CS131 Week 4

Week 4 Lecture 1

==========

Types

Java
Parallelism

roles:
- annotation
  - for programmers
  - for compilers
- inference
  - can infer type from other info

time of type check:
- dynamic (more flexible) (simpler)
- static (more reliable)

strongly typed:
- cannot escape type checking

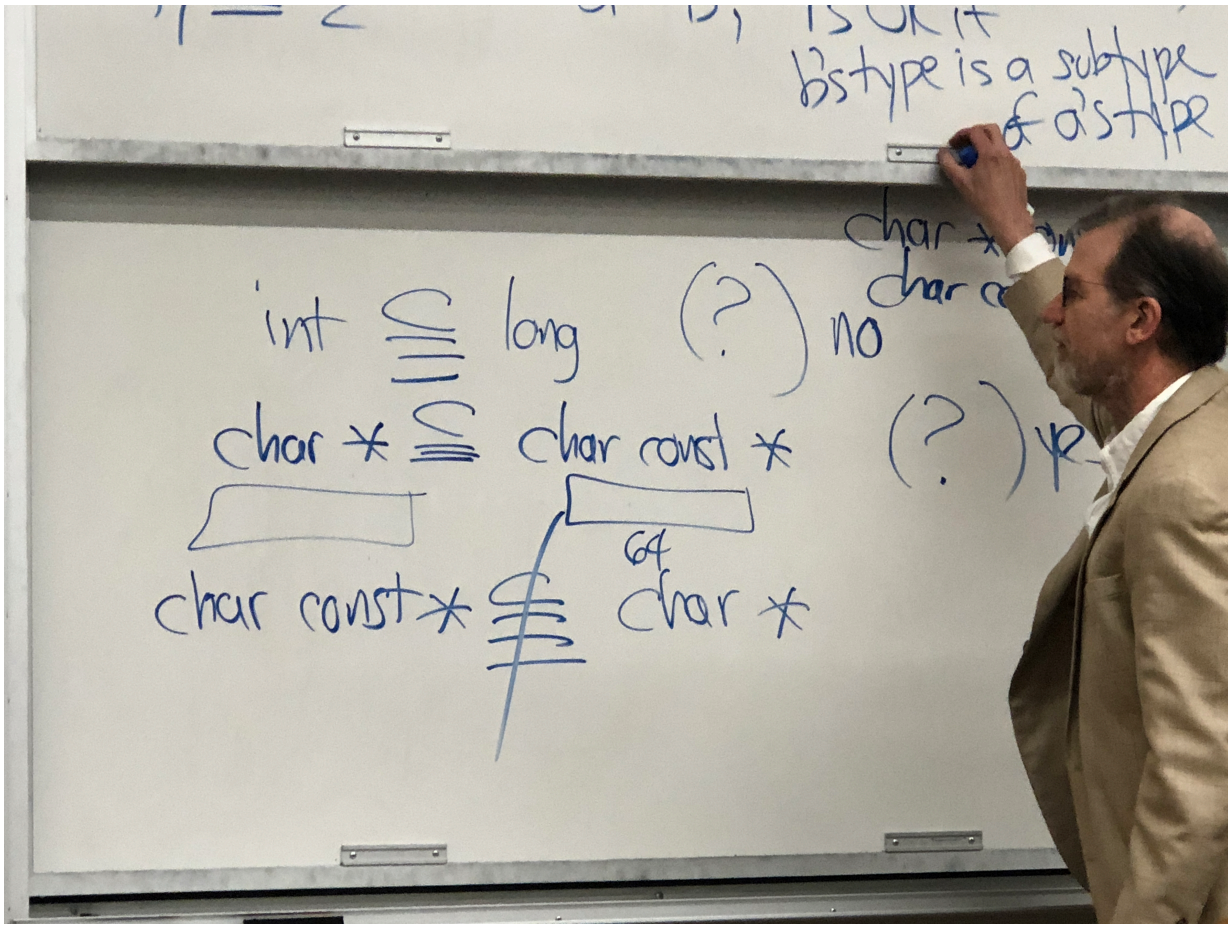
Type equivalence:
- languages with info hiding

