишате и писате

- ако највећи 1 за виткија од 3 држача → дозвола за извршавате
  - 2 → дозвола за писате; 4 → дозвола за пишате
- и садиратељи свих "осталих" вредности садијмо све остале дозволе

- ако идемо **копирате** садржај неког фајла, прво је Moifređko отварате што фајла и креирајте нови фајл у који идемо да копијамо садржај искривети, а иако што садо извешено фајл **Copy ("новијај", "originalнијај")** → показивање оригиналног фајла показивање на нови фајл који садо прешај. добији и као рез. добијамо који је близија прешаји и све тајну друштву до које је идло доји

**err = newfile.Sync()** → како оно се осигуруји да је садржај записат на диск у моменту када то нишће даји  
(показивање на нови фајлом или када вратимо неке измене)

- Када идемо отворати фајл и не очекује се да иницијијализује садржај у Меморију вет тир само теки зео, бројимо извршитијате још путу фајл **Seek** која врши померање курсора на извршбу која најма оги.



### Seek (offset, whence)

за којимо објашњујате се инструкције померање

#### ПИСАЊЕ

- структуре File у суштини имплементирају један интерфејс Writer који је задатаким свим структурима које извршавају било какво **писање** и све имају **Write()** методу. Методу Write() који је задатак да пресети влије близија јужне у задату стручку (у облику даска)

Креирали смо Slice који bytesSlice := []byte("Bytes!\n")  
bytesWritten, err := file.Write(bytesSlice)  
if err != nil {  
 log.Fatal(err)  
}  
log.Printf("Wrote %d bytes.\n", bytesWritten)

- фајл која се идентичи моте искоришћенији је из io/ioutil пакета

err := ioutil.Writefile(fileName, []byte("Hi!\n"), 0666);  
if err != nil {  
 log.Fatal(err)  
}

написају се свијетла близија који идемо садијамо  
зато што је дано моте фајл која моте свашија  
да буди у сано једну имену!

јединије се чуо садржај фајла, чема извештај да је текши вет прешају  
обе обе објекти за нека **ЛАКША УПИСИВАЊА** која не захтевају  
што идемо обраде, извршијатија и са, па у структуром који су  
иако захтевију идемо **стручку** Writer (који као прешај.  
што идемо Writer итерфејс, али идемо када идемо стручку  
која је билементијираја тај итерфејс)

→ од тајких структура моте се креирају штој bufferedWriter који је  
заштитију да врши бројијате садржају који се чујије у фајл  
и вршије врши само уписивате  
(заштитије што бројија када се креира је **4096B** што моте пропишији)

Writer чујије садржају само у басфер, па ако што идемо да  
идемо и фајл чујији садржај басфера, извештај **Flush()** која  
одија вршије да ће садржај из басфера бити чујијан на диск оди даска  
Вршији речија да се промиком чујије у басфер **Не сме**

**Прекорачниот метод** (кога буфер јасните и) вр. од тип `Човечко`  
все се дава дигитален, без се направи или извади `Flush()` или  
извади `Reset()`

Човечкиот содржински буфер ќе се објаснати фазата

- `Size()` - пребрзувач на без се буфера
- `Buffered()` - колико је буфера покриено
- `Available()` - колико има свободних буфера у буфера

```
bufferedWriter := bufio.NewWriter(file) → креираше slice тј. пресетујќи
bytesWritten, err := bufferedWriter.Write ([]byte {65, 66, 67})
if err != nil {
    log.Fatal (err)
}
log.Printf ("Bytes written: %d\n", bytesWritten)
```

```
bytesWritten, err := bufferedWriter.WriteString ("Buffered string\n")
unflushedBufferSize := bufferedWriter.Buffered () → колико је буферт
log.Printf ("Bytes buffered: %d\n", unflushedBufferSize)
```

```
bytesAvailable := bufferedWriter.Available () → количина буфера што можат
if err != nil {
    log.Fatal (err)
}
```

```
log.Printf ("Available buffer: %d\n", bytesAvailable)
```

```
bufferedWriter.Flush () → буфер записан содржини из буфера
на файл
bufferedWriter.Reset (bufferedWriter) → кога си било напишал содржини из буфера, очисти без буфера
```

```
{ bufferedWriter = bufio.NewWriterSize (file, 8000)
```

ако не имаш буфер по подразумјевано defaut - то било буфер на  
файл имаш буфер по подразумјевано

Без буфера у буферирана

## ЧИТАЊЕ

- спомињамо како ког мисава, мисли изпленети чардак Reader интерфейс  
кој има метод `Read()` на файл је напомена корисникот употребува тога  
файл

```
bytes := make ([]byte, 2)
_, err = file.Read (bytes)
if err != nil {
    panic (err)
}
```

Учењето је содржински файл без огуштина slice-a, чимаше се само  
ако ништо буфера колико slice може пресечат у така напомена, а

```
_, err = io.ReadAll (file, bytes) → декодирај файл кој се напоменети  
објект се напоменети читав slice је декодиран доколку ѓеше да решите  
що содржини који се напоменети из файла која дешти беше или  
јединак огуштина slice-a
```

```
_, err = io.ReadAtLeast (file, bytes, 1) → важи испоменети файл преша  
прешети, само објект кој што прешети наведене многу други буфери  
кој напоменети прешети
```

Нпр. ако је slice огуштина 5, а декоди у само шина и багај - така проблем  
кој са прешети методот, само је напоменети исклучително да файл напоменети  
напоменети бара 1 багј кој се напоменети друг. што дешти да решите

- али ако не напоменети метод така бр. која је файл кој се напоменети:  
`bytes, err := ioutil.ReadAll (file)` → учитава цел файл, кој  
напоменети бр. објект кој се напоменети из файла

- шакоје, посетији и файл ReadFile кој је јединак разлика што шакоје  
се дешти не напоменети доказувач на файл, без огуштина файл:

```
bytes, err := ioutil.ReadFile ("folder/file")
if err != nil {
    panic (err)
}
fmt.Println (bytes)
```

- ако шакоје и шакоје у шакоје захтевни  
обрад е шакоје структуру Reader кој ради шакоје било каквих  
метод-чардак који шакојен чардак Reader шакоје шакоје

- Read() - ако шакоје да полегато содржини дрвих и буфера  
тога файл поседује

- једини је начин да се курсор сваким митњачем или птицелем, помера, или ког peek-a што НЕЋЕ ПОДАТИ (у случају као Read без података курсора)
  - **ReadByte()** - митњаче само једнот барграђеног у односу на претходну позицију), а ако некада не може да прошире чији то (дошли до краја или је фарм пресат) - добијено чешће неки
  - ког некада **ReadBytes()** и **ReadString()** задаје се неки дефинисанији и не ће вратити барграђену фразу пре дефинисаног заједно са дефинисаним
- изборни бр. барграви      изборни бр. ширите
- Уколико градите са текстуалним фармовима, скупинама **Scanner** је подобра за што и она чији фарм может да имаје
  - может може тј. дели сваки ред или реч у фарму, а можемо и написати најчешћи иницијализатору што ћемо да пресадимо может (иницијализатори split фарм који се простирује Scannerу) и онда се чији фарм има и может а које смо ми дефинисали
  - Не може се гарантирати да је сваки карактер један барграђен при UTF Encodingу карактер може имати 1, 2, 3, 4 барграђена (што је користимо лаптчица, бројеви, знакови - чудавином чијада је један барграђен, док тј. кинеска слова имају 4)

## Серијализација

- серијализација је користна када је потребно изградити текуће адекватне структуре (текући компоненти јади)
- **серијализација** се бави што је како најчешћи текући формат, који предаје адекватнији формат што адекватно може исклопити, а текући што је и десеријализован одн. реконструисан на оствару што сваје алигаторашите формате
- што формате за серијализацију пострема поделим на:
  - текстуалне - json, xml, csv, yaml, ...
  - бинарске - BSON, MessagePack, protobuf,

- Чудавином чијимо обратните најдат формат, већ користимо врло често је стандард и што најбоље задовољава доступност захтеве
- У доду имамо шакеј encoding где су числа дешифрована осима једног шакеј који се бави серијализацијом и десеријализацијом тј. у json, csv, xml итд.
- За бинарне формате имамо још шакеје **Binary**, где...
- **encoding / Binary шакеј** - Користим је у једноставнијим случајима мада имамо број филсте дужине или имамо низ, где који садржи само број филсте дужине тј. др. симулација тада ће баш односити, па њему заснованим др. шакеј
- Овој шакеју тада посматре да постукује структуру што ћемо сваку од тих броја користити као unsigned int дужине 16, 32, 64 и посматре да одредимо што структуру
- имамо прописане LittleEndian и BigEndian како бисмо очували редослед барграђених

`b := [3Byte{1,2,3,4}]`

`val1 := binary.LittleEndian.Uint32(B) → исписује у бинарном облику 4 3 2 1`

`val2 := binary.BigEndian.Uint32(B) → исписује у бинарном облику 1 2 8 4`

`binary.LittleEndian.PutUint32(B, val2) → исписује 4 3 2 1`

- па сваки почиње постепено редом записивати структуре, односно по реду

шакеју сваки шакеј чији објекат је **Read** и **Write**:

`tip: err = Binary.Write(file, binary.LittleEndian, vertex)`

пред. чији креирају  
у који формат ћемо  
послати

`newv := 4Vertex{3} → креирају  
данашњи на Vertex`

`err = Binary.Read(file, binary.LittleEndian, newv)`

предвиђено је да је формат као што је тај из једног формата  
погодан променити нову на коју показује и  
која је једна Vertex{3}

## fmt.Println(\*newv)

- **encoding/gob пакет** - пакет који нам помага да ради encoding и decoding за било шта што је могуће у gob (само се не може encode/decode дате и катали)
- два врата серијализације претпостављају имплементација само у gob
- сама информација оим података обавља се што су структуре Encoder и Decoder који се креирају на ова начин:

```
student := Student {"Petar Petrović", 33}
encoder := gob.NewEncoder (file) → претварање структуре
err=encoder.Encode (student)   → показвање на
if err!=nil {                   новодобијени фајл
    fmt.Println (err)           структурно садржај
}                                и серијализација та у фајл
```

→ ако нямамо да промитамо садржај тога фајла, морамо радији декодирање на сличан начин (добраја вредност која је сама декодују)

```
decoder := gob.NewDecoder (file)
var studentRead = new(Student) → креирање новог
file.Seek(0,0)                показвивања на аудијентик
for { → докуметавамо да ћемо имати све податке који досидоје
    err=decoder.Decode(&studentRead)
    if err!=nil { → иако све је још нема решење, огнт
        break
    }                                до краја фајла
}
fmt.Println (*studentRead)
}
```

- ово губитак радијо изводи методе Decode којеји и треба да се изјављују на тај који нямамо да имплементирамо

- пример које структуре се могу декодирати у тај (тје нутно тај да буде скроз једне структуре)  
- Морајући тај да је једна која не морајући да имплементирају

- шакоје, штојући могућност да ми само управљамо имплементацијом серијализације тај како ће ово структуру изгледати у бинарном формату
- шо поистовећу уградим шако што ће наша структуре имплементирати шта имплементирају њене структуре gobEncoder и gobDecoder и имати тоју же методу

gobEncode () (Error, error) и gobDecode ([]byte) error  
→ иако њих имплементирају, жеће преузећи контролу првих пруга структуре или декодирања тај који су што има

- у том случају шта имплементирају тешко није релевантна, већ тај који смо ми направили и она се имплементише што има
- сакаш методом Encode и Decode обе методе даваш дајују изјавати
- наша имплементација је користна када тај. шако структуре тај су нека што има exported, а нека unexported јер ће тај да бричим серијализацију (кодирање) само exported што, а unexported што ће затемарити као public
- exported што - тај ће бити најбољу челије апликације, моте ћи се присујити и из некој другој структуре
- unexported што - тај ће који се моте присујити само у оквиру тај структуре

- шакет предаји текућу колекцију до фајлова који су ови у један файл и шо моте оних користи када премају десеријализација њенога модуларизованог или извођених текући. Битнији ће бити да ногди користиши и други користиши записивање кода за потребну искористивост

- ћако и за пројекат:  
ако тај. поте ће задесеријализовати други утицати фајловима - не сме се изјављати садржај челеји фајла, већ се морамо премишићи имати да други прачуја (не моте readFile или ReadFull)

- рад са фајловима

- **серіалізація** - шано подаємо са корінна архітектура  
серіалізації, одн. об'єкти у пам'яті розміщені

- да бисмо тає об'єкт самі вали на диску, морем бисмо її  
десеріалізувати у тиз байтів

- http закінч і обов'язок - некі бінарні змінні які претендують  
поділити

- нехто да тає об'єкт десеріалізувати у тиз байтів да бисмо  
можли що да складши тає некі мініфайди

• **текстуальні фајлови** - контейнери з текстом фајлови та єдині  
текстуальні

• **бінарні фајлови** - 90% фајлови які юмо користими є у  
бінарні фајлови

- **ієрархія (permits ~ дозволе)** - ко ѹшаємо да раху у тих  
шампінгах

- або друга ієрархія:

- власник
- приватна група користника
- осіталі користники

- скла од тих садом ієрархії тає каліте да ли неко може да  
чита, змінить чи зберіга

- ко се смайдж власником (да є найдавно фајл ако некі працюю  
коєма бінарну дату)

- користник що користує фајл раху шампіра се да є найдавно  
фајл

read-only

write-only

read-write

append

create

} тиє рентні користимо

- ако тиє креира фајл, дівоте та креираши ѹа чиниши

- ієрархії се збільшу є обзор єдно приміжко креира фајл  
4? - за приватну групу користника

чита - 4, змінить - 2, зберіга - 1  
да чита чи да змінить - 6 (4+2)

- до функції отише таєко да не може ѹашим променів єрд  
паралелізма

- ако немоє create тає ѹачи паралелізар (нр. 0644) єс  
занадташи ѹїша єдно да змінить

- **panic** - діректива зберігавати програма  
що єдно єе испочини из сітка єзовіва

- да є користуєт єдни codin за серіалізацію, а други за  
десеріалізацію → даємо бисмо тає брюшту  
щонеє є за колико байтів

file.Seek ( ↓ , ↑ )

у єдиному на послідовність фајла

1 - є єдиному на шретчі єдно тає се наш показівач налази  
2 - є єдиному на краї фајла

- **бінарні формати** є компактні - на обрітній компактні  
підтримка заужина зажиткою наше програма, док тає шкідливі  
для єдно компактні формати за результатом єдно більше  
число

- example.go є ренчте єдно задані  
(єдно обробіти єдно єдно пресіх єдно)

найактивне bool

- юз чиншем єдно єдно, чверк єдно єдно єдно

- string єдно дроменів єдно

код серіалізації little endian, ота єдно little endian  
що єдно десеріалізації