Scott Ch 3
"Names, Scopes and Bindings
Not section 3.4 — in general skip content of book that has anything to do with implementation

Binding times
- language design
- language implementation — integer precision is specified by language
  - C=No; Java,Go=Yes
- program writing
- compile — layout of static memory, etc
- link — separate modules come together
  - go — imports
- load — memory layout on machine
- run

For example, in Java .. consider the program **static_java/Sttc.java**. What do you expect the output to be.
- implementers of JVM made a choice for speed to statically allocate integers -128—127
- How does this improve speed???
- Note: this can be changed to increase size of cached ints
  - java -Djava.lang.Integer.IntegerCache.high=1024 Sttc

Each important. Each has effect on everything. Discuss
Early binding == speed. late binding==flexibility
- Early=C,Go,Java   Late=Python (and interpreted langs), Lisp, Elixir
- Also, early binding allows code analyzers (e.g. compiler) to detect issues before run time.

static vs dynamic :: usually static == fixed at compile time and dynamic==changeable at run time

Object lifetime
- following book with use the word "object" to refer to a thing in memory
- lifetime == time between creation and destruction.
- 3 basic storage allocation mechanisms
  - static
    - globals exists as long as program exists
    - limited by space on device.
    - lifetime - life of program
  - stack
    - An area of memory for holding the set of currently active functions.
      - a single function may be on the stack more than once
      - Functions are represented on the stack by a "stack frame"
    - stack is literally a stack of stack frames
      - What is a "stack frame"?
        - A thing that exists as long as the function is "running"
        - Could be much longer as a result of closures and first class functions
        - contains: variables in scope in function
        - contains: pointer back to the calling frame to the place where the function was called

    - Java thread call stack size 1MB
    - C: given (and changable) by "ulimit -s" default 8192

Go: “While the minimum stack size is defined as 2048 bytes, the Go runtime does also not allow goroutines to exceed a maximum stack size; this maximum depends on the architecture and is **1 GB for 64-bit** and 250MB for 32-bit systems”

Recursion depth: depends on stack size

See **static_java/RecursionTest.java**

Java 1M=~10000

heap

space limited by space on machine

THIS IS NOT the Data structure for priority queues and heap-sort

lifetime == from explicit creation until either explicit destruction or GC

If there is no automatic garbage collection Objects allocated from heap have no necessary way in which they are de-allocated. Memory leaks.

Java — “new” allocates memory from heap. Has GC

Garbage collection or not — just mention

C malloc and free. No GC.

Go make(). Has GC

See L04/life_go

stack allocation and recursion

tail recursion special form that can be dome without allocating / deallocating a new stack frame so much quicker. We will return to this in discussion of recursion

Scope

“The textual region of a program in which a binding is active”

Alternately “a scope is a program region of maximal size in which no bindings change”

NOTE — this is related to , but distinct from , lifetime

static — almost every language and probably any language you encounter

so called because the scope of every var can be determined at compile time.

when you go into a function, the variables “in scope” are globals plus vars in fun

Note” Static” here is NOT same as java static

dynamic — vars available depend on EVERY function on the stack

write quick example on board

nested subroutines

Java does not allow, but Java does have nested objects that present many of the same issues

Go allows nested funcs but with syntax change

cannot do “func a() rtn {}”

can do “a:=func () rtn {}” or “var a= func() rtn {}”

NOTE: outside a fun can declare a function

“var a = func() rtn {}” or “func a() rtn {}”

BUT NOT “a := func() rtn {}”

Blocks — in many languages denoted by {}

blocks define another scope

Javascript

Block scoped variable

function scoped variable

global scoped variable

Q: for a var defined within a block, what is its scope

whole block? Only after it appear within the block?

Blocks can nest. What happens with same var name in nested blocks

Java — NOT allowed

GO — nest2_go

Declaration order — does a block scoped variable exist everywhere within its block?
This is especially a problem for recursive structures (linked lists, trees, etc)
If name is not known throughout block, then how can item refer to itself?
declaration vs definition.

Name Meaning
alias — single object with multiple different names
aliases require a reference rather than a value
Go uses value model but make in Go returns references
alias_go
alias works because slices are actually pointers to a memory location , so a and b point to same thing

polymorphism — single name — multiple objects
Overloading
+ can be applied to lots of things
some langs allow program to add new capabilities
function names following is legal in Java but not in Go
```
func a(i int) {
	fmt.Println(i)
}
func a(i int, ii int) {
	fmt.Printf("%v %v \n", i, ii)
}
```
What does java do to make reuse of function names legal/possible??
Why does Go not do this??
“””Method dispatch is simplified if it doesn't need to do type matching as well. Experience with other languages told us that having a variety of methods with the same name but different signatures was occasionally useful but that it could also be confusing and fragile in practice. Matching only by name and requiring consistency in the types was a major simplifying decision in Go's type system.”””

Generics
Note that overloading must be resolvable at compile time
Java
Hidden variables.
name reused in enclosing scope.
Java does not allow in functions but you can get this with inheritance
see static_java AA and AB functions pp, p3 and p4

Closures
“A closure in a lanugage with static scoping captures the current instance of every object at the time the closure is created”
Closures still apply with recursion, but don’t go there if you can avoid it.
closures only apply in languages that allow nested functions and functions that can be returned from other functions.
NO Java
YES Go
see closure_go
Extent!!!
with closures …

you need to know not just if a var is in scope, but if it can ever be in scope again.

in java, scope and extent are same — because java does not have closures

in Go, a var defined on the stack can live on as a result of closure, so while is scope is static and known at compile time, extent only be known at runtime

see closure_go

- closures and extent apply all over the place!!!

First class:

- value can be:
  - passed as param
  - returned from function
  - assigned to a variable.

Second class

- only passed as param

Third class

- None of these
  - Java!!
    - BUT “object closures”
      - see ObClo_java

Lambda expressions — another day