- 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

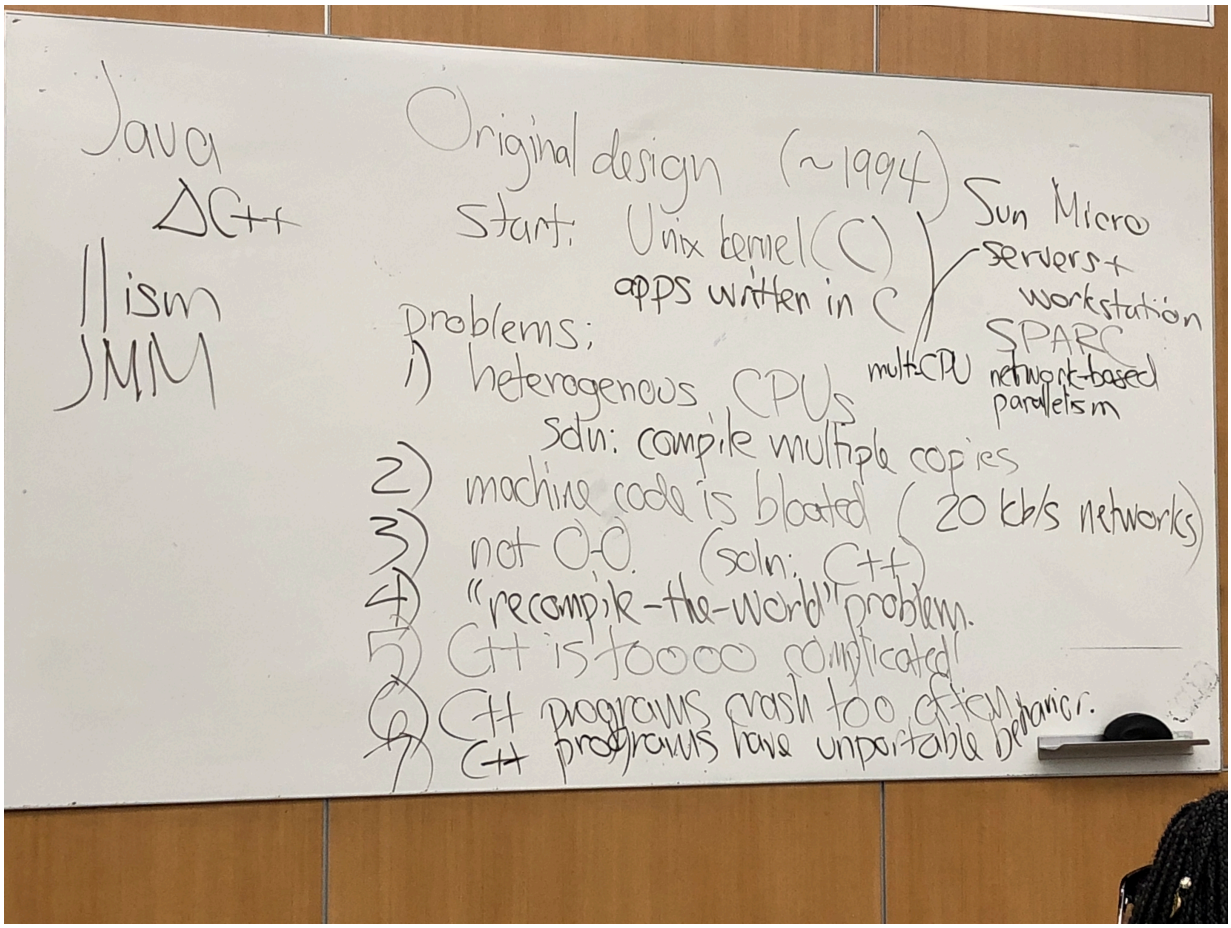
- 40,960 nodes
- each contains a SW26010 processor (each 4 CPU clusters)
  - each 64 compute CPUs, 1 mgmt CPU
  - Therefore each node has 260 cores
- 10,649,600 cores in total
- 93 PetaFlops (93 * 10^15 FLOPS) (100 million x faster than laptop)
- 15.37 MW to run this computer
- 6 GFLOPS/W (US machines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

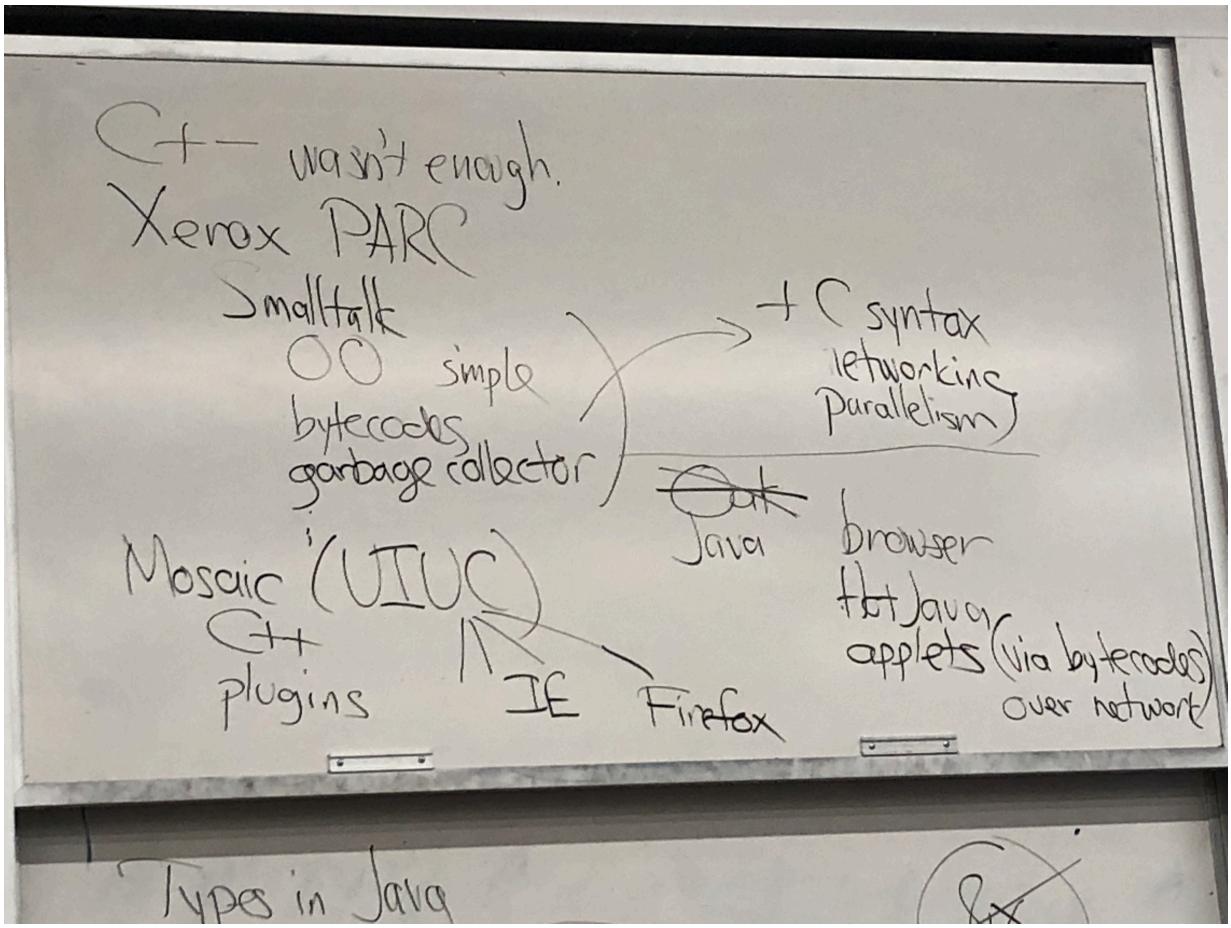
— 1/30/18

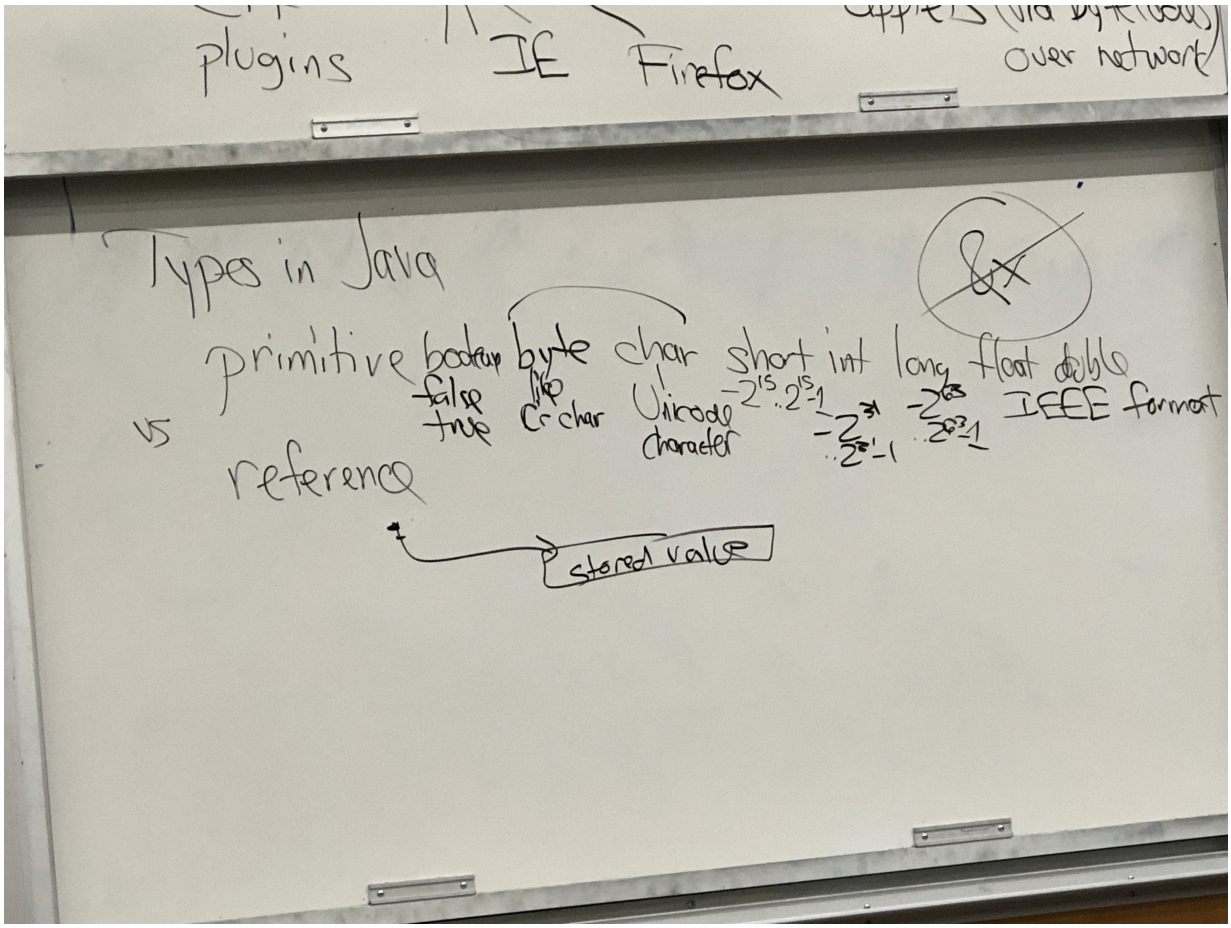
Week 4 Lecture 2

==========

Parts I missed:

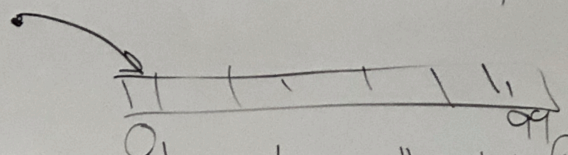

Java
ΔC++
||ism
JMM
Original design (~1994)
start: Unix kernel (C)
apps written in C
Sun Micro
servers + workstation
SPARC
multi-CPU
network-based parallelism
Problems:
1) heterogeneous CPUs
soln: compile multiple copies
2) machine code is bloated (20 kb/s networks)
3) not O-O. (soln: C++)
4) "recompile-the-world" problem.
5) C++ is toooo complicated!
6) C++ programs crash too often.
7) C++ programs have unportable behavior.


C++ wasn't enough.
Xerox PARC
Smalltalk
OO
simple
bytecodes
garbage collector
+ C syntax
networking
Parallelism
Oak
Java
browser
hot Java or
applets (via bytecodes)
over network
Mosaic (UIUC)
C++
plugins
IE
Firefox
Types in Java


plugins
IE
Firefox
over network
Types in Java
primitive
boolean
byte
char
short
int
long
float
double
false
true
16
Unicode
character
IEEE format
vs
reference
stored value

Arrays in Java are references

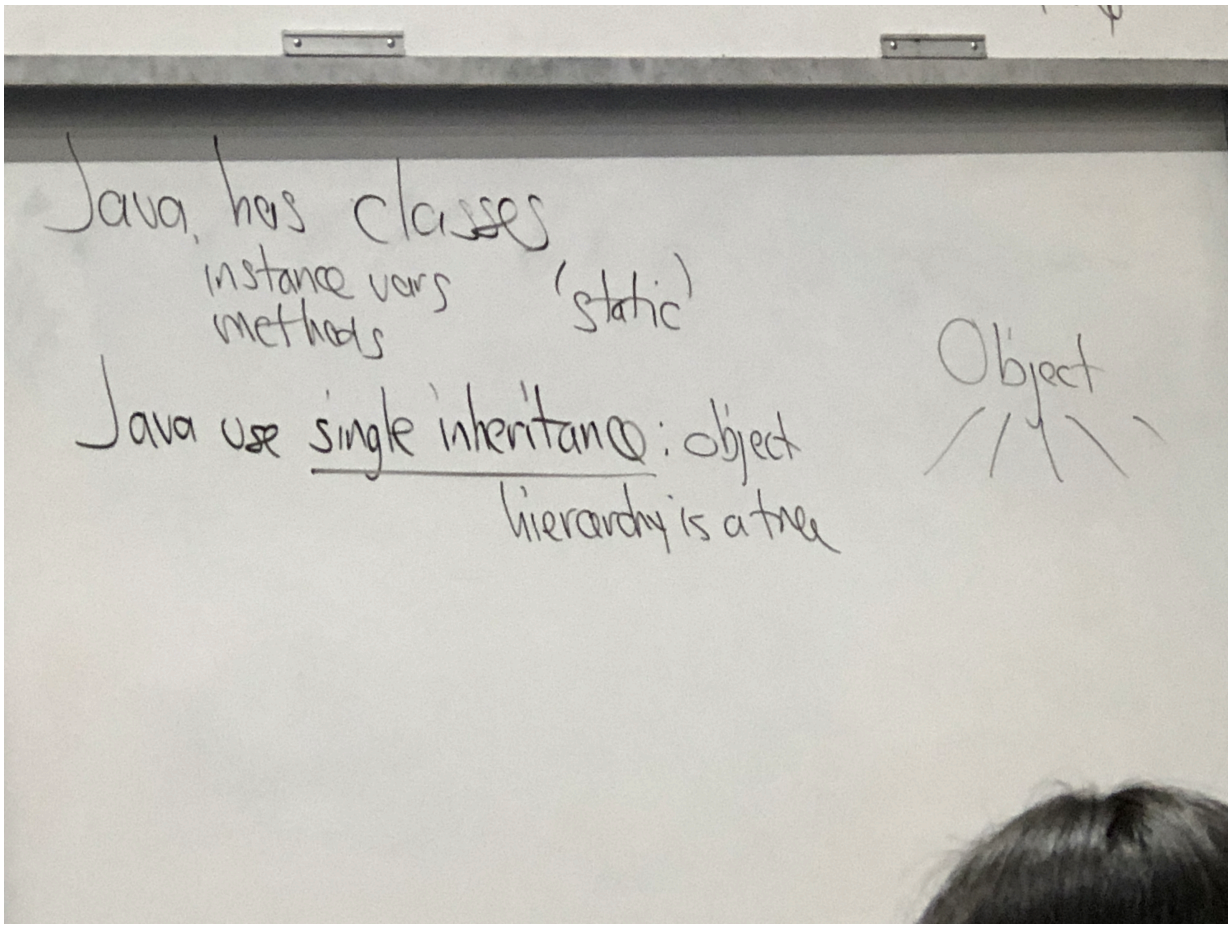
```
int[] foo = new int[x+3];
```

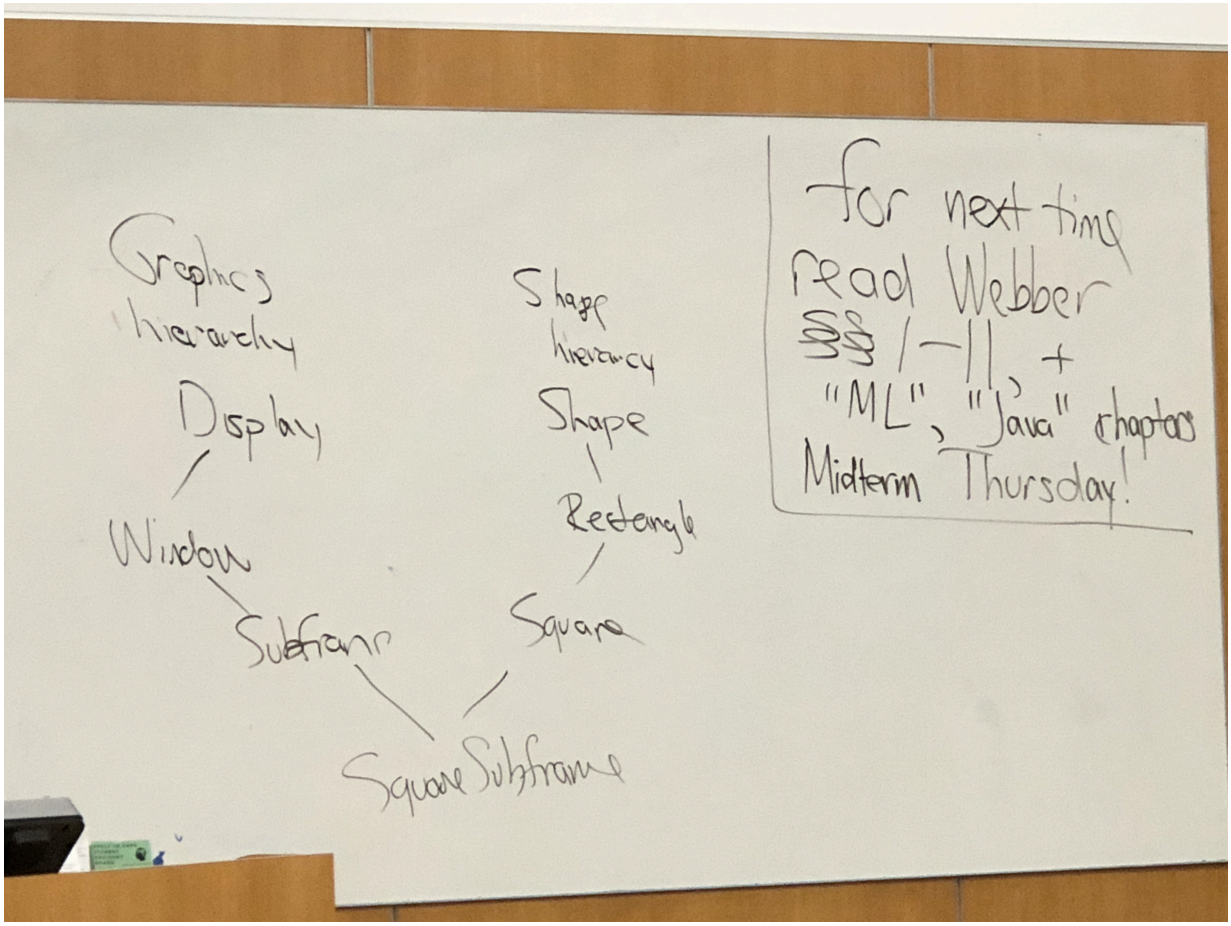


arrays are always heap-allocated (init ∅)
they have fixed size once allocated
you can return arrays from methods

```
a[i] = b[k+1];
```

Subscript checking is required


Java has classes
instance vars
methods
(static)
Java use single inheritance: object
hierarchy is a tree
Object



Java has classes:
- instance vars
- methods
- 'static' # a var not associated with any class

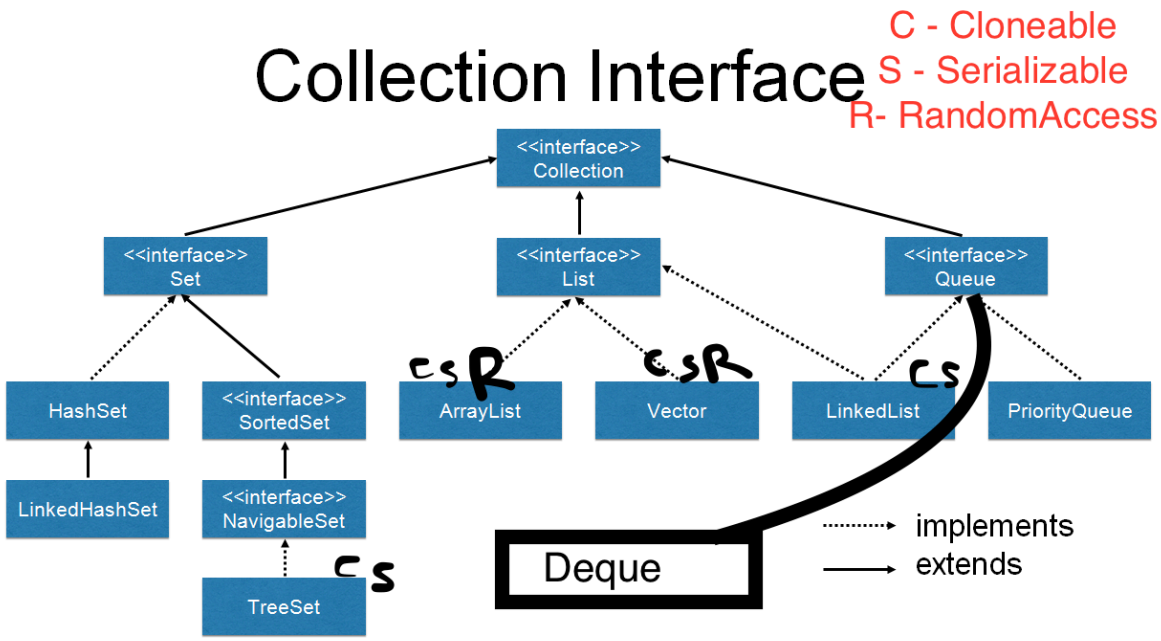
Java uses single inheritance object:
- hierarchy is a tree
- there is multiple inheritance though

Interfaces
Substitute for M.L.
i.e.
parent: define m(), n(), a()
interface: declare o(), p() # but no define
child: gets an inheritance that saves it work (its parents force it to carry out certain tasks, child has to implement these)
- define p() with implementation

class p {

```
    define m(), n()   <= (ordinary)
    abstract int foo();          <== declares but does not define
}
```

Abstract Class
A compromise between interfaces and superclasses




```
public class Object {
    public Object();
    public boolean equals (Object obj);
    public int hashCode();
        o1.equals(o2) -> o1.hashCode() == o2.hashCode();
    public String toString();  (default: ("#oxfffe0o"))
    public final Class getClass();
                ^ Class <? extends Object>
}
```

new Object()
(o1 == o2 bydefault) o1.equals(o2); o.hashCode(); "(int) o"
Class objects are runtime representation of classes

Final vs Abstract
Final is the exact opposite of abstract: if a superclass has the final keyword all base classes that extend the superclass must use the parent implementation: cannot redefine

Throw Exception
i.e. finalize() -> function called before object is about to die

protected void finalize() **throws Throwable**;
^ called by garbage collector before object is reclaimed

*protected means that this function is used by the garbage collector, and is called only right before object is about to die
^ is this really def of protected?
*protected is supposed to mean function/var can only be used by subclasses of current class

if your method can throw an exception, you have to delare that in the signature
this method is allowed to throw an exception that escapes the method itself so caller will have to worry
this is a signal to the programmar that i fyou call the finalize method you better be prepared to catch an exception

new Thread();
life cycle
- Thread t = new Thread();
- t.start() #allocates OS resources, invokes t.run()
- thread code itself can:
  - run
  - sleep
  - wait
  - I/O
  - exit (the run method)

— 2/1/18