Topic 8: Types
Ch 7 Scott

2 basic questions : what / why
What??
- bits are untyped!!!
- most basic: a type defines how many, and how, to interpret bits. (OK, so how does elixir have unlimited size integers?) Similarly, in any language, if a string is a “basic” type, how because you do not know its size
- also—the set of operations that are allowed it.
  - primitive types “built in” — usually at hardware level
    - different from Java int, …
  - composite types

Why?:
1. Types supply context — Useful for compiler as it specified what to do
2. Limit what is allowed to be done
3. Make the program more readable to user — effectively a form of documentation — especially useful when there are a lot of types (OO langs). So why type inference (as in Go)?
4. Compile time optimization

Most of these are arguments in favor of static types, What about languages (elixir, python) with dynamic types   point 2 is still valid.

Type system:
- 1. mechanism to define types
- 2. Definition of
  - type equivalence
    - structural vs name
  - type compatibility
    - what is allowed with what
    - for + suppose one is Int, what is the other allowed to be
      - in a weakly typed anything
      - Go, Java
  - type inference (may not be available in some langs)
- Terms
  - static vs dynamic type
    - Elixir: is it really dynamically typed since immutability means that the storage location changes.   Simulate immutability in Go? Test Question??
  - strongly typed
    - See below

“primitive types” vs composite types
- composites in next chapter
  - struct, array, set, pointers, list, file
- Primitive — int (at what precision?) should a lang care about precision?
  - character? ASCII, 16-bit ascii? rune? UTF-8
- enums — primitive or composite.   Why????   How??
  - consecutive integers?   Powers of two?

Do functions have types?
- Why?

If they are first or second class, they do / must
What is the type of function??
Go:
```
type af func(a int) int
func(incr int) int { return aa + inc }
```
Elixir:
late binding / dynamic type.   The only thing you know is the number of args.   And that is the type!!
```
iex(2)> h String.split
                    def split(binary)
  @spec split(t()) :: [t()]
delegate_to: String.Break.split/1
```
Java— function type is its name and all of the types of its arguments

Strongly typed — language prohibits even trying to do something that is not allowed for a type. Thrown out at compile
Weak—usually implies doing more work at run time — strong==fast
for instance, to make the "+" work, javascript must do what?
can interpreted language be strongly typed?
realistically this is a spectrum. Language may have holes …
weakly typed —ex language allows application of operators when it does not make necessarily make sense.   For instance, javascript is weakly typed (and dynamically typed)
```
f = some function
q = 5 + f
```
Go? Elixir?   Javascript?
Statically typed — strong AND type checking is a compile time.

Polymorphism
Ad hoc polymorphism
2 modes:
A. Overloading: e.g. + works on int and float
B. Subtyping — common in OO languages — allow uses of subtype where base type is specified.
Parametric
same function can be used for different arg types
Generics == "Explicit parametric polymorphism"
implemented at compile time!!!
In strongly typed language generics are only way to get polymorphism (except subtypes)

Lots of types
Basic type: integer, float …
Intergers
Java: byte, short, int, long. Also, Byte, Short, Integer, Long, BigInteger!!!
Elixir: integer
Go: [u[int[8,16,32,64]
Why so many int types???
Floating point: similar
char — what is a char?
one byte — ASCII
char in c

- 2 bytes — UNICODE16 — JAVA
  - char in Java
- Go does not actually have a char type it has a “rune”
  - WHAT IS A RUNE IN GO?
  - Up to 4 bytes — UTF8 —- variable
    - 0xxxxxxx — 1 byte — plain old ASCII
    - 110xxxxx 10xxxxxx —-
    - 1110xxxx 10xxxxxx 10xxxxxx
    - 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
    - “rune” in Go

- is String a basic type?
  - in Java? C? Go?
    - Java — NO..it is a class
      - (Are classes in java.lang really “basic” to Java??
      - You cannot do ANYTHING without java.lang.Object
      - To know would have to look at implementation of String class
    - C — definitely NOT
    - Elixir: YES
    - Go — from book “a string contains an array of bytes that, once created, is immutable”
      - This indicates that string is a composite type, maybe
      - Going further Go explicitly mirrors string functions with byte array functions
      - OTOH — “The underling type of every constant is a basic type” boolean, string or number”

- Enumerated types
  - What: a type that has a specific, finite (usually small), and bounded set of possible values.
  - Why?
  - Go: **enum_go/enum.go**
    - They do not really exist like in other languages so you get little benefit
  - Java: **enum_java/GTEnum.java**

- Type checking
  - Java: obvious and handled by compiler
  - Go: often do not require explicit types (type inference)
    - type inference
      - why have type inference?
        - you loose the readability of the implicit documentation
        - what do you gain?

When are two types the same???
- structural vs name equivalence
  - structural
    - same order, or just same number and kind?
    - what work needs to be done to get this?

what does Go/Elixir do?
why not use structural equivalence?
name
what about type aliases?

what are Go, Java
Go: **equiv_go/equiv.go**
strict name equivalence
Java: no typealias (quite) **equiv_java/Equiv.java**
you can define a class that extends another class without addition
Why would you??
limitation — class cannot be final (e.g. String is final, why?) what is final with respect to classes in Java?
Also this does not really get you equivalence
Elixir — structs are a form of type — sort of.

Casting — converting from one type to another
in strongly typed languages "weird" casts are not allowed
GO: **casts_go/casts.go**

```
func t5() {
   str := "abc"
   fmt.Println(str)
   var num  int64
   num=40
   fmt.Println(num)
   num = int64(str) // Compiler flags as not allowed
}
```

Problem is that casting requires changing bits and you have to know how.
what is the problem with changing bits???   time!
Some langs allow "non-converting" casts. That is, do not change bits just interpret bits differently. What is problem? (C does this)
Go: **pun_go/pun.go**

Question — can you do this in Java?? Why/why not??

type coercion
implicit casting????
allow 3+2.4 without explicit casing
pros/cons
Go — no coercion
Java — happy to coerce among numeric types
Javascript— (weak) happy to coerce pretty much anything
— "JAVASCRIPT WANTS THINGS TO BE TRUE"
Elixir — coerce between integer and float but not between integer and string
== vs === in elixir and javascript

```
iex(1)> a="12"
"12"
iex(2)> b=12
12
```

```
iex(3)> a==b
false
iex(4)> a===b
false
iex(5)> c=12.0
12.0
iex(6)> b==c
true
iex(7)> b===c
false
```

Type inference (in statically typed language):
- go does it:
  - **infer_go**
- type inference in Java??
  - does <> in some generics count as type inference??

Advantages / disadvantages of type inference (in a strongly typed language)???

Generics
- they are much more complex that you thought (and you probably thought they were pretty complex)
- Java “Generic Gotchas”
  - See the web article

Covariance & Generics:
- For example
  - Integer extends Number — True
  - By Covariance Integer[] extends Number[]
  - Hence this is legal:
    - Number[] nArray = new Number[10];
    - Integer[] iArray = nArray;
      - can put integers into iArray and it is guaranteed to be fine with nArray
    - See **ArrayCov_java**
      - point when passing into methods covariant type inherit just like their base types.   But this can cause issues at run time.
- generics are NOT covariant It would break type saftey
- For instance consider ArrayList
  - ArrayList<Integer> ai = new ArrayList<>();
  - ArrayList<Number> an = ai; // WILL NOT COMPILE
  - ln.add(Double.doubleValue(2.2));
- See also **Cov1_java**
  - (note arrays actually have the same issue)
- Generics with wildcards
  - see covar_java
  - see Wildcard_java
  - ArrayList<? extends Number>

ArrayList<?>
ArrayList<*>
Wildcards can be handy
limit a function to taking an array list that contains anything that extends number (you need it here because generics are NOT covariant)
But wildcards result in other issues, specifically immutability.
See **Immut_java**

Type erasure in Java
generics are known only by compiler, they are "erased" after compile so all of that info is gone at runtime.
see **Erasure_java**
EG
ArrayList<String> ss = new ArrayList<>();
eventually gets translated to
ArrayList ss = new ArrayList();
So at run time, anything that the compiler let pass is OK. It could cause runtime issues.
Erasure also causes things that might see legal to NOT be legal. For instance
public class JavascriptNumber implements Comparable<String>, Comparable<Number> { …}
does not work because compiler reduces this to
public class JavascriptNumber implements Comparable, Comparable { …}

Generics in Go
See **GoGen1** for basics
NO erasure in Go … see **GoGen2**
Any — kind of like Object in Java. More like ?
LinkedList is a good example, but not until next chapter!

Object equality (sec 7.4)
deep vs shallow equality
deep vs shallow assignment
in ref-model and value model languages
why in Go if == defined over array but not slice
"deep assignment"
== vs === in Elixir

When are two objects the same?
Deep vs shallow checks?
Java == vs equals
Deep vs shallow assignment
Only applied to reference model languages
see **copy_go**
Value languages effectively always deep copy
Shallow
copy and assign pointer (**SCopy.java**)
make a new copy of object and assign.