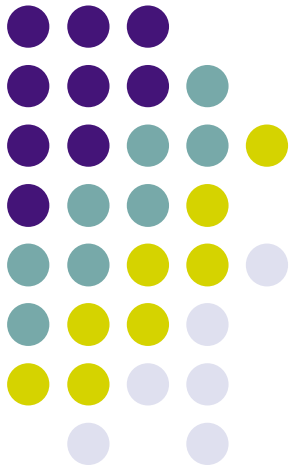
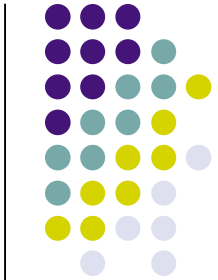
Day 3
Hashing, Collections, and Comparators

Wed. January 25th 2006

Scott Ostler

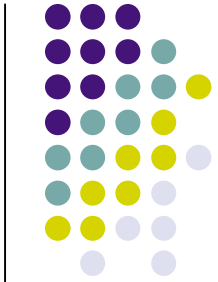


Hashing



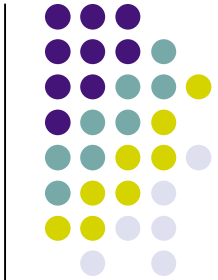
- Yesterday we overrode .equals()
- Today we override .hashCode()
- Goal: understand why we need to, and how to do it

What is a Hash?



- An integer that “stands in” for an object
- Quick way to check for inequality, construct groupings
- Equal things (should) have equal hashs

What is .hashCode()

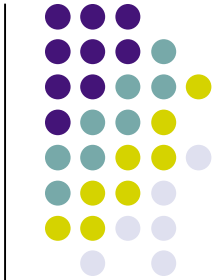


- Well known method name that returns int
- Is defined in java.lang.Object to return a value mostly unique to that instance
- All classes either inherit it, or override it

hashCode Object Contract

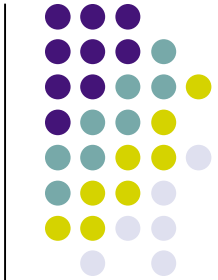
- An object's hashcode cannot change until it is no longer equal to what it was
- Two equal objects must have an equal hashCode
- It is good if two unequal objects have distinct hashes

Hashcode Examples



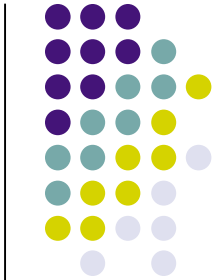
```
String scott = "Scotty";
String scott2 = “Scotty”;
String corey = "Corey";
System.out.println(scott.hashCode());
System.out.println(scott2.hashCode());
System.out.println(corey.hashCode());
=> -1823897190, -1823897190, 65295514

Integer int1 = 123456789;
Integer int2 = 123456789;
System.out.println(int1.hashCode());
System.out.println(int2.hashCode());
=> 123456789, 123456789
```

A Name Class with equals()



```
public class Name {
    public String first;
    public String last;

    public Name(String first, String last) {
        this.first = first;
        this.last = last;
    }
    public String toString() {
        return first + " " + last;
    }
    public boolean equals(Object o) {
        return (o instanceof Name &&
                ((Name) o).first.equals(this.first) &&
                ((Name) o).last.equals(this.last));
    }}
```

Do our Names work?

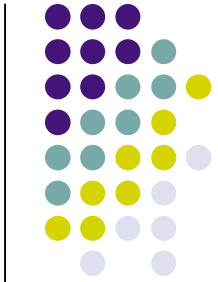


```
Name kyle = new Name(“Kyle”, “MacLaughlin”);
Name jack = new Name("Jack", "Nