About the Scott Text: Scott mentions a LOT of PLs. I will only discuss Go, Elixir, Java in detail and will only expect you to know about those. Also probably mention C and Python. (So if you find yourself reading about the details of C++, red this only for key concepts, not details.)

Lec 3: Go Intro
Why? C was designed in 1970 with those machines in mind. Go is C - 40 years later. Biggest change — no explicit memory management (malloc and free).   Rather more java-like with new and **garbage collection**.

Green comparison between Go and C
      numbers from whiteboard.
      factoring:
            C to 1,000,000: 1.8sec
            Go to 1,000,000: 4.6 sec

Go (in PL jargon):
      imperative
      statically scoped
      functions are first class
      static variable types
      strongly typed
      pass-by-by-value
      return-by-value

Writing Go:
Put every different go program in a different folder.
put program files in files that end in .go

in the main directory for a program (you will usually only have one directory)
     go mod init aaa/bbb
aaa/bbb does not matter. (This is used in large team development)

```
package main     // REQUIRED
import "fmt"     // won't compile unless imports exactly match
uses (unlike java).
func main() { // the function to start the program.  Should be
exactly one instance of a main function in a directory
    fmt.Println("hello geoff!") // Do something!!!
}
See hw.go
```

Note semi-colon allowed but not required
     Good/bad/yawn?

Once you have a program file:

```
     go run xxx.go
OR
     go build -o xxx xxx.go
     xxx
```

Go has lots of packages. We will discuss later. (VSC will automatically add imports.)

```
Variables
     lots of types :: usually you do not need to know.  Go
figures it out
     var i = 0
   var i int
     var i int = 7
     i := 0
These are all mostly equivalent.  Go initializes all integers to
0 (second case). (All types have a "zero" value. Go figures out
that i is an int (first and third). := gives "short form"
initialization … "=" does assignment ":=" does initialization
and assignment

Type Coercion:
     in Java
          int iinntt = 7;
          long lloonngg = 7;
          boolean bboo = iinntt==lloonngg;
     perfectly legal

No type coercion in Go
     var i int16 = 7
     var j int32 = 7
     kk := i==j  // not allowed, will not compile
So need to cast
     kk := i==int16(j)
Why does go not have type coercion??
```

Go uses value model of variables (as does Java for primitive
types). As does C. So like C, go has pointers and the
complexities of referencing and dereferencing pointers. Will
talk about this in ch 6. Unlike C, go has garbage collection
(more on that in ch 8.5.3)

Go uses pass and return by value
see pbv.go

fmt.Printf
%v   the value in a default format
      when printing structs, the plus flag (%+v) adds field names
%t   the word true or false
%d   base 10
%f   decimal point but no exponent, e.g. 123.456
%s   the uninterpreted bytes of the string or slice
\n CR-LF

Statements and Expressions
      statement = done for side effect only (eg print statement)
no return value
      expression = may have side effect but also returns a value

      in Java ++ is an expression   (j++)
      so order/Order.java compiles and runs. (what is printed?)
      In Go ++ is a statement. So the equivalent program does not
compile
            Was this a good decision by Go designers?

Strings UTF-8
      1-4 bytes to encode a character
      1 byte for ASCII chars
      if start with: 0 then 1 byte
                                  only 7 usable bits
                           110 2 bytes each byte begins 10
                                  so only 11 usable bits
                           1110 3 bytes then each byte begins 10
                                  so 16 usable bytes
                           11110 3 bytes, each byte begins 10
                                  so 21 usable types

"Tuple Assignment" and tuple return from procedure
```go
package main
import "fmt"
func main() {
    j,k := 5, 20  // initialize j and k
    fmt.Printf("j:%3d  k:%d\n", j,k);
    k,j = j,k // swap j and k    uses only one line!!!
    fmt.Printf("j:%3d  k:%d\n", j,k);
    l, m := mul(j,k)  // call function and initialize l and m
for return values
    fmt.Printf("l:%3d  m:%f\n", l,m)
}
```

```
so swap
     b,a = a,b

/**
 * do something
 * @param i an integer
 * @param j an integer
 * @return an integer and a float32
 **/
func mul(i , j int) (int, float32) { // return two values
    ii := i*j;
    jj := float32(i) / float32(j);  // casting
    return ii,jj


}
```
See tupl_go/tuple.go


```
if and for
no parens required, must have {}
package main
import "fmt"
func main() {
    ii,f1, f2 := 0,1,1
    for { // Go does not have a while loop!   Just for with
nothing (or ;;) No Parens MUST {}
        ii++;
        f1,f2 = f2, (f1+f2)
        if f2<0 {  // no parens must {}
            break
        }
        fmt.Printf("%d  %d %d\n", ii, f1, f2)
    }
}
```

Scope — very much like java   We will discuss scope in great
detail

arrays and slices
arrays — homogeneous collection with length **fixed at compile
time spaces for all arrays are allocated at compile time**
"the size of an array is a part of its type"
Arrays pass and return by value
        DIFFERENT FROM JAVA
        Like Java, an error to read off the end of an array

C does not throw an error in this case
        Reading off the end of the array is why Hoare invented
null
    see array_go/array.go

Also, Go had guaranteed zero assignment. Java "definite
assignment" errors, C whatever
     see Defin.java

slice – somewhat Java ArrayList
slice a **run-time allocated** piece of memory. When you make a
slice you really have a pointer a memory location.
see slic.go

also with slices you can get a piece
slice[start:end]
for example see remove fun in slice_go or slisli_go

When you pass a slice to a function, you pass the memory
location pointer.
see slifunapp

structs
much like java classes, with some different syntax.   Structs
can have methods!
     speed.go
Structs do "inherit" – somewhat
     – embedding (embed.go)
     – **static method binding** (funcbind_go/funcbind.go)
          contrast with Java funcbind_go/FuncBind.java

Program across multiple files
In same directory
UNIX> mkdir AAA
UNIX cd AAA
UNIX> go mod init GGT/AAA   /// GGT/AAA can be anything,
except .
UNIX> go run .   /// doc says to use everything in current
directory.   If explicitly name file, then all you get is that
file

```go
// a.go
package main

func main() {
    println("aaa")
    aaa("bbc")
}
```

```go
// b.go
package main

func aaa(bbb string) {
    println(bbb)
}
```

Encapsulation and multiple directories:
Everything in a package is public to everything in the same
package. In other packages, capitalization indicates public to
other packages. See encap_go
Also note that fmt.Println, fmt is initial cap, hence is is
public from the fmt package.

Command Line Arguments
NOT in main function (a la Java / C)
rather in os package
see comlin_go

      Advantages/disadvantages?

Generics — new in Go 1.18
Later