

Inner Classes & the True Nature of Static

Administrivia

- Today: R2
- Next Wed (Mar 30): Quiz 3
 - Multithreading and synchronization
 - JDK `java.lang.Thread`,
`java.lang.Runnable`, `synchronized` keyword
 - Experiment a bit...
 - Know how to
 - Create and terminate new threads
 - What synchronization is and why you want it
 - Synchronized methods and blocks

Design tip o' the day

- Want a bug to show up as soon as possible
 - As close to the *creation* of the bug as you can
 - Bug should produce (highly) visible evidence
 - Helps debugging -- bug isn't hidden by many layers of propagation
- **Fail fast principle:** design so that bugs cause failures immediately -- as close to source as possible

Fail fast example

- Use *illegal* default values when you don't know the correct final value for a thing
 - `private int _arrMax=0;`
- vs.
 - `private int _arrMax=-1;`
- Also shows up in `java.util.*.iterator()`
- Most built-in JDK iterators are fail fast -- die if you change the collection out from under them

Inner classes

- Basic idea: class defined within scope of another class

```
public class MondoHashMap implements Map {
    // most of MHM body
    private class _MHMIter implements Iterator {}
    private static class _MHMEntry implements
        Entry {}
}
```

Benefits of inner classes

- Logical grouping of functionality
 - `_MHMIter` is only useful to `MondoHashMap`, so should be defined with (within) it
- Data hiding
 - `_MHMIter` is only *usable* by `MHM`, so should only be visible to `MHM`
 - Useful side effect: Avoids namespace clutter
 - Every class can have its own thing named `_innerIter` (or whatever)
- Linkage
 - `_MHMIter` has privileged access to `MHM`

Subtleties...

- Java officially recognizes 4 different types of enclosed classes:
 - Inner classes
 - Nested classes
 - Local classes
 - Anonymous inner classes
- Bleh...
- Real differences:
 - Is it `static` or not?
 - Is it declared within a *class* or within a *method*?
 - Does it have a name or not?

Usual case: basic inner

- “Basic” inner class: not static, class scope, has name
- Mostly like a normal class *but*:
 - “True name” is
 - `packageName.outerClass.innerClass`
 - E.g.,
`unm.cs351.s05.terran.p1.MondoHashMap._MHMEntry`
- If declared `private`, only visible to enclosing class
- Only enclosing class can call `new` to create one

Coolness of basic IC

- Very useful bit: Inner class has direct access to *all* of its outer class's parts
- Including `private` methods and fields
- Note: not true in reverse: Any inner object can be created by only *one* outer object, but an outer object can create *many* of the same inner object

```
Map m=new HashMap( );
```

```
Set s1=m.keySet( );
```

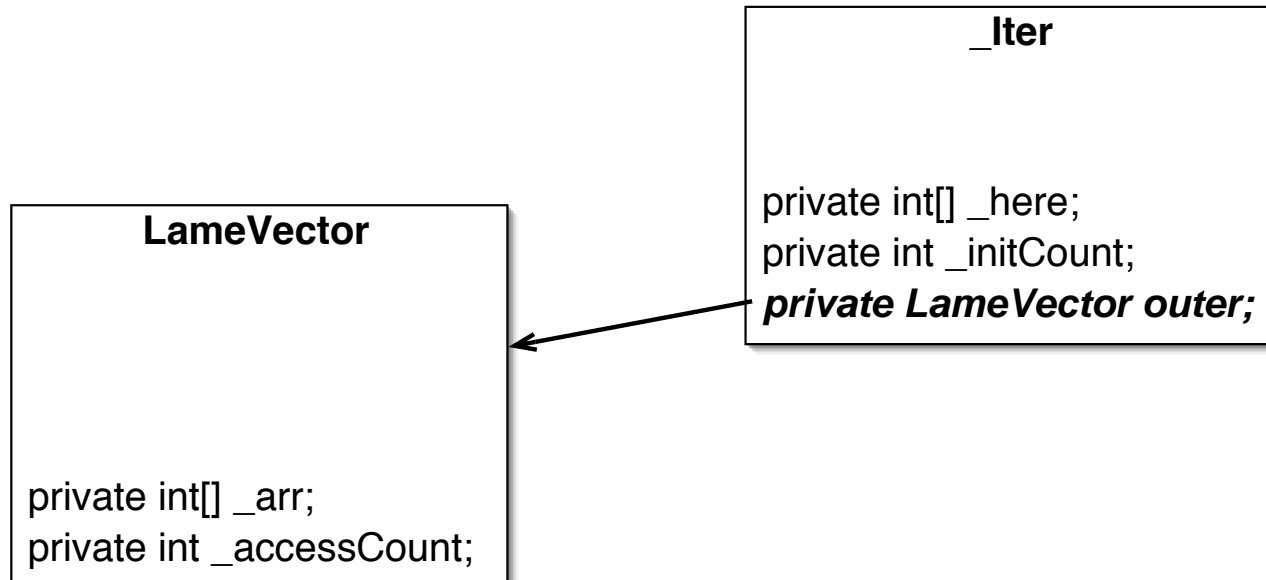
```
Set s2=m.keySet( );
```

```
Set s3=m.keySet( );
```

- `s1...s3` are all diff objects, but each derives from same `m`

How that works...

- Every inner object has a hidden pointer to its parent outer object



- In `_Iter.next()` Java transparently translates `if (_accessCount != _initCount)`
- to `if (outer._accessCount != _initCount)`

The catch...

- B/c every inner class is attached to its outer class:
 - *You must have an instance of the outer class in order to create an object of the inner class*
- E.g., must have an instance of `LameVector` to create an `_Iter`
- Makes sense -- would be meaningless to have the inner w/o the outer
- But...
- What about `Map.Entry`?
 - Does it *really* need to be tightly attached to the `Map` that created it?