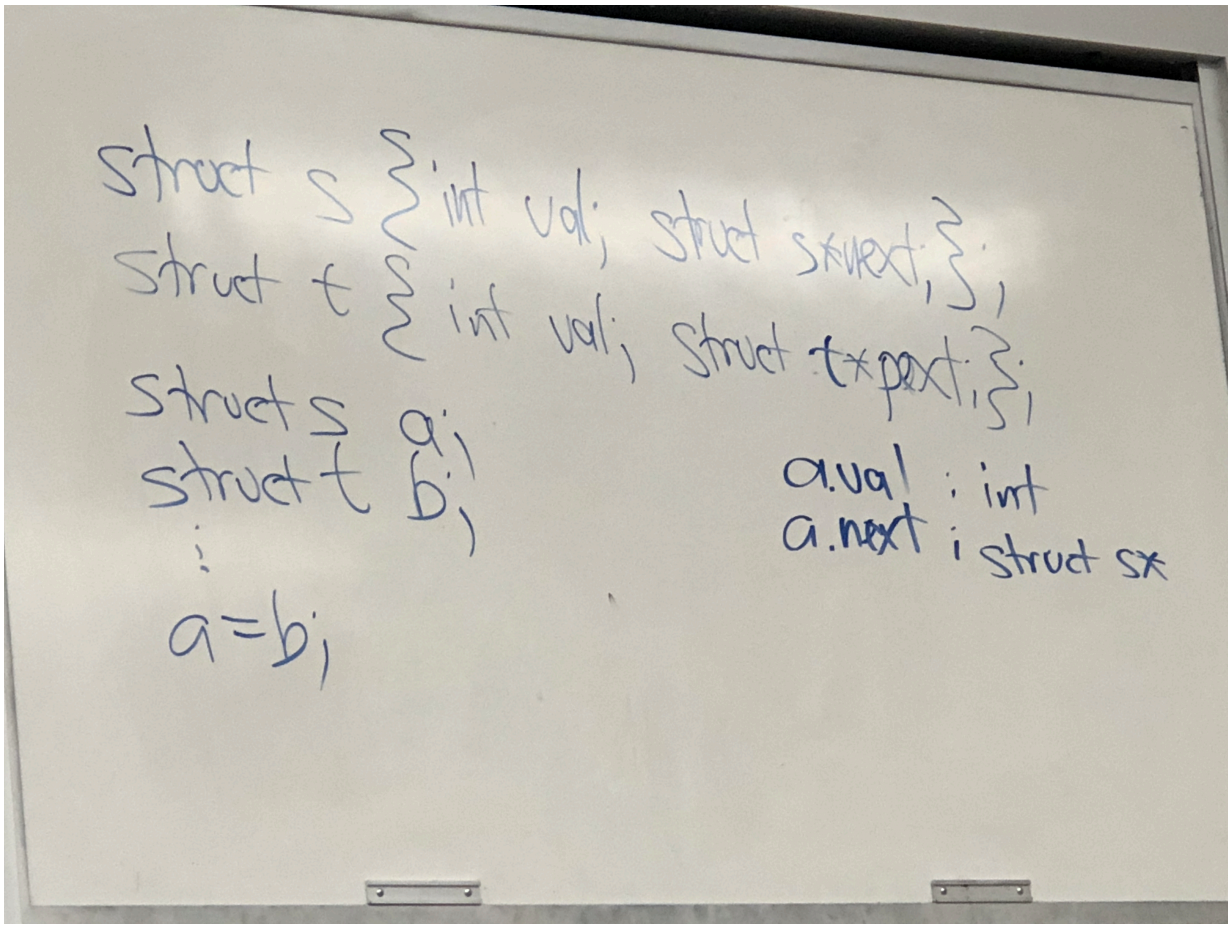
structural equivalence:
- types are same if layout (in memory) or operations are same
- Stack x; Set y; // both are same since appear same in memory, same ops

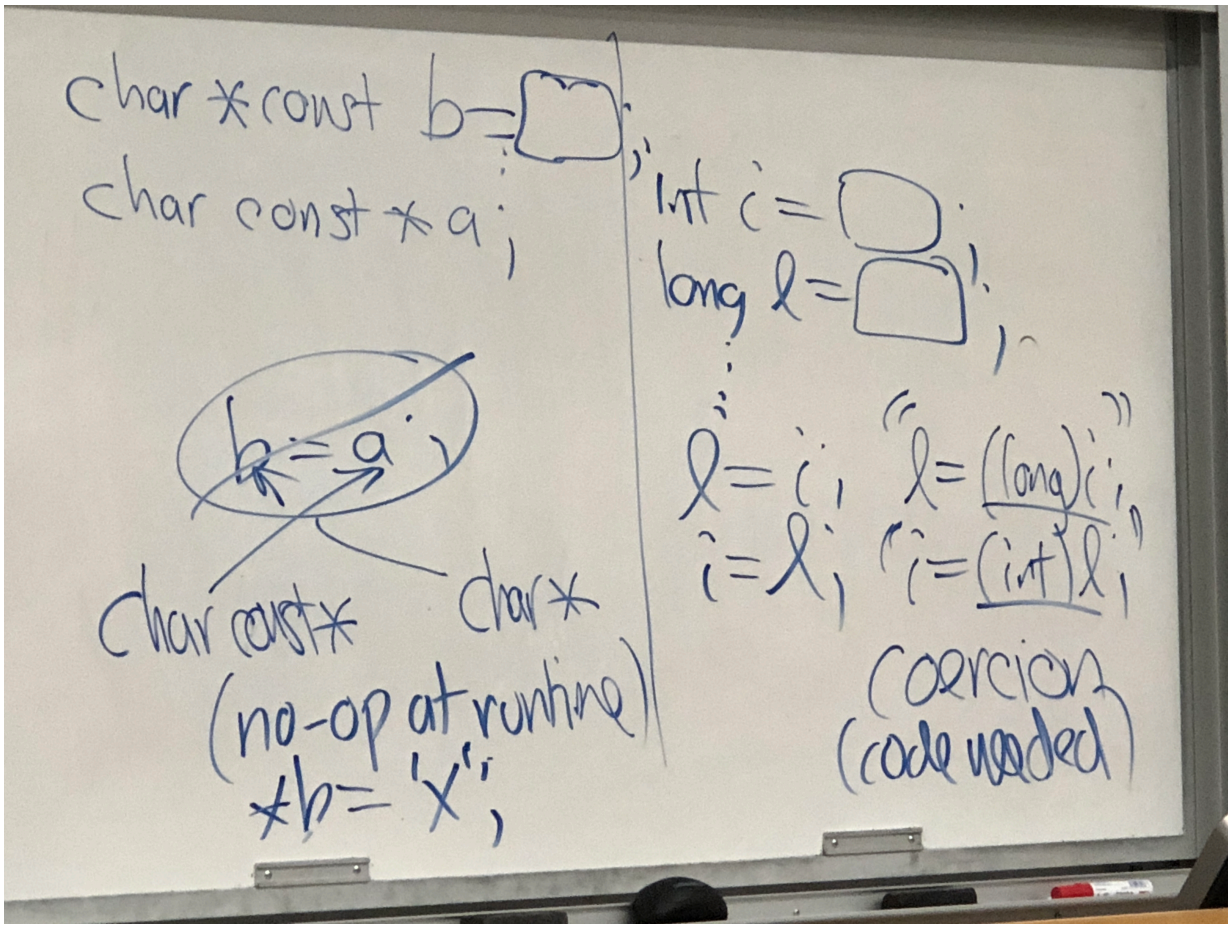
name equivalence:
- types with same name are same
- e.g. Stack x, y; // both x and y are same type

IMPORTANT:
Types are **SETS** AND **OPERATIONS** (done on sets)

Examples:
subclass has more operations than a superclass
char * vs char const *
- char * more complicated: allows more operations


b's type is a subtype
of a's type
int ⊆ long (?) no
char * ⊆ char const *
64
char const * ⊈ char *


struct s { int val; struct s *next; };
struct t { int val; struct t *next; };
struct s a;
struct t b;
:
a=b;
a.val : int
a.next : struct s*



Polymorphism & Java

- **ad hoc**
- **parametric**
  - **templates**
  - **generics**

Ad hoc polymorphism

Overloading: Identify implementation by looking at **arg types** (or result type)
e.g. sin(x) *depends on arg type

- float
- double
- long double

Name mangling: “flat” symbol table

- how we implement overloading using nonoverloaded system

sin $d	0x7f937c
sin $L	0x4f902
sin $f	0x6c93f

*technically, names are different (not all sin). They are MANGLED
*$d doesn't show up anywhere, source code can be the same

Coercion

- type cast (coerce int to float when multiplying them together)

Universal polymorphism

- infinite # of types: programmer specified
- **parametric polymorphism**
  - when a function's type contains a type variable
  - e.g. length: 'a list -> int

e.g.

```
// Without Generics
static void remove1(Collection c) {
	for (Iterator i = c.iterator(); i.hasNext(); ) {
		if ( ( (String) i.next() ).length() == 1) i.remove();
		//	^ ugly as shit (too many parenth)
	}
}

// WIth Generics
static void remove1(Collection<String> c) {
	for (Iterator<String> i = c.iterator(); ... ) {
		if (i.next().length() == 1) i.remove();
		// ^ easy to read
	}
}
```

Templates vs. Generics

With templates, compiler recompiles the code for each instantiation

- multiple copies of code
- e.g. one copy for Collection<String>, one for Collection<Thread>

With generics, just one copy of compiled code

- only one copy of compiled code, checked generically
- harder to type check generically

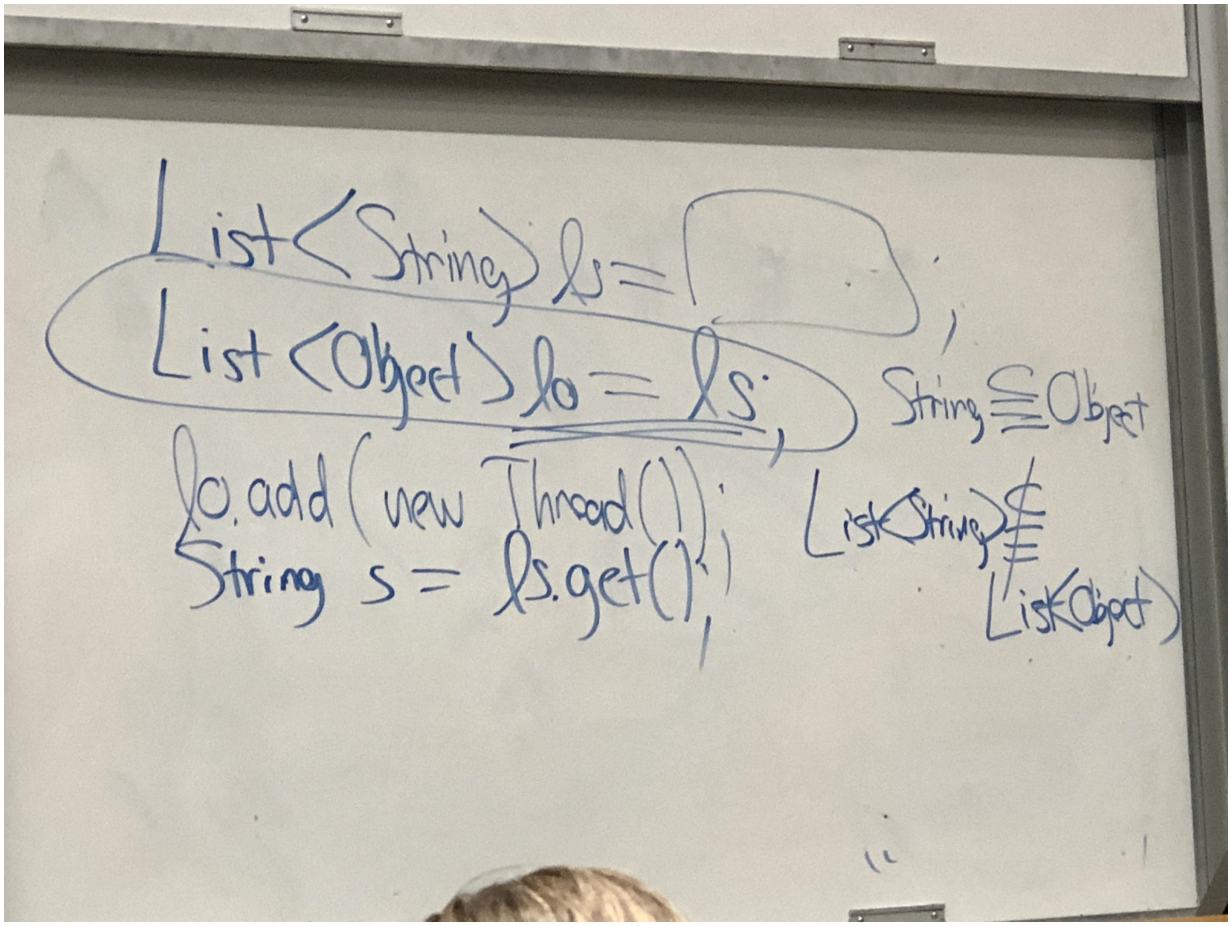
Java Generics and subtypes

```
List<Integer> l = new LinkedList<Integer>();
l.add(new Integer(0)) ;
Integer n = l.iterator().next();
```

Example of why types are SETS and OPS

Every line is valid except circled line:

- even though it makes sense List<String> should be subset of List<Object>, it IS NOT because operations you can do to List<Object> cannot be done to List<String> (e.g. add new thread()).



CODE BREAKS

```
void printAll(Collection<Object> c) {
    for (Object i: c) System.out.println(i);
}
```

printAll(cs) // collection of Strings WILL NOT WORK
- will not work because then you could get around the List<String> !< List<Object> requirement
- compile time error message

Therefore...
Wildcards

```
void printAll(Collection<?> c) {
	for (Object i: c) System.out.println(i);
}
/* or same meaning:
void<T> printAll(Collection<T> c) {
	for (T i: c) System.out.println(i);
}
*/
```
printAll(cs) // collection of Strings WILL WORK now

Bounded Wildcards

```
void<T extends Shape> printAll(Collection<T> c) {
	for (T i: c) outShape(i);
}
```
Forces things to be shape variants

```
static void<T> convert(T [] a, Collection<? super T> c) {
	for (T o : a) c.add(o) ;
}
/* or ...
static void<T, U> convert(T [] a, Collection<U super T> c) {
	for (T o : a) c.add(o) ;
}
*/
convert(String [], Collection<Object>)
```

Duck Typing

~~type duck~~

```
d.quack()
d.waddle()
```

Pros

- Don’t ask for type
- If methods work, don’t care about type
- i.e. System.out.println(something)
  - if it works, we lit
  - if it isn’t, give error
- Python uses duck typing

Cons

- runtime errors!!
- makes it harder to trace code

Parallelism

Sunway TaihuLight

- 40,960 nodes
- each contains a SW26010 processor (each 4 CPU clusters)
  - each 64 compute CPUs, 1 mgmt CPU
  - Therefore each node has 260 cores
- 10,649,600 cores in total
- 93 PetaFlops (93 * 10^15 FLOPS) (100 million x faster than laptop)
- 15.37 MW to run this computer
- 6 GFLOPS/W (US machines are more efficient)
- 1.3 PB RAM (petabytes)

— 1/30/18

Week 4 Lecture 2

==========

Parts I missed: