Lecture 05
Introduction to Golang

Dr. Shujaat Hussain

Policy for Late comers

- Late comers would lose any awarded or to be awarded bonus absolute!
- This can be undone by answering any future bonus questions.

- To whom this policy applies?
  - More than 5 minutes late in two classes

- Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.

Why should we learn a new Language?

- Go is expressive, concise, clean, and efficient – says a lot !!
- Go is easy and fun ☺

- Just play along and you'll find out

Go at Google

- Go is a programming language designed by Google to help solve Google's problems, and Google has big problems.

- The hardware is big and the software is big.
  - many millions of lines of software, with servers mostly in C++ and lots of Java and Python for the other pieces.
  - Thousands of engineers work on the code, at the "head" of a single tree comprising all the software,
  - From day to day there are significant changes to all levels of the tree.
  - A large custom-designed distributed build system makes development at this scale feasible, but it's still big.

Go at Google

- And of course, all this software runs on zillions of machines, which are treated as a modest number of independent, networked compute clusters.

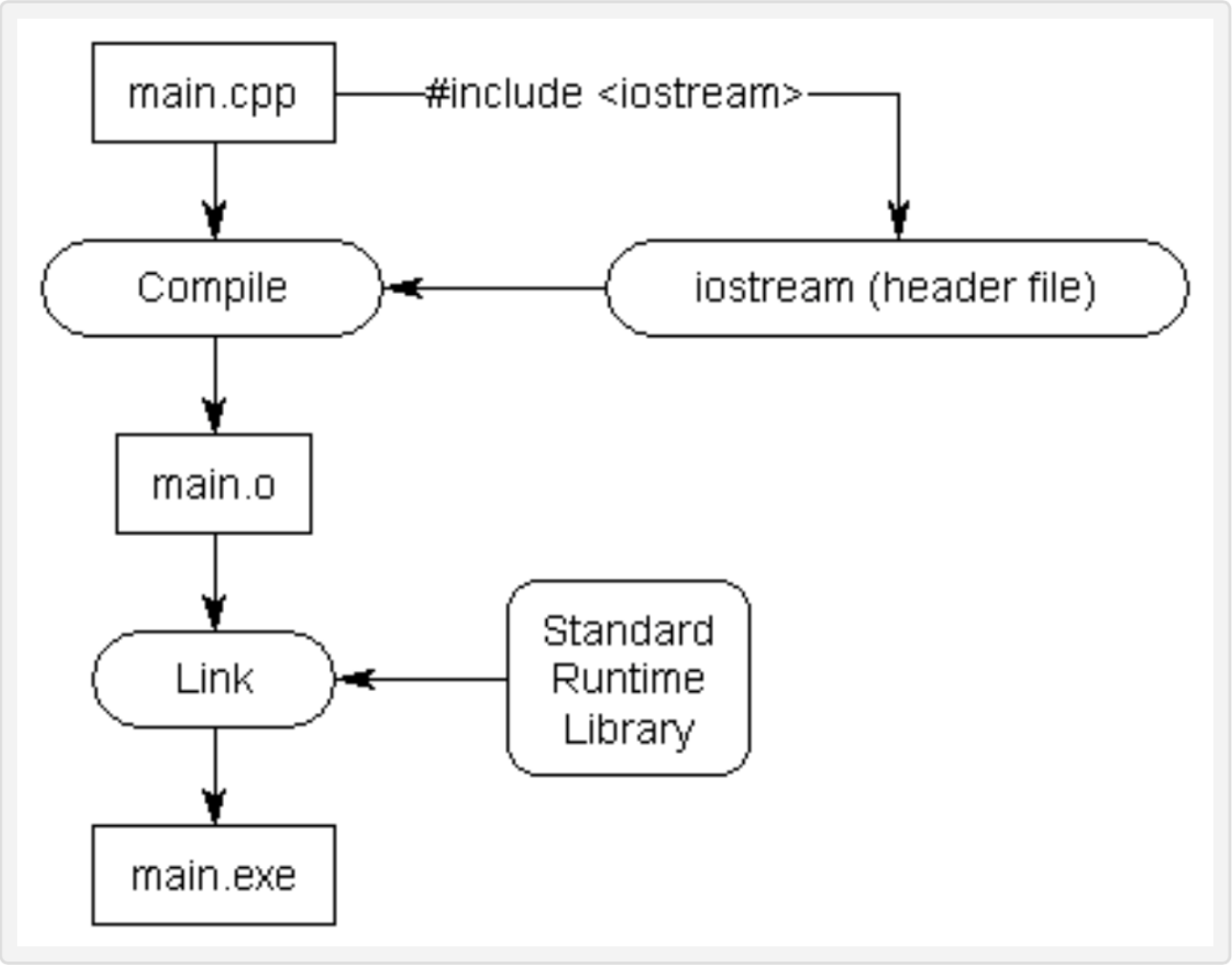
Pain points

- slow builds
- uncontrolled dependencies
- each programmer using a different subset of the language
- poor program understanding (code hard to read, poorly documented, and so on)
- duplication of effort
- cost of updates
- difficulty of writing automatic tools
- ….

Header files in C++

```
1  #include <iostream>
2  int main()
3  {
4      using namespace std;
5      cout << "Hello, world!" << endl;
6      return 0;
7  }
```

Header files in C++



Header files in C++

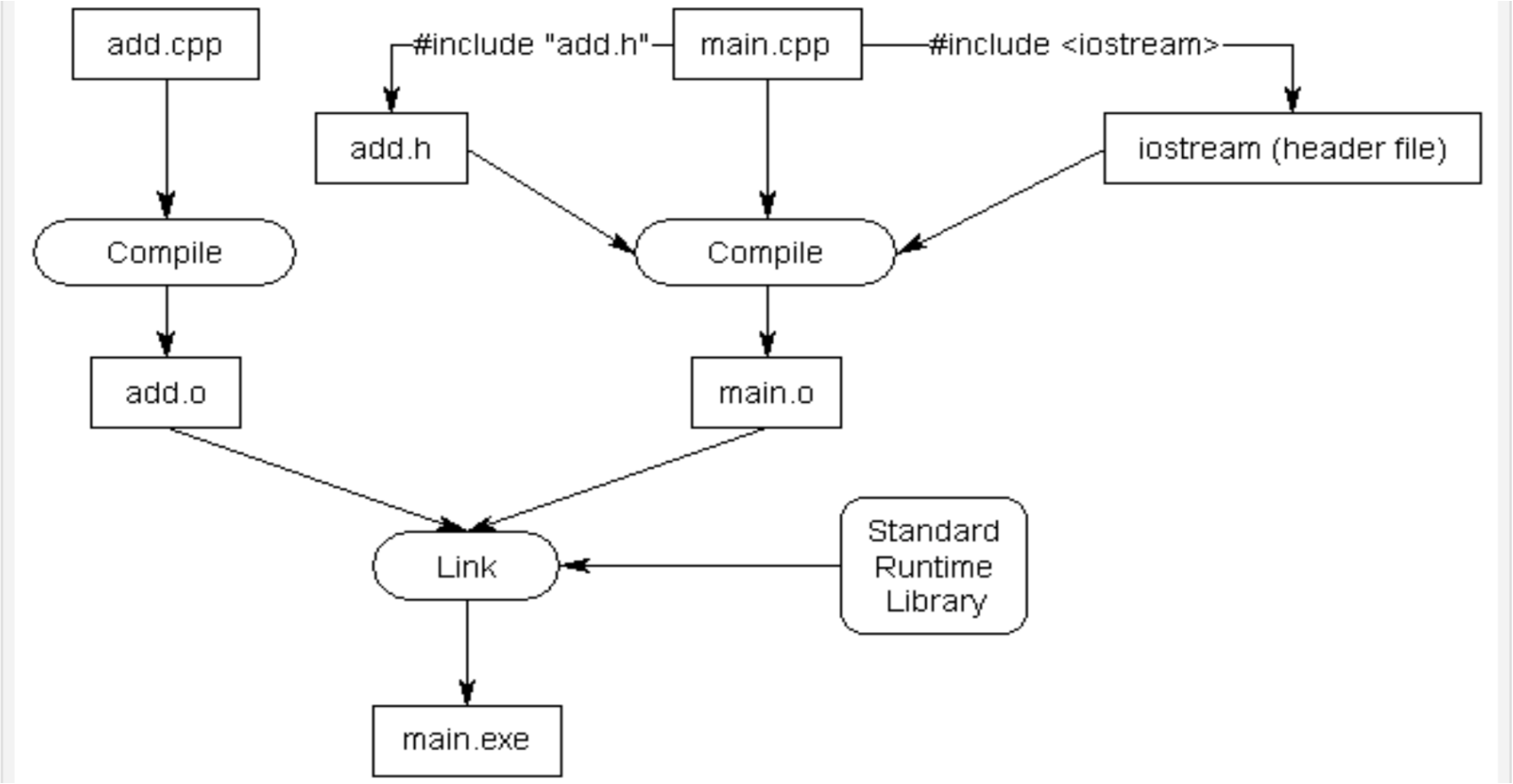
main.cpp that includes add.h:

```cpp
#include <iostream>
#include "add.h"

int main()
{
    using namespace std;
    cout << "The sum of 3 and 4 is " << add(3, 4) << endl;
    return 0;
}
```

add.cpp stays the same:

```cpp
int add(int x, int y)
{
    return x + y;
}
```

Header files in C++



Header files in C++

- Header files consist of two parts.
  - The first part is called a **header guard**
  - The second part is the actual content of the .h file

Header files in C++

```
// This is start of the header guard.  ADD_H can be any unique name.  By convention,
#ifndef ADD_H
#define ADD_H

// This is the content of the .h file, which is where the declarations go
int add(int x, int y); // function prototype for add.h -- don't forget the semicolon

// This is the end of the header guard
#endif
```

Dependencies in C and C++

- The #ifdef preprocessor directive allow the preprocessor to check whether a value has been previously #defined. If so, the code between the #ifdef and corresponding #endif is compiled. If not, the code is ignored.

```
#define PRINT_JOE

#ifdef PRINT_JOE
cout << "Joe" << endl;
#endif

#ifdef PRINT_BOB
cout << "Bob" << endl;
#endif
```

Dependencies in C and C++

- #ifndef is the opposite of #ifdef, in that it allows you to check whether a name has NOT been defined yet.

```
#ifndef PRINT_BOB
cout << "Bob" << endl;
#endif
```

Dependencies in C and C++

- **Header guard**, prevent a given header file from being #included more than once from the same file.

```
#ifndef SOME_UNIQUE_NAME_HERE
#define SOME_UNIQUE_NAME_HERE

// your declarations here

#endif
```

Nice... but it scales very badly

- In 1984, a compilation of ps.c, the source to the Unix ps command, was observed to #include <sys/stat.h> 37 times by the time all the preprocessing had been done.

- The construction of a single C++ binary at Google can open and read hundreds of individual header files tens of thousands of times.

Dependencies in C and C++

- In 2007, build engineers at Google instrumented the compilation of a major Google binary.
  - Two thousand files that, when concatenated together, totaled 4.2 megabytes.
  - By the time the #includes had been expanded, over 8 gigabytes were being delivered to the input of the compiler, a blow-up of 2000 bytes for every C++ source byte.

Dependencies in C and C++

- That 2007 binary took 45 minutes using a distributed build system
- When builds are slow, there is time to think. The origin myth for Go states that it was during one of those 45 minute builds that Go was conceived.

Dependencies in Go

import "encoding/json"

The first step to making Go scale, dependency-wise, is that the *language* defines that unused dependencies are a compile-time error (not a warning, an *error*).

Dependencies in Go

package A imports package B;

package B imports package C;

package A does not import package C

This means that package A uses C only transitively through its use of B; that is, no identifiers from C are mentioned in the source code to A, even if some of the items A is using from B do mention C

Dependencies in Go

- To build this program,
  - first, C is compiled; dependent packages must be built before the packages that depend on them.
  - Then B is compiled; finally A is compiled, and then the program can be linked.

- When A is compiled, the compiler reads the object file for B, not its source code. That object file for B contains all the type information necessary for the compiler to execute the import "B" clause in the source code for A.

Results

- Google measured the compilation of a large Google program written in Go to see how the source code fanout compared to the C++ analysis done earlier.
- They found it around fifty times better than C++ (as well as being simpler and hence faster to process),
- It can be further improved !!

Lecture 05 - Introduction to Golang

let's go!

Installing Go

- The Go Playground, http://play.golang.org

- The Go Playground is a web service that runs on golang.org's servers. The service receives a Go program, compiles, links, and runs the program inside a sandbox, then returns the output.

https://golang.org

The Go Programming Language

Documents | Packages | The Project | Help | Blog | Search

Try Go

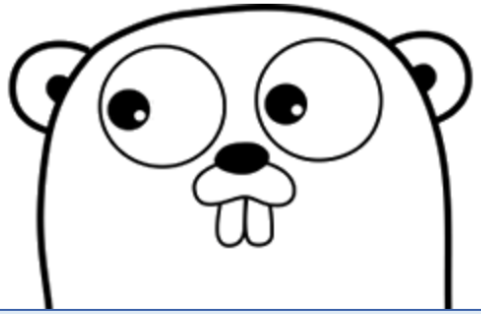
Pop-out

```
// You can edit this code!
// Click here and start typing.
package main

import "fmt"

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

Hello, World!

Run | Share | Tour

Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.



Download Go

Binary distributions available for
Linux, Mac OS X, Windows, and more.

Installing Go

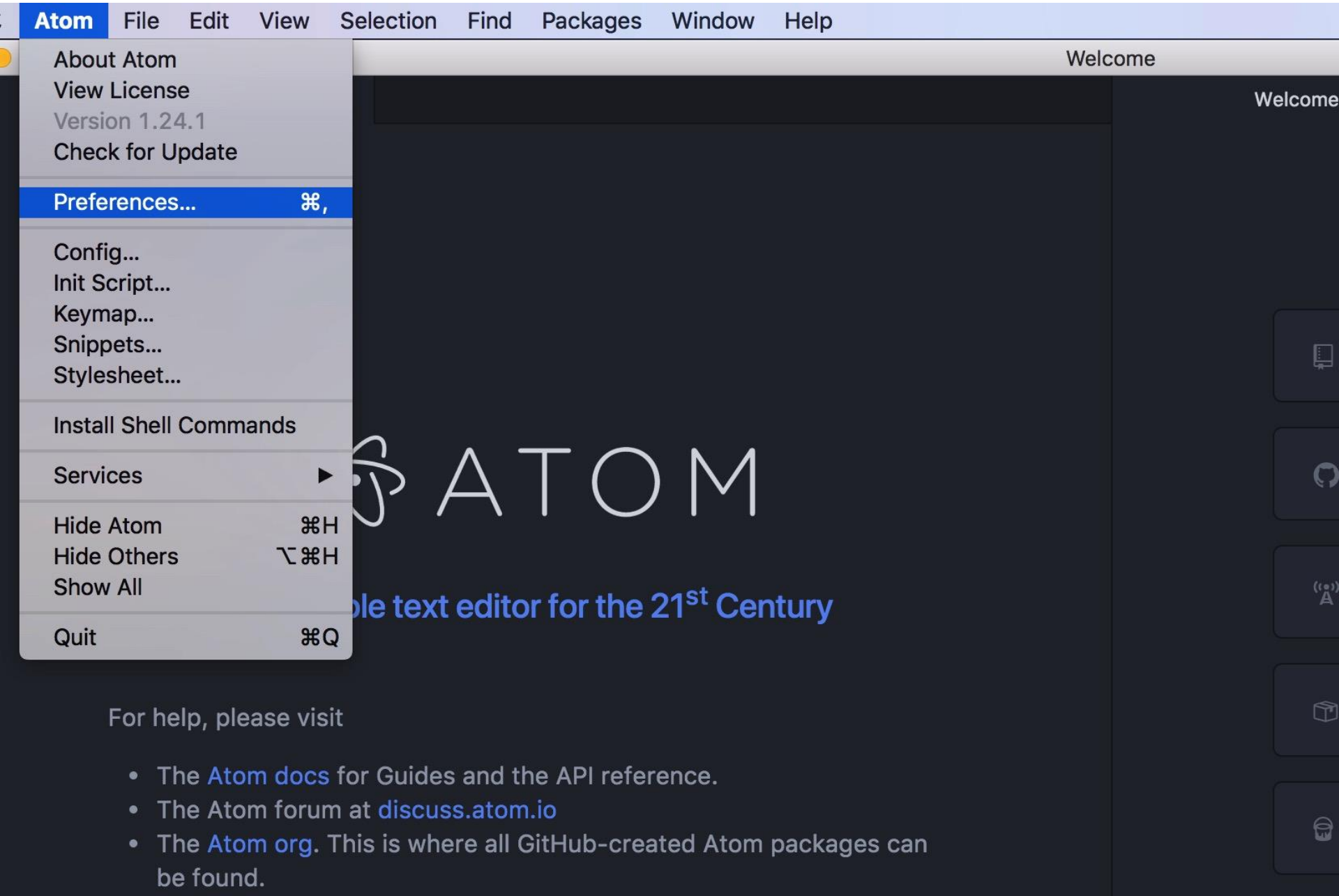
- You should however install it locally
  - Follow instructions at https://golang.org/doc/install, simple steps and binaries provided.
  - Download and install using the setup and do the following to test your work
  - The GOPATH environment variable should be setup

Some editors with support for Go

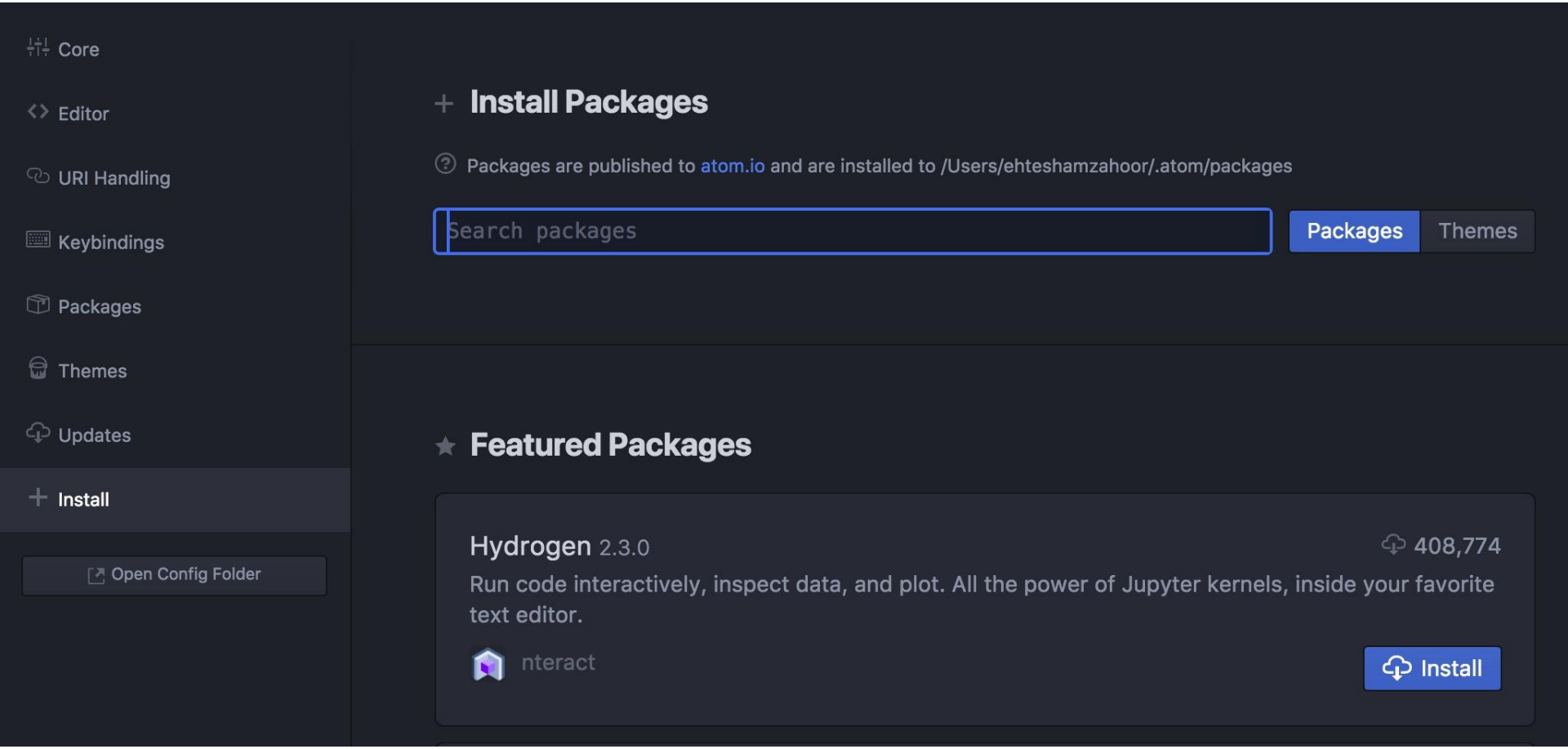
- SublimeText with GoSublime
- Atom with go-plus

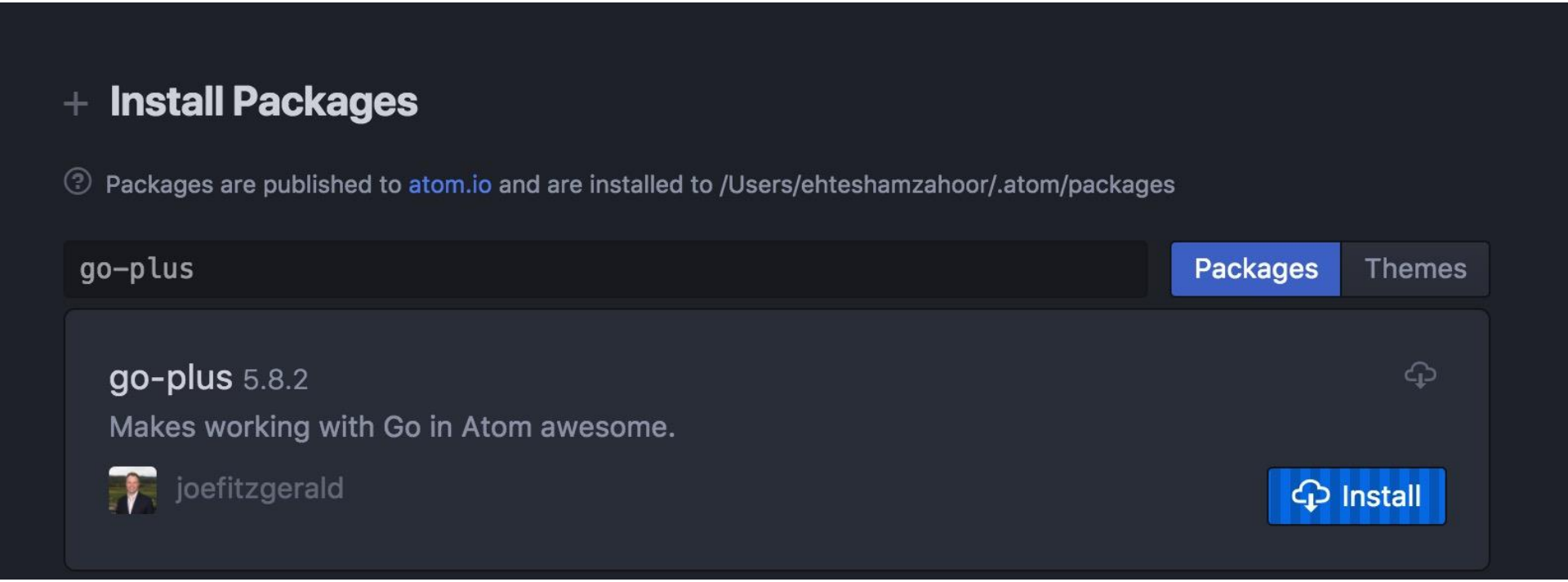
Lecture 05 - Introduction to Golang

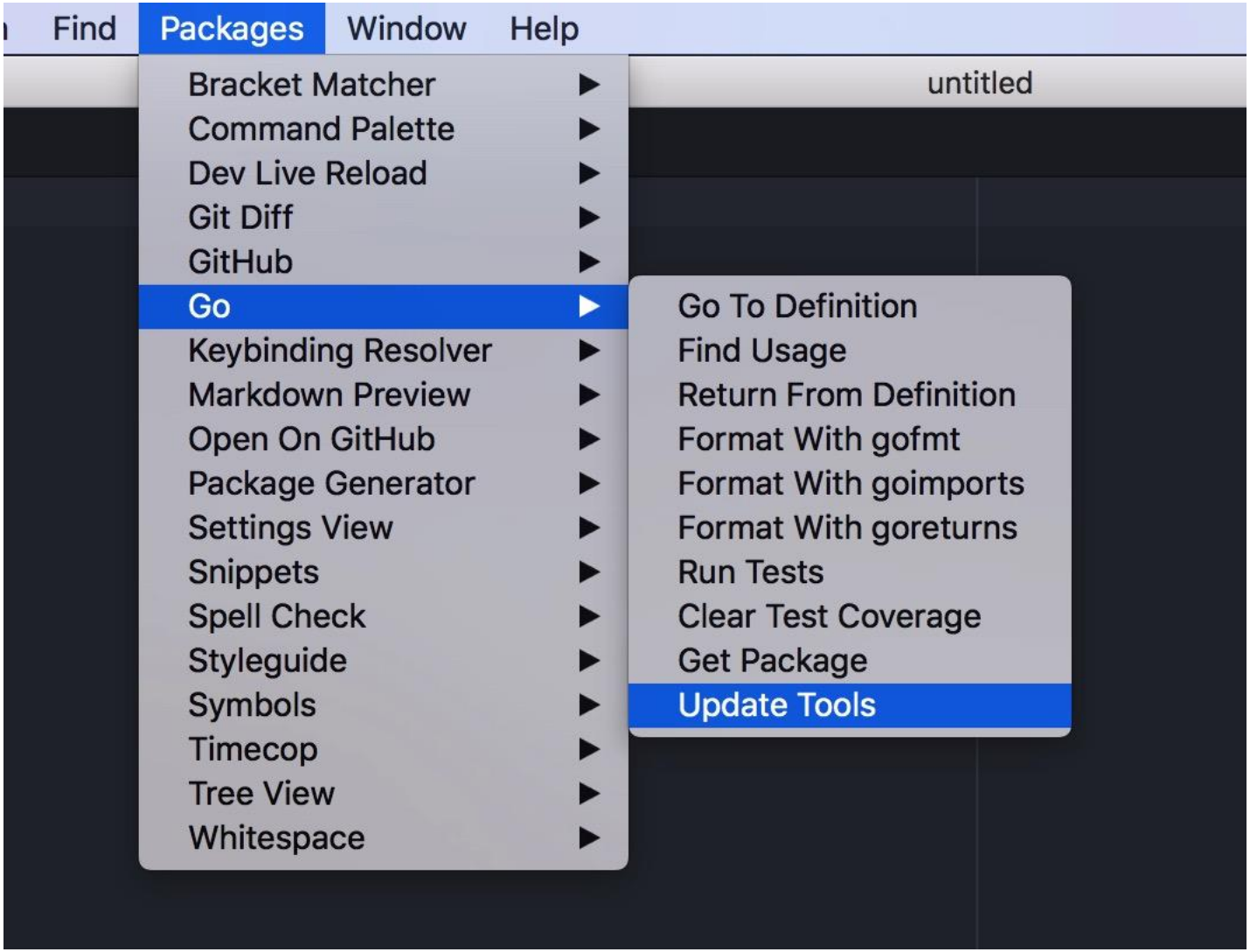
Lecture 05 - Introduction to Golang



Lecture 05 - Introduction to Golang




+ Install Packages
Packages are published to atom.io and are installed to /Users/ehteshamzahoor/.atom/packages
go-plus
Packages
Themes
go-plus 5.8.2
Makes working with Go in Atom awesome.
joefitzgerald
Install


Find
Packages
Window
Help
untitled
Bracket Matcher
Command Palette
Dev Live Reload
Git Diff
GitHub
Go
Keybinding Resolver
Markdown Preview
Open On GitHub
Package Generator
Settings View
Snippets
Spell Check
Styleguide
Symbols
Timecop
Tree View
Whitespace
Go To Definition
Find Usage
Return From Definition
Format With gofmt
Format With goimports
Format With goreturns
Run Tests
Clear Test Coverage
Get Package
Update Tools

Lecture 05 - Introduction to Golang

go-plus

Go

Output

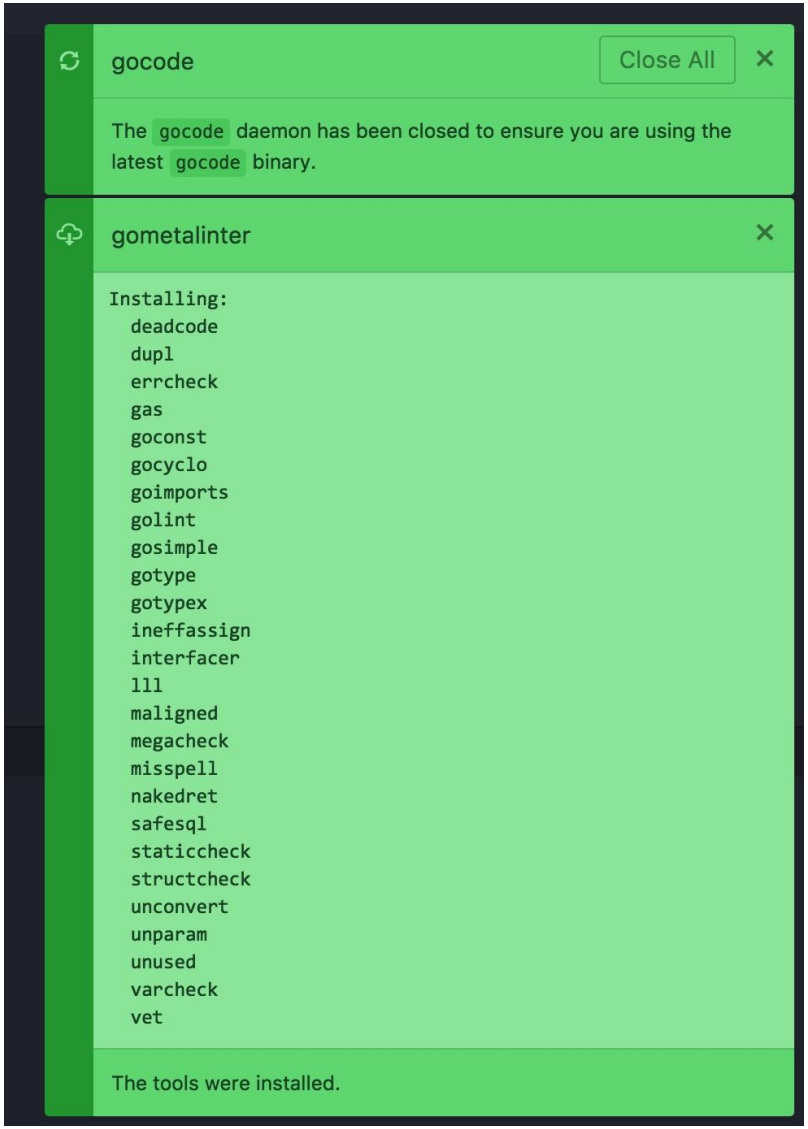
Reference

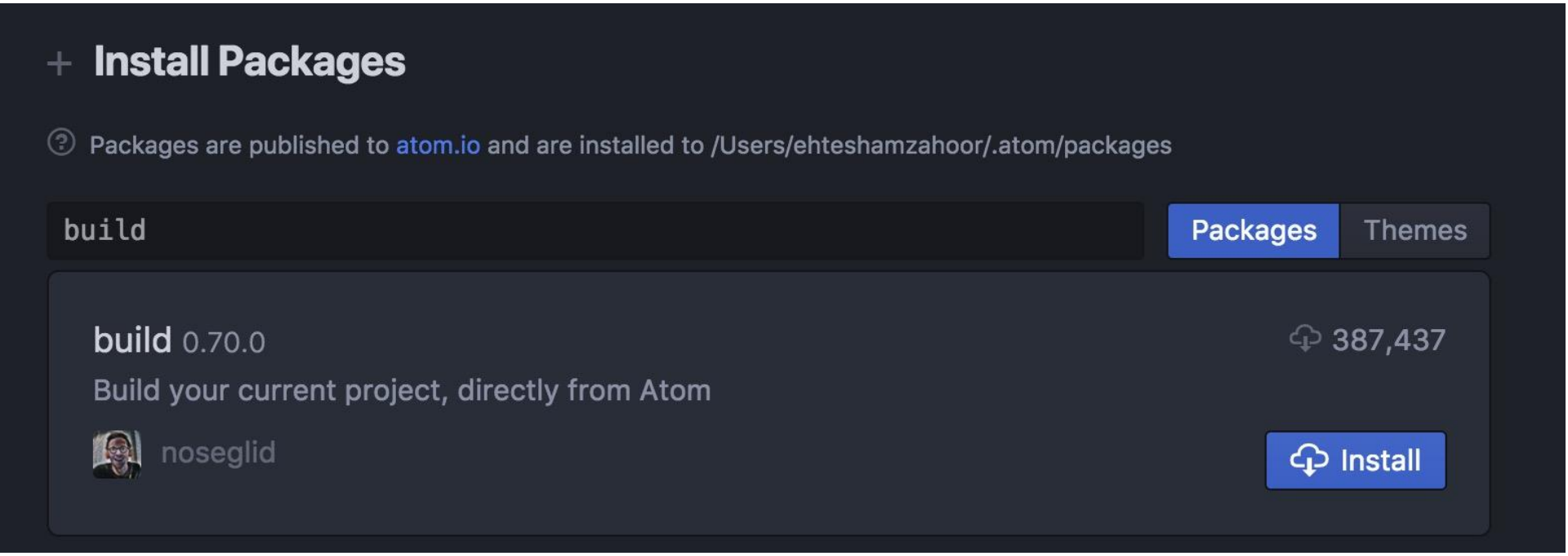
Usage

Implements

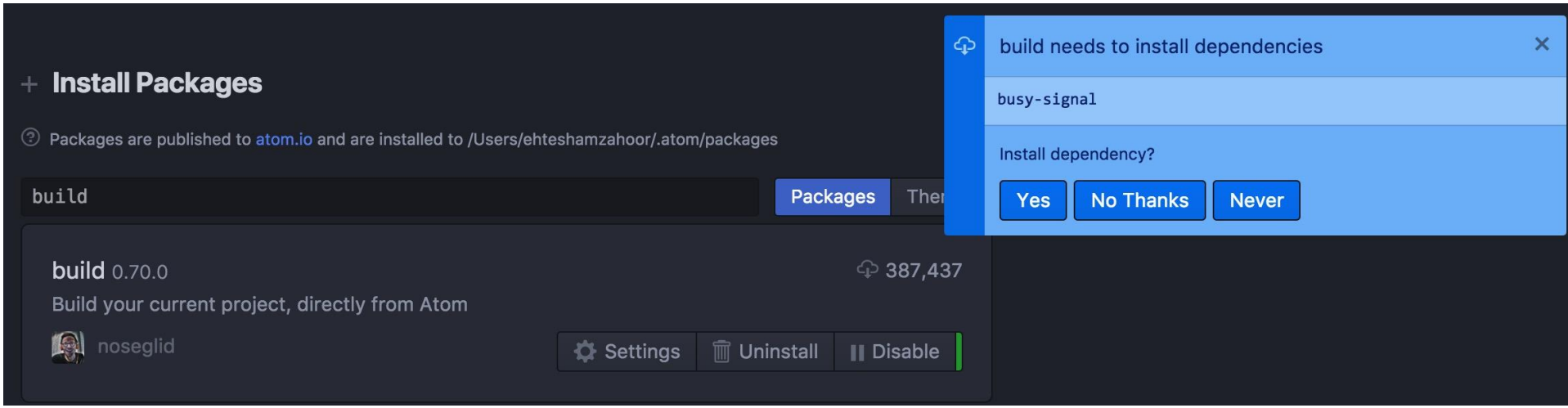
```
$ go get -u golang.org/x/tools/cmd/goimports
$ go get -u golang.org/x/tools/cmd/gorename
```

Lecture 05 - Introduction to Golang



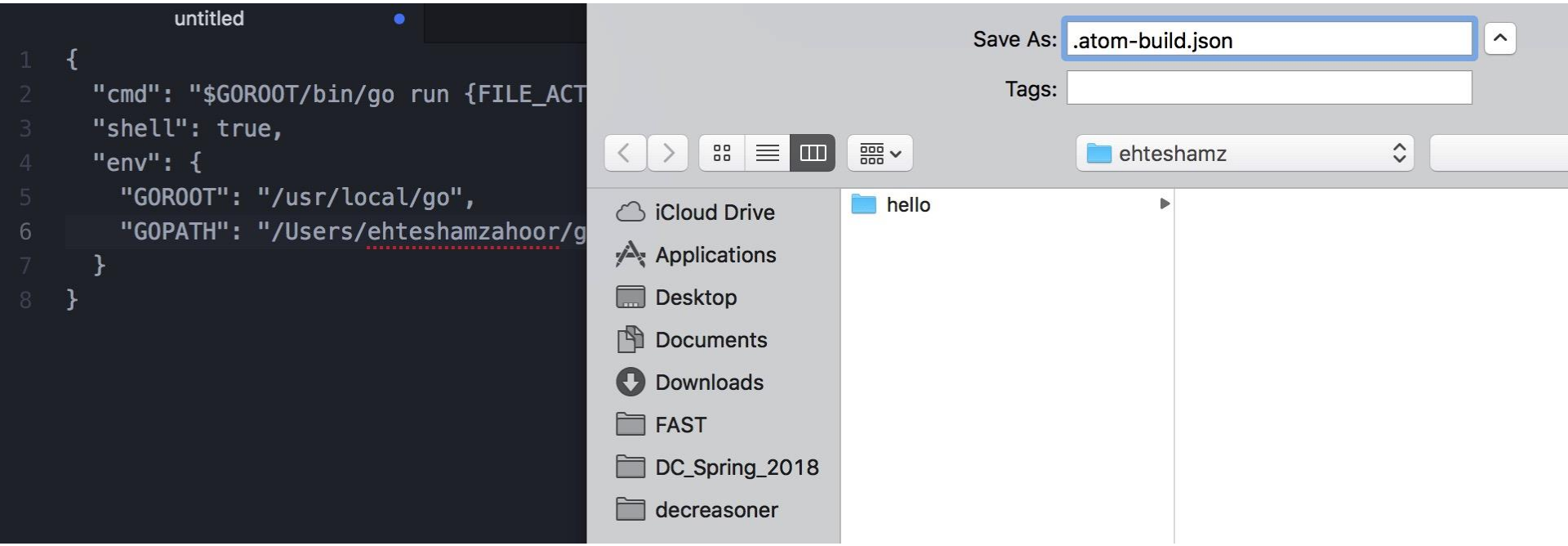

+ Install Packages
Packages are published to atom.io and are installed to /Users/ehteshamzahoor/.atom/packages
build
Packages
Themes
build 0.70.0
387,437
Build your current project, directly from Atom
noseglid
Install

Lecture 05 - Introduction to Golang



```
{
  "cmd": "$GOROOT/bin/go run {FILE_ACTIVE}",
  "shell": true,
  "env": {
    "GOROOT": "/usr/local/go",
    "GOPATH": "/Users/ehteshamzahoor/go"
  }
}
```

Lecture 05 - Introduction to Golang



Lecture 05 - Introduction to Golang

Project

- ehteshamz
  - hello
    - hello.go
  - .atom-build.json
  - .DS_Store

.atom-build.json | hello.go

```go
package main

import "fmt"

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

Hello World !!

```
package main
import "fmt"

func main() {
        fmt.Printf("Hello, world.\n")
}
```

A web server !!

```go
package main
import (
	"io"
	"net/http"
)
func hello(w http.ResponseWriter, r *http.Request) {
	io.WriteString(w, "Hello world!")
}

func main() {
	http.HandleFunc("/", hello)
	http.ListenAndServe(":8000", nil)
}
```

Let’s not rush

- Before we can move on, it is better to get organized.

- Good organization of our code and directories would do wonders ☺

Go and GitHub

- The go tool is designed to work with open source code maintained in public repositories.

- Although you don't need to publish your code, the model for how the environment is set up works the same whether you do or not.

Workspaces

- Go code must be kept inside a workspace.
- A workspace is a directory hierarchy with three directories at its root:
  - src contains Go source files organized into packages (one package per directory),
  - pkg contains package objects, and
  - bin contains executable commands.

- The go tool builds source packages and installs the resulting binaries to the pkg and bin directories.

An example

```
bin/
    hello                           # command executable
    outyet                          # command executable
pkg/
    linux_amd64/
        github.com/golang/example/
            stringutil.a            # package object
src/
    github.com/golang/example/
        .git/                       # Git repository metadata
        hello/
            hello.go                # command source
        outyet/
            main.go                 # command source
            main_test.go            # test source
        stringutil/
            reverse.go              # package source
            reverse_test.go         # test source
```

Go and Version Control

- If you're using a source control system, now would be a good time to initialize a repository, add the files, and commit your first change.
  - Why you should be using one?
  - Which one to use?

What is Version Control? (why would I need it...)

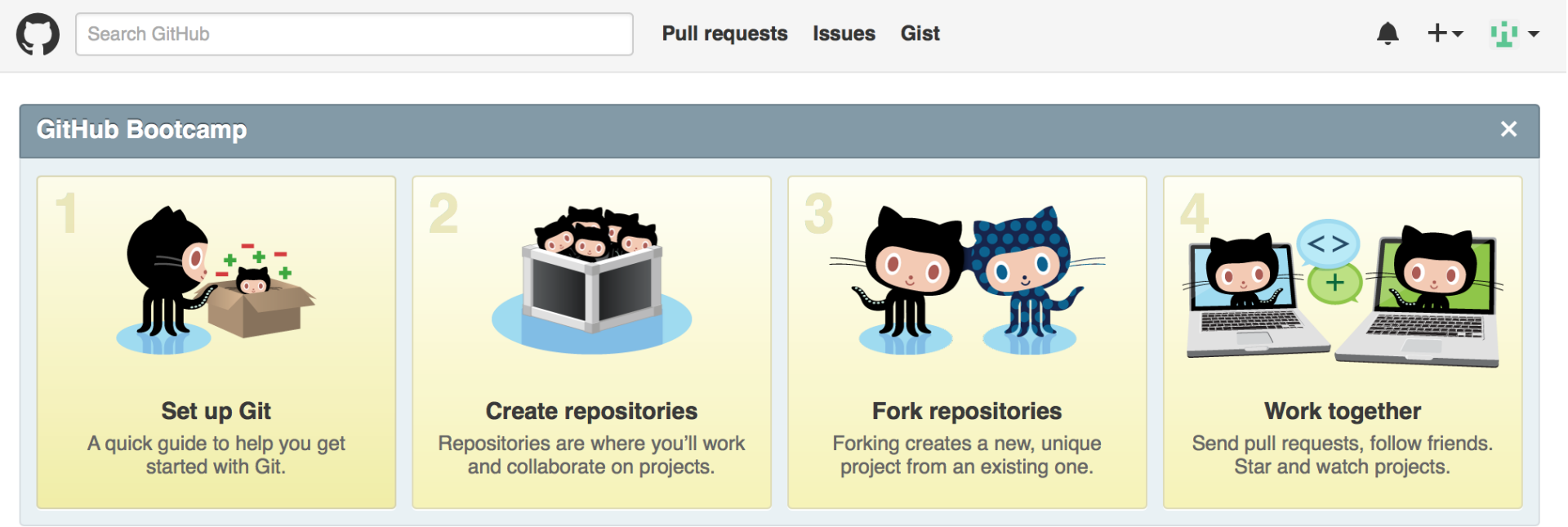
- Files goes through frequent changes, new files coming in and old ones being deleted.
- If you build any kind of application and need to keep track of “versions” of the filesystem, you need version control.
- If you need to share coding responsibilities or maintenance of a codebase with another person, you need version control.

- If you want to ***look like a pro*** at managing your code, keep a consistent history (with ease), and feel comfortable handing it off to someone else, you need version control.

Types of Version Control Systems

- Local only - keeps a local database of changes in your local machine filesystem.
- Centralized - (Subversion, CVS), require a connection to a central server and “checkout”
- Distributed - (Git, Mercurial) allow for local systems to be “mirrors” of the central repo. You don’t need to be connected to the central server to get work or commits done.

GitHub


Search GitHub
Pull requests
Issues
Gist
GitHub Bootcamp
1
Set up Git
A quick guide to help you get started with Git.
2
Create repositories
Repositories are where you'll work and collaborate on projects.
3
Fork repositories
Forking creates a new, unique project from an existing one.
4
Work together
Send pull requests, follow friends. Star and watch projects.

What Git is and what it does

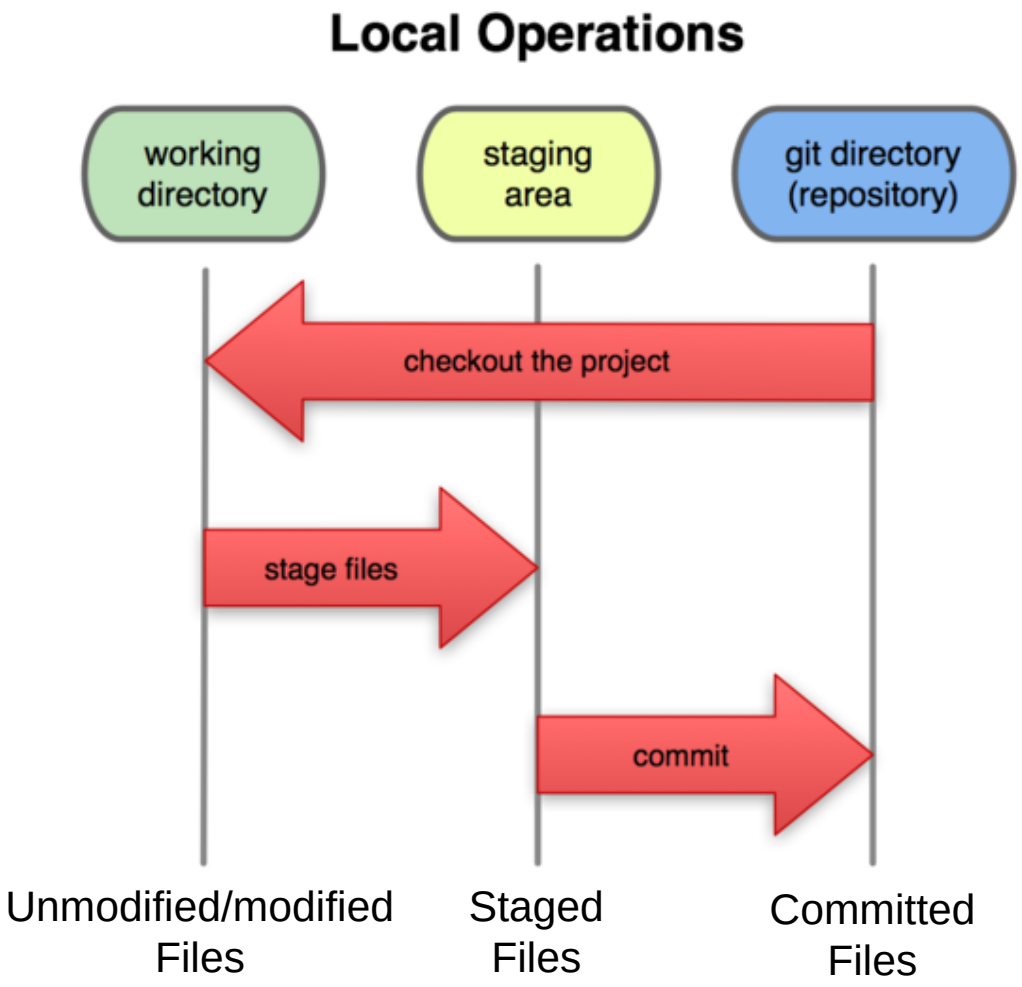
- small piece of software.
- tracks all your files in it's database. You have to add and remove files into this tracking system and commit them.
- a single .git directory at the top of your filesystem "watches" the changes going on and helps you deal with them at commit time.
- "git status" will tell you what has been added, removed, modified, etc. etc.
- When you make a "commit." Git records a "snapshot" of changes to your filesystem and records an index number to it. You also write a brief message about the commit.
- Commit frequently!

Where it all happens

- In each local .git folder.
- most activity happens locally. Frequent commits, additions, and changes.

- When you’re ready to share your code with others in your team, or want to get it ready for deployment to your servers, you can “push” it to your "remote"

A Local Git project has three areas



Note: working directory sometimes called the “working tree”, staging area sometimes called the “index”.

Basic Workflow

Basic Git workflow:

1. **Modify** files in your working directory.
2. **Stage** files, adding snapshots of them to your staging area.
3. Do a **commit**, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

- Notes:
  - If a particular version of a file is in the **git directory**, it's considered **committed**.
  - If it's modified but has been added to the **staging area**, it is **staged**.
  - If it was **changed** since it was checked out but has <u>not</u> been staged, it is **modified**.

Aside: So what is github?

- GitHub.com is a site for online storage of Git repositories.
- Many open source projects use it, such as the Linux kernel.

Question: Do I have to use github to use Git?

Answer: No! you can use Git locally for your own purposes

Aside: So what is github?

- You can get free space for open source projects or you can pay for private projects.

Student Developer Pack

Get ready to use Git!

1. Set the name and email for Git to use when you commit:

```
$ git config --global user.name “Bugs Bunny”
$ git config --global user.email bugs@gmail.com
```

- You can call `git config –list` to verify these are set.
- These will be set globally for all Git projects you work with.
- You can also set variables on a project-only basis by not using the `--global` flag.

Create a local copy of a repo

2. Two common scenarios: (only do one of these)

   a) To **clone an already existing repo** to your current directory:

   ```
   $ git clone <url> [local dir name]
   ```

   This will create a directory named *local dir name*, containing a working copy of the files from the repo, and a `.git` directory (used to hold the staging area and your actual repo)

   b) To **create a Git repo** in your current directory:

   ```
   $ git init
   ```

   This will create a `.git` directory in your current directory.

   Then you can commit files in that directory into the repo:

   ```
   $ git add file1.java
   $ git commit –m "initial project version"
   ```

Git commands

command	description
`git clone` ***`url [dir]`***	copy a git repository so you can add to it
`git add` ***`files`***	adds file contents to the staging area
`git commit`	records a snapshot of the staging area
`git status`	view the status of your files in the working directory and staging area
`git diff`	shows diff of what is staged and what is modified but unstaged
`git help` ***`[command]`***	get help info about a particular command
`git pull`	fetch from a remote repo and try to merge into the current branch
`git push`	push your new branches and data to a remote repository
others: `init, reset, branch, checkout, merge, log, tag`	

Committing files

- The first time we ask a file to be tracked, *and* **every time before we commit a file** we must add it to the staging area:

```
$ git add hello.java
```

- This takes a snapshot of these files at this point in time and adds it to the staging area.
- To move staged changes into the repo we commit:

```
$ git commit –m “Fixing bug #22”
```

Note: These commands are just acting on **<u>your local version of repo</u>**.

Status and Diff

- To view the **status** of your files in the working directory and staging area:

```
$ git status
```

or

```
$ git status -s
```

- To see what is modified but unstaged:

```
$ git diff
```

Viewing logs

To see a log of all changes in your local repo:

- `$ git log` or
- `$ git log --oneline` (to show a shorter version)

  1677b2d Edited first line of readme
  258efa7 Added line to readme
  0e52da7 Initial commit

- `git log -5` (to show only the 5 most recent updates, etc.)

Undoing Changes - git checkout

- Undo changes to a file since last commit
  - git checkout -- *filename*
- Look at previous version of project

```
git log --oneline
```

```
b7119f2 Continue doing crazy things
872fa7e Try something crazy
a1e8fb5 Make some important changes to hello.py
435b61d Create hello.py
9773e52 Initial import
```

```
git checkout a1e8fb5
```

```
git checkout master
```

Undoing Changes - git checkout

- If you liked some particular file in previous version – you can restore it

```
git checkout a1e8fb5 hello.py
```

Remote Repositories - Push and pull

- To push our local repo to the GitHub server we'll need to add a remote repository.

  git remote add origin <remote-repo-url>

  git push -u origin master

- Pulling is just as simple

  git pull origin master

A Tour of Go

- We would be following the online tour
  - *https://tour.golang.org* *or* *go tool tour*

Hello World !!

```
package main
import "fmt"

func main() {
        fmt.Printf("Hello, world.\n")
}
```

Printing on Console - The fmt package

```go
package main

import "fmt"

func main() {
    // The Println method can handle one or more arguments.
    fmt.Println("cat")
    fmt.Println("cat", 900)

    // Use Println on a slice.
    items := []int{10, 20, 30}
    fmt.Println(items)
}
```

Output

```
cat
cat 900
[10 20 30]
```

Printing on Console - The fmt package

```go
package main

import "fmt"

func main() {
    elements := []int{999, 99, 9}

    // Loop over the int slice and Print its elements.
    // ... No newline is inserted after Print.
    for i := 0; i < len(elements); i++ {
        fmt.Print(elements[i] + 1)
        fmt.Print(" ")
    }
    fmt.Println("... DONE!")
}
```

Output

```
1000 100 10 ... DONE!
```

Printing on Console - The fmt package

```go
package main

import "fmt"

func main() {
    name := "Mittens"
    weight := 12

    // Use %s to mean string.
    // ... Use an explicit newline.
    fmt.Printf("The cat is named %s.\9", name)
    // Use %d to mean integer.
    fmt.Printf("Its weight is %d.\9", weight)
}
```

Output

```
The cat is named Mittens.
Its weight is 12.
```

Printing on Console - The fmt package

```go
package main

import "fmt"

func main() {
    result := true
    name := "Spark"
    size := 2000
    // Print line with v format codes.
    fmt.Printf("Result = %v, Name = %v, Size = %v",
        result, name, size)
}
```

Output

```
Result = true, Name = Spark, Size = 2000
```

A tour of Go

- **Packages/Imports/Exported names/Functions/Variables/Initializers**

Go Packages and Exported Names

- Let's write a library and use it from the hello program, ***greetings***.
- $ **mkdir $GOPATH/src/github.com/user/greetings**
- Create a file named mygreetings.go with following code

```
//Package greetings shows the greetings
package greetings

//GreetingsString is a global variable
var GreetingsString = "Hello World"

//PrintGreetings is a global function
func PrintGreetings(name string) string {
        return GreetingsString + "-" + name
}
```

Go Packages and Exported Names

- Next, modify the hello.go to have the following code:

```
package main

import (
        "fmt"
        "github.com/ehteshamz/greetings"
)

func main() {
        fmt.Printf(greetings.PrintGreetings("ez "))
}
```

A few words about godoc

- A simple, elegant and highly effective approach to document your work (my own words!)
- We added some comments with our package
- If we godoc -http=:6060 and navigate to localhost:6060/pkg/github.com/ehteshamz/greetings, we can see these comments!!

Remote repositories

- It is good time to push our package to the GitHub,
- Anyone can then use our wonderful greetings package

- You know how to do this?
  - git init/add/commit/push … piece of cake ☺

Remote repositories

- Lets delete our local repo and get it from GitHub
- This shows how to use remote repos in the Go

  go get github.com/ehteshamz/greetings

  …and we have our package back ☺

Remote repositories

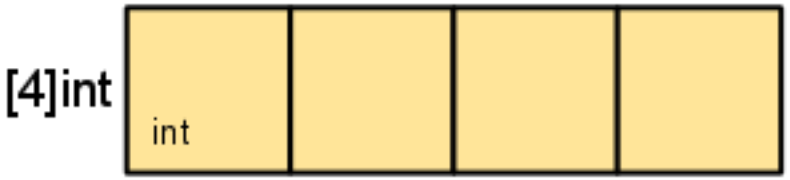
- Lets do some image processing …

  go get -u github.com/disintegration/imaging

  …and we have an image processing library

```
package main
import (
        "github.com/disintegration/imaging"
)

func main() {
        img, _ := imaging.Open("01.jpg")
        thumb := imaging.Thumbnail(img, 100, 100, imaging.CatmullRom)
        imaging.Save(thumb, "dst.jpg")
}
```

Arrays



- Go's arrays are values.
- An array variable denotes the entire array; it is not a pointer to the first array element (as in C/C++).

- This means that when you assign or pass around an array value you will make a copy of its contents.

Passing Arrays

- Go's documentation makes it clear that arrays are passed by copy.

- How to achieve pass by reference?
  - *Option A - use pointers, complex and less elegant*
  - *Option B - Use slices*

Slices

- Slices build on arrays to provide great power and convenience.
- A slice literal is declared just like an array literal, except you leave out the element count:

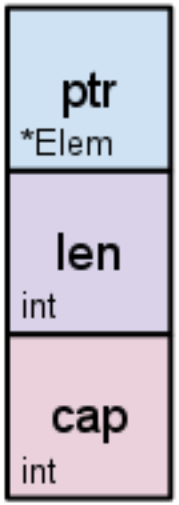
```
letters := []string{"a", "b", "c", "d"}

var s []byte
s = make([]byte, 5, 5)
// s = []byte{0, 0, 0, 0, 0}
```

Slicing Slices

```
b := []byte{'g', 'o', 'l', 'a', 'n', 'g'}
```

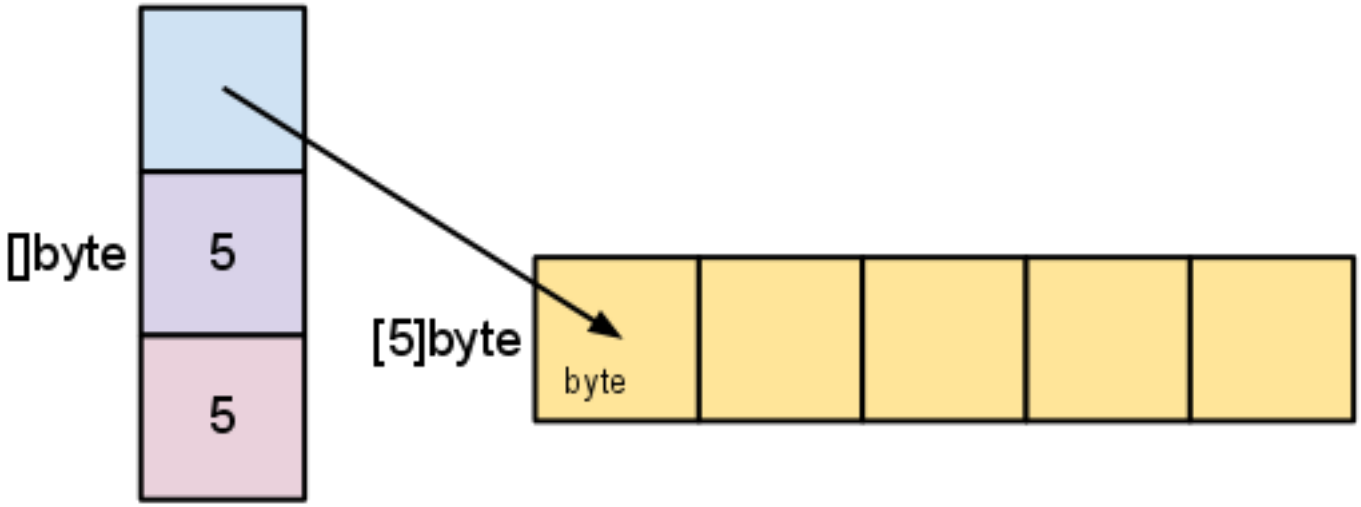
```
// b[1:4] == []byte{'o', 'l', 'a'}, sharing the same storage as b
// b[:2] == []byte{'g', 'o'}
// b[2:] == []byte{'l', 'a', 'n', 'g'}
// b[:] == b
```

Slice Internals



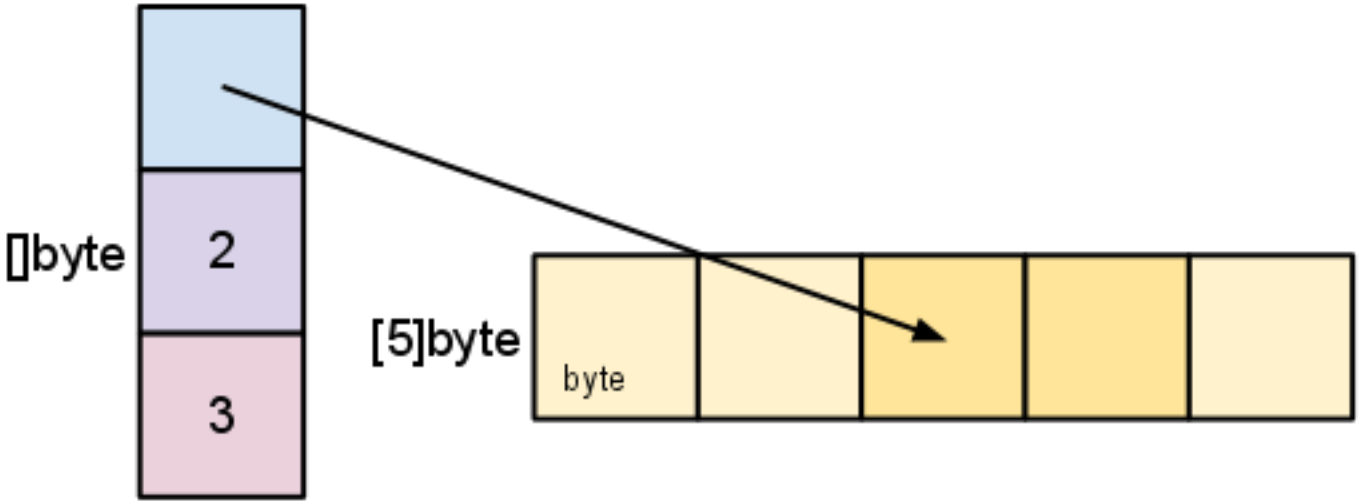
- A slice is a descriptor of an array segment. It consists of a pointer to the array, the length of the segment, and its capacity (the maximum length of the segment).

- Our variable s, created earlier by make([]byte, 5), is structured like this:



Slice Internals - Slicing

s = s[2:4]



Slice Internals - Slicing

- Slicing does not copy the slice's data.
- It creates a new slice value that points to the original array.

- Therefore, modifying the *elements* (not the slice itself) of a re-slice modifies the elements of the original slice

```
d := []byte{'r', 'o', 'a', 'd'}
e := d[2:]
// e == []byte{'a', 'd'}
e[1] = 'm'
// e == []byte{'a', 'm'}
// d == []byte{'r', 'o', 'a', 'm'}
```

Erratum - Go Modules

- The support of go modules was added in v1.11 but wasn’t enforced for $GOPATH packages till recent versions.

- Packages are handled differently with the support and enforcement of go modules.

GO111MODULE

- If you have made the change as announced on the classroom, you need to undo. It should be on or empty

- You can check the go environment variables by using the command shown below:

```
go env
```

Go modules - Introduction

- They serve many purposes
  - Dependency management, using multiple versions of the downloaded packages
  - Packages from go get are not cluttered in $GOPATH/src and go code can be written in directories other than $GOPATH
  - …

back to the greetings package

```go
//Package greetings shows the greetings
package greetings

//GreetingsString is a global variable
var GreetingsString = "Hello World"

//PrintGreetings is a global function
func PrintGreetings(name string) string {
	return GreetingsString + "-" + name
}
```

back to the greetings package

- With the go modules its path is not guided by the directory structure but rather a file named go.mod

Getting started

- Go modules are initialized by the go init command followed up by the path of the module

- This creates a file named go.mod in the same directory. The contens of the file are shown below, it names the module and go version needed for this module.

Version control using git

- We know how to do this, right?

```
>git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        go.mod
        greetings.go

nothing added to commit but untracked files present (use "git add" to track)
>git add .
>git commit -m "go mod version v0.0.1"
[master (root-commit) c3f06dc] go mod version v0.0.1
 Committer: ez <ehteshamzahoor@ezmbp.local>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 2 files changed, 13 insertions(+)
 create mode 100644 go.mod
 create mode 100644 greetings.go
>
```

Git tags

- Using git we can tag our releases (provide versions) and push on the github.
- We call our first release v0.0.1

```
>
>git remote add origin https://github.com/ehteshamz/greetings.git
>git tag -a v0.0.1 -m "The first release"
>git tag
v0.0.1
>git push origin v0.0.1
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 579 bytes | 579.00 KiB/s, done.
Total 5 (delta 0), reused 0 (delta 0)
To https://github.com/ehteshamz/greetings.git
 * [new tag]         v0.0.1 -> v0.0.1
```

*get*ting and using the greetings package

- Lets create another directory for the main file that uses this package and add the appropriate content.
- With the go modules we need to again initialize and the choice of name is less important, as it is main package.

```
>mkdir callgreetings
>cd callgreetings/
>pwd
/Users/ehteshamzahoor/go/src/github.com/ehteshamz/callgreetings
>touch callgreetpackage.go
>go mod init somename/somepackage
go: creating new go.mod: module somename/somepackage
go: to add module requirements and sums:
        go mod tidy
>
```

*get*ting and using the greetings package

- We add the following code for invoking the package

```
package main

import (
        "fmt"
        "github.com/ehteshamz/greetings"
)

func main() {
        fmt.Printf(greetings.PrintGreetings("ez "))
}
```

go tools at atom are not happy

```
package main

import (
  "fmt"

  "github.com/ehteshamz/greetings"
```

go build

```
no required module provides package
github.com/ehteshamz/greetings; to add it:
 go get github.com/ehteshamz/greetings (build)
```

vet

```
no required module provides package
github.com/ehteshamz/greetings; to add it: (vet)
```

Adding dependency

- We have not added dependency for our main package that it needs the greeting module.
- This can be done by editing the go.mod file or just running go get in the same folder
- You can notice that we are adding version info with the package and how the contents of go.mod change

```
>cat go.mod
module somename/somepackage

go 1.17
>go get github.com/ehteshamz/greetings@v0.0.1
go get: added github.com/ehteshamz/greetings v0.0.1
>cat go.mod
module somename/somepackage

go 1.17

require github.com/ehteshamz/greetings v0.0.1 // indirect
```

Adding dependency

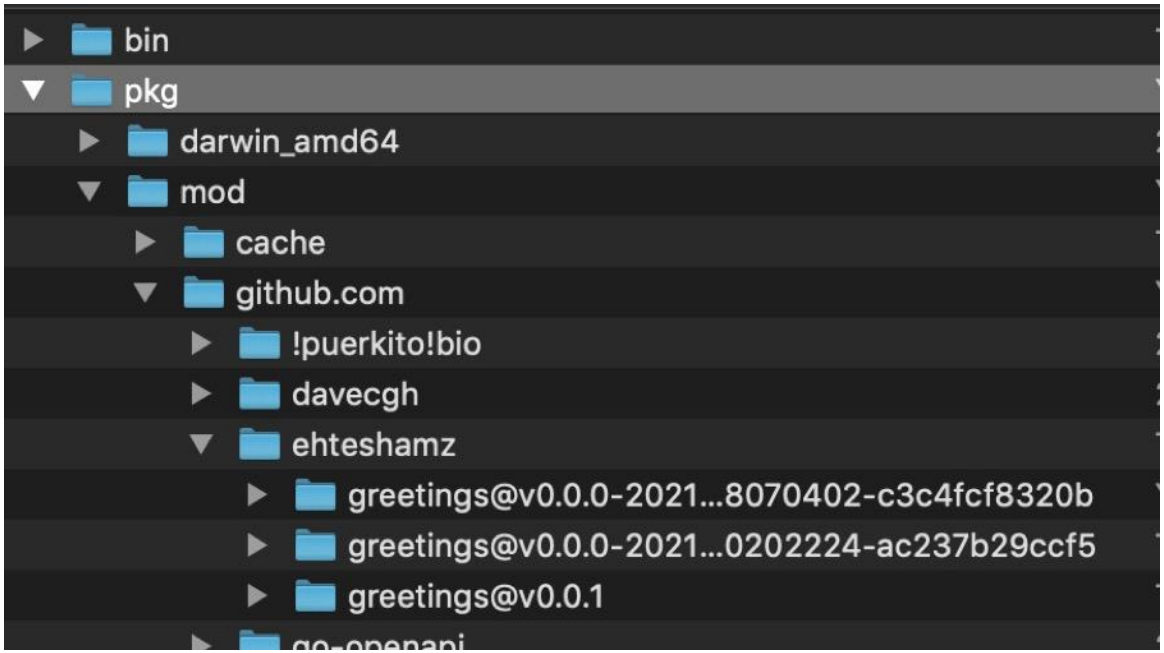
*If you receive an error while running the go get command with text including “incorrect version” and ”could not read Username” you need to set an environment variable

```
go env -w GOPRIVATE=github.com/yourusername
```

What does it say? It says our repository is private and go should not use the proxy service (goproxy.io) check its validity by checksum.

Where the packages are downloaded

- No longer in the $GOPATH/src but rather in the $GOPATH/pkg
- Notice we have now multiple versions (downloaded earlier) of greetings package including the one needed



Running the code

>go run callgreetpackage.go
Hello World-ez >

- For Atom, the *script* package can now run go code directly. Use it instead of *build*

So far so good but …

- What if we change out local greetings module to say Bonjour Monde instead of Hello World?

```
//Package greetings shows the greetings
package greetings

//GreetingsString is a global variable
var GreetingsString = "Bonjour le Monde"

//PrintGreetings is a global function
func PrintGreetings(name string) string {
        return GreetingsString + "-" + name
}
```

Working locally – the (very) wrong way

- Running the *callgreetpackage.go* again doesnt reflect the change ☹
- We can again publish this on the github say v0.0.2 and then go get, update go.mod and use the newer version.

Working locally - the right way

- We can add a *replace* directive at the go.mod (for the callgreetings module).

```
module somename/somepackage
go 1.17
replace github.com/ehteshamz/greetings => ../greetings
require github.com/ehteshamz/greetings v0.0.2 // indirect
```

- What does it say?

Greetings in Pashto ☺

- We change the local version to this but do NOT push this to github as say v0.0.3

```
//Package greetings shows the greetings
package greetings

//GreetingsString is a global variable
var GreetingsString = " سلام نړی "

//PrintGreetings is a global function
func PrintGreetings(name string) string {
        return GreetingsString + "-" + name
}
```

Running the main package …

go run callgreetpackage.go
ez-سلام نړی