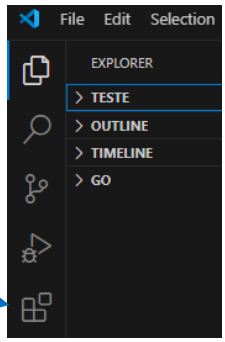
ATIVIDADE DIRIGIDA 1 de CSC-27/CE-288

CTA - ITA - IEC

Prof Hirata e Prof Juliana

Objetivo: Ambientar-se com Golang usando VSCode e simular processos rodando e trocando mensagens.

TAREFA 1: Ambientar-se com Golang usando VSCode com Hello World.



- Prepare o ambiente:
  - Instale o Go localmente.
    - ✓ https://go.dev/doc/install
  - Instale o VSCode (provavelmente você já tem)
    - ✓ Se desejar, atualize em *Help* > *Check for Updates*
  - Habilite a extensão do VSCode para Go

- Inicialmente, vamos fazer um código único (sem *packages*)

  - Crie a pasta `teste` para guardar os seus códigos em Go.
    - ✓ `%HOMEPATH%` será aqui usado para indicar o diretório escolhido (onde você criou a pasta `teste`.
  - No VSCode, abra a pasta `teste`.
  - Crie a pasta `myhello` para fazermos o nosso primeiro projeto.
    - ✓ Agora vamos ficar apenas na pasta `myhello`.
  - Crie o arquivo `hello.go` abaixo e salve.

```go
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello World")
}
```

Ao escrever `package`, veja que VSCode já sugere possíveis packages. Usamos `main` aqui pois é o código principal.

Ao escrever `fmt`, veja que VSCode já escreve para você o `import` do `fmt` (caso ele não exista ainda).

  - Abra um terminal na pasta `myhello` mesmo:
    - ✓ Compile e rode usando: `go run hello.go`
    - ✓ Outra opção:
      - Compile: `go build hello.go`
      - Depois rode o `exe`: `.\hello`
      - Em geral, vamos usar essa opção, pois trabalharemos com diferentes terminais (cada um com um processo diferente rodando).

Em alguns ambientes (ex: Windows PowerShell ou no terminal do VSCode), precisa usar esse .\

- Agora vamos trabalhar com ***packages***

  - **Lembre sempre de salvar todos os arquivos**, senão aparecem uns erros como se não tivéssemos feito o que já fizemos.
  - Go organiza os pacotes através do arquivo `go.mod`.
    - ✓ `mod` (*module*) é uma unidade de distribuição (com as dependências do projeto) e versionamento (com a versão do Go usada) do software.
    - ✓ Através do terminal, na pasta `myhello`, crie o arquivo `go.mod`:
      - `go mod init example/myhello`
    - ✓ Abra o arquivo `go.mod` só para ver como é simples.
      - Atenção: `go.mod` deve sempre ficar na pasta principal do projeto (ou seja, a pasta `myhello`)

Aqui é o nome do seu `module`. Usa-se o endereço/site onde está o seu projeto. Como está local, usamos um nome qualquer. No futuro você pode usar o endereço do github.

Para ficar mais didático, usaremos aqui o mesmo nome da nossa pasta.

  - Na pasta `myhello`, crie a pasta `csc27` (esse é a nossa nova `package`)
    - ✓ Dentro dela crie o arquivo `specialHello.go` abaixo.

```go
package csc27

func Welcome() string {
	return "Welcome to CSC-27!!"
}
```

Letra maiúscula no nome da função para ela ser vista externamente.

Tipo de retorno da função.

  - Complete a `main` do arquivo `hello.go` com uma chamada à função `Welcome`:

```go
	fmt.Println(csc27.Welcome())
```

Usamos o nome da `package` onde a função está. O VSCode vai criar automaticamente o `import` de: `"example/myhello/csc27"`

  - Finalmente, rode `hello.go` para ver se funciona.

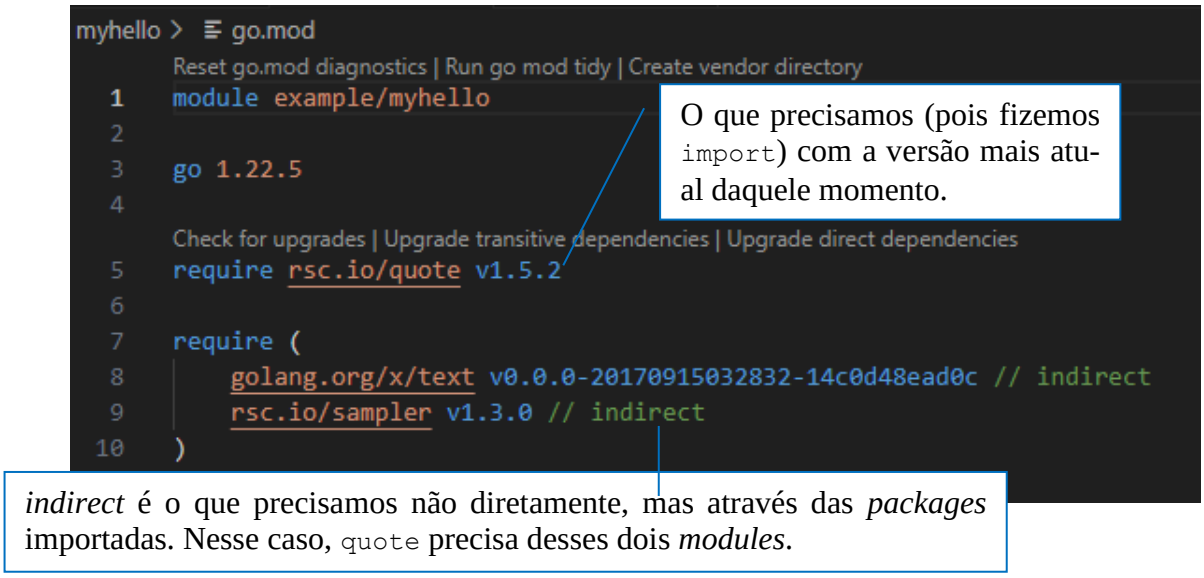
- Agora vamos trabalhar com ***external packages*** (prontas na Internet)

  - Você pode procurar *packages* prontas no site: `pkg.go.dev`
  - Vamos usar *package* `quote` que é disponibilizada no module `rsc.io/quote`
    - ✓ https://pkg.go.dev/search?q=quote
    - ✓ Documentação em: https://pkg.go.dev/rsc.io/quote/v4
  - Em `hello.go`:
    - ✓ Inclua no `import`: `"rsc.io/quote"`
    - ✓ Inclua na `main`: `fmt.Println(quote.Go())`
      - Essa função Go imprime na tela uma mensagem que é o lema de Go: "*Don't communicate by sharing memory, share memory by*

communicating".

- No terminal, rode: `go mod tidy`

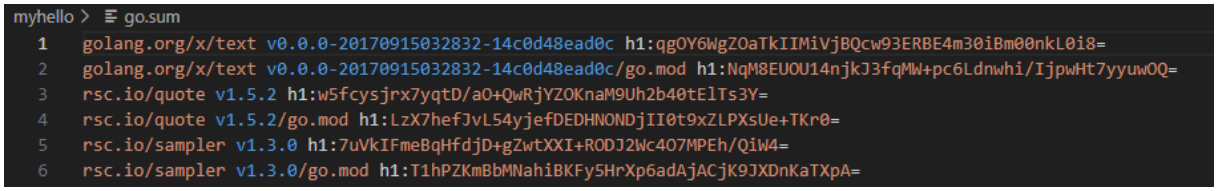
  - ✓ Isso atualiza `go.mod` com todos os *modules* necessários para o seu projeto: adiciona/mantém os usados (ex: o `quote` que fizemos `import`) e retira os não usados. Veja como ficou o `go.mod`:



O que precisamos (pois fizemos `import`) com a versão mais atual daquele momento.

indirect é o que precisamos não diretamente, mas através das *packages* importadas. Nesse caso, `quote` precisa desses dois *modules*.

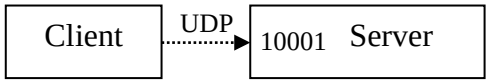
  - ✓ Isso também cria/atualiza o arquivo `go.sum`
    - `go.sum` mantém o *checksum* para não precisar fazer download dos *modules* toda vez que rodarmos o projeto. Ele usa o cache localizado no diretório: `$GOPATH/pkg/mod`
      - Obs: Para saber o seu `GOPATH`, rodar: `go env GOPATH`



- Finalmente, rode `hello.go` para ver se funciona.

TAREFA 2: Simular processos rodando e trocando mensagens.

DICA 1: Compreenda o funcionamento do programa cliente-servidor usando conexão UDP. Obs: Código abaixo retirado de:
https://varshneyabhi.wordpress.com/2014/12/23/simple-udp-clientserver-in-golang/



A ideia aqui é o `Client` enviar indefinidamente valores inteiros (em ordem crescente) para o `Server`.

Obs: Veja que no código a porta 10001 é fixa para o servidor “escutar”.

Server.go

```go
package main

import (
    "fmt"
    "net"
    "os"
)

/* A Simple function to verify error */
func CheckError(err error) {
    if err != nil {
        fmt.Println("Error: ", err)
        os.Exit(0)
    }
}

func main() {
    /* Lets prepare an address at any address at port 10001*/
    ServerAddr, err := net.ResolveUDPAddr("udp", ":10001")
    CheckError(err)

    /* Now listen at the selected port */
    ServerConn, err := net.ListenUDP("udp", ServerAddr)
    CheckError(err)
    defer ServerConn.Close()

    buf := make([]byte, 1024)

    for {
        n, addr, err := ServerConn.ReadFromUDP(buf)
        fmt.Println("Received ", string(buf[0:n]), " from ", addr)

        if err != nil {
            fmt.Println("Error: ", err)
        }
    }
}
```

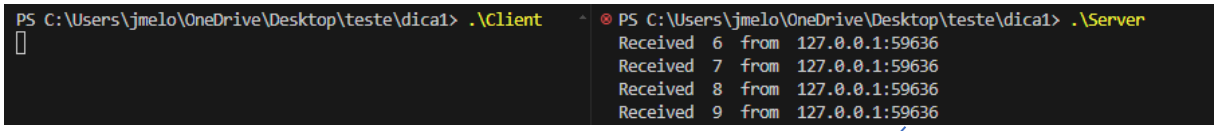
Client.go

```go
package main

import (
    "fmt"
    "net"
    "strconv"
    "time"
)
```

```go
func CheckError(err error) {
    if err != nil {
        fmt.Println("Error: ", err)
    }
}
func main() {
    ServerAddr, err := net.ResolveUDPAddr("udp", "127.0.0.1:10001")
    CheckError(err)

    LocalAddr, err := net.ResolveUDPAddr("udp", "127.0.0.1:0")
    CheckError(err)

    Conn, err := net.DialUDP("udp", LocalAddr, ServerAddr)
    CheckError(err)

    defer Conn.Close()
    i := 0
    for {
        msg := strconv.Itoa(i)
        i++
        buf := []byte(msg)
        _, err := Conn.Write(buf)
        if err != nil {
            fmt.Println(msg, err)
        }
        time.Sleep(time.Second * 1)
    }
}
```

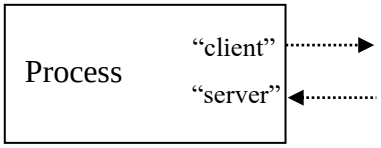
Usamos *localhost* 127.0.0.1 no endereço

- Ajuste o ambiente:
  - Crie a pasta `dica1` na nossa pasta `teste`.
  - Crie os arquivos `Client.go` e `Server.go`.
  - Compile ambos:
    - ✓ `go build Server.go`
    - ✓ `go build Client.go`
- Teste o sistema assim:
  - Terminal 1: `.\Client`
  - Terminal 2: `.\Server`
- Resultado esperado
  - `Client` não imprime nada.
  - `Server` imprime os números recebidos.
    - ✓ Como o `Client` foi iniciado antes no terminal, perceba que `Server` começou a receber em 6 (nesse exemplo), perdendo então os primeiros números enviados:



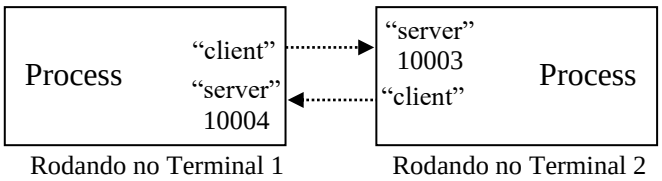
No código do cliente, a porta foi definida como 0. Isso faz com que o sistema escolha uma porta adequada, nesse caso 59636. Tente definir a porta (diferente de 0, por exemplo 3) e veja que aparece corretamente aqui.

DICA 2: Junte num só arquivo `Process.go` o código de cliente e servidor. Assim o processo terá capacidade de enviar mensagens (pelo cliente) e receber mensagens (pelo servidor).



Desejamos ter vários processos se comunicando! Então precisamos receber como parâmetro (no momento de executar o programa) a porta do servidor do processo em questão (primeiro valor na chamada abaixo) e as portas dos servidores dos demais processos (demais valores na chamada abaixo). Exemplo com dois processos:

- Terminal 1: `.\Process :10004 :10003`
- Terminal 2: `.\Process :10003 :10004`



A ideia aqui é um `Process` enviar indefinidamente valores inteiros (em ordem crescente) para todos os demais `Process`.

Obs: O endereço 127.0.0.1 pode continuar fixo no código. Caso queira testar em diferentes máquinas, isso deve ser configurável.

Abaixo consta uma sugestão para o código `Process.go`. Analise!

```go
package main

import (
    "fmt"
    "net"
    "os"
    "strconv"
    "time"
)

// Variáveis globais interessantes para o processo
var err string
var myPort string          //porta do meu servidor
var nServers int           //qtde de outros processos
var CliConn []*net.UDPConn //vetor com conexões para os servidores
// dos outros processos
var ServConn *net.UDPConn //conexão do meu servidor (onde recebo
//mensagens dos outros processos)
```

```go
func CheckError(err error) {
    if err != nil {
        fmt.Println("Error: ", err)
        os.Exit(0)
    }
}

func PrintError(err error) {
    if err != nil {
        fmt.Println("Error: ", err)
    }
}

func doServerJob() {
    buf := make([]byte, 1024)
    //Loop infinito mesmo
    for {
        // Ler (uma vez somente) da conexão UDP a mensagem
        n, addr, err := ServConn.ReadFromUDP(buf)
        PrintError(err)
        // Escrever na tela a msg recebida (indicando o
        // endereço de quem enviou)
        fmt.Println("Received ", string(buf[0:n]), " from ", addr)
    }
}

func doClientJob(otherProcess int, i int) {
    // Enviar uma mensagem (com valor i) para o servidor do processo otherSer-
ver.
    msg := strconv.Itoa(i)
    i++
    buf := []byte(msg)
    _, err := CliConn[otherProcess].Write(buf)
    PrintError(err)
}

func initConnections() {
    myPort = os.Args[1]
    nServers = len(os.Args) - 2
    /*Esse 2 tira o nome (no caso Process) e tira a primeira porta (que é a
minha). As demais portas são dos outros processos*/
    CliConn = make([]*net.UDPConn, nServers)

    /*Outros códigos para deixar ok a conexão do meu servidor (onde recebo
msgs). O processo já deve ficar habilitado a receber msgs.*/

    ServerAddr, err := net.ResolveUDPAddr("udp", "127.0.0.1"+myPort)
    CheckError(err)
    ServConn, err = net.ListenUDP("udp", ServerAddr)
    CheckError(err)
```

```
    /*Outros códigos para deixar ok a minha conexão com cada servidor dos ou-
tros processos. Colocar tais conexões no vetor CliConn.*/
    for s := 0; s < nServers; s++ {
        ServerAddr, err := net.ResolveUDPAddr("udp", "127.0.0.1"+os.Args[2+s])
        CheckError(err)

        /*Aqui não foi definido o endereço do cliente.
        Usando nil, o próprio sistema escolhe.  */
        Conn, err := net.DialUDP("udp", nil, ServerAddr)
        CliConn[s] = Conn
        CheckError(err)
    }
}

func main() {

    initConnections()

    /*O fechamento de conexões deve ficar aqui, assim só fecha
   conexão quando a main morrer (usando defer)*/
    defer ServConn.Close()
    for i := 0; i < nServers; i++ {
        defer CliConn[i].Close()
    }

    /*Todo Process fará a mesma coisa: ficar ouvindo mensagens e mandar infi-
nitos i's para os outros processos*/
    go doServerJob()
    i := 0
    for {
        for j := 0; j < nServers; j++ {
            go doClientJob(j, i)
        }
        // Espera um pouco
        time.Sleep(time.Second * 1)
        i++
    }
}
```

Temos uma *goroutine* só para o trabalho do 'server'.

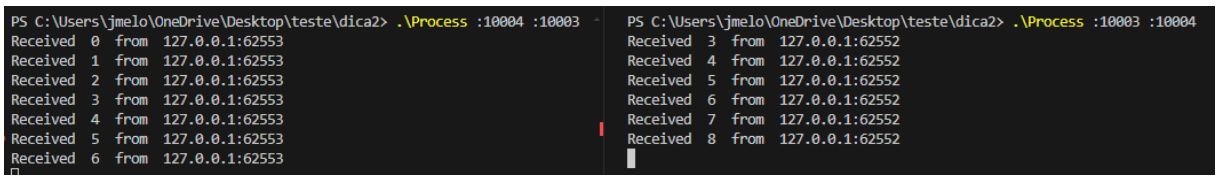
Temos uma *goroutine* para cada trabalho do 'client'.

Tudo rodando de modo concorrente!

- Teste o sistema assim:
  - Terminal 1: .\Process :10004 :10003
  - Terminal 2: .\Process :10003 :10004

Lembre sempre de compilar antes!

- Resultado esperado:
  - Perceba que o segundo processo (iniciado depois) não imprime os valores iniciais (no caso, após 3). Isso porque ele começou depois e perdeu o que o outro processo o enviou.



- Depois teste o sistema assim:
  - Terminal 1: `Process :10002 :10003 :10004`
  - Terminal 2: `Process :10003 :10002 :10004`
  - Terminal 3: `Process :10004 :10002 :10003`

Da forma implementada, a primeira porta é a do processo corrente e as demais portas (em qualquer ordem) são dos outros processos.

DICA 3: Queremos "controlar" cada processo de modo independente para ele fazer uma ação que desejarmos. Assim eles não terão o comportamento padrão de antes e poderemos simular diferentes situações. Vamos "controlar" o processo através do envio de texto pela janela de comando (do terminal em que o processo roda).

Para isso, adicione a função abaixo ao código do processo. Lembre-se de importar a biblioteca "bufio".

```
func readInput(ch chan string) {
    // Rotina que "escuta" o stdin
    reader := bufio.NewReader(os.Stdin)
    for {
        text, _, _ := reader.ReadLine()
        ch <- string(text)
    }
}
```

Substitua o comportamento anterior do processo (parte da 'main' após o fechamento das conexões) pelo seguinte:

```
    ch := make(chan string) //canal que guarda itens lidos do teclado
    go readInput(ch)        //chamar rotina que "escuta" o teclado

    go doServerJob()
    for {
        // Verificar (de forma não bloqueante) se tem algo no
        // stdin (input do terminal)
        select {
        case x, valid := <-ch:
            if valid {
                fmt.Printf("From keyboard: %s \n", x)
                for j := 0; j < nServers; j++ {
                    go doClientJob(j, 100)
                }
            } else {
                fmt.Println("Closed channel!")
```

Temos uma *goroutine* só para escutar o teclado!

Trata o canal de forma não bloqueante. Ou seja, se não tiver nada, não fico bloqueado esperando.

Também tratamos canal fechado. Mas nesse exemplo ninguém fecha esse canal.

```
            }
        default:
            // Fazer nada!
            // Mas não fica bloqueado esperando o teclado
            time.Sleep(time.Second * 1)
        }

        // Esperar um pouco
        time.Sleep(time.Second * 1)
    }
```

Note que agora o processo ainda pode receber mensagens dos outros processos (com `doServerJob`). Mas a ação dele (i.e. enviar o valor 100 para todos os demais processos) é ativada somente com o recebimento de algo pelo terminal.

- Teste o sistema assim:
  - Terminal 1: `.\Process :10004 :10003`
  - Terminal 2: `.\Process :10003 :10004`
  - Terminal 1: a
  - Terminal 1: x
  - Terminal 2: x

Processo 1 envia 100 aos colegas (no caso, somente o Processo 2)

Processo 1 envia 100 aos colegas (no caso, somente o Processo 2)

Processo 2 envia 100 aos colegas (no caso, somente o Processo 1)

- Resultado esperado:

```
PS C:\Users\jmelo\OneDrive\Desktop\teste\dica3> .\Process :10004 :10003
Received  100  from  127.0.0.1:64935
a
From keyboard: a
x
From keyboard: x
Received  100  from  127.0.0.1:64191
```

```
PS C:\Users\jmelo\OneDrive\Desktop\teste\dica3> .\Process :10003 :10004
Received  100  from  127.0.0.1:64190
Received  100  from  127.0.0.1:64190
x
From keyboard: x
```

- Agora teste com mais processos!

DICA 4: Queremos enviar mensagens estruturadas (definida como `struct` com diferentes informações) para os processos. Para isso precisamos trabalhar com serialização (em especial, aqui usaremos `json`).

- Crie a `struct` (insira após os `imports`):

```
type Message struct {
    Code    int
    Text    string
}
```

Devemos usar letra maiúscula para definir os campos da `struct`. Assim eles serão exportados, ou seja, vistos por outras *packages* (incluindo o `json`).

- Substitua a função `doClientJob` por:

```
func doClientJob(otherProcess int, i int) {
    msg := Message{123,"teste"}
    jsonMsg, err := json.Marshal(msg)
    PrintError(err)

    _, err = CliConn[otherProcess].Write(jsonMsg)
```

A mensagem será fixa. Sempre com *Code* 123 e *Text* 'teste'.

Agora `json` fará parte dos *imports*.

```
    PrintError(err)
}
```

- Substitua a função `doServerJob` por:

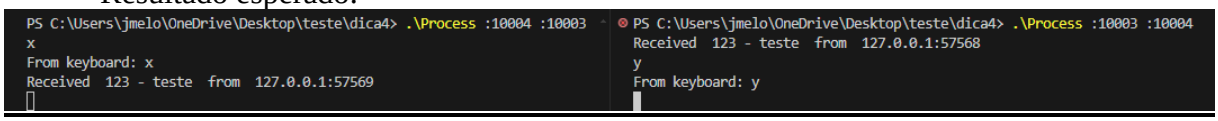
```
func doServerJob() {
    buf := make([]byte, 1024)
    //Loop infinito mesmo
    for {
        // Ler (uma vez somente) da conexão UDP a mensagem
        n, addr, err := ServConn.ReadFromUDP(buf)
        PrintError(err)
        var msg Message
        err = json.Unmarshal(buf[:n], &msg)
        PrintError(err)
        // Escrever na tela a msg recebida (indicando o
        // endereço de quem enviou)
        fmt.Println("Received ", msg.Code, "-", msg.Text, " from ", addr)
    }
}
```

- Teste o sistema assim:
  - Terminal 1: `.\Process :10004 :10003`
  - Terminal 2: `.\Process :10003 :10004`
  - Terminal 1: x
  - Terminal 2: y

Processo 1 envia “123 teste” aos colegas (no caso, somente o Processo 2)

Processo 2 envia “123 teste” aos colegas (no caso, somente o Processo 1)

- Resultado esperado:



- Agora teste com mais processos!

Bom estudo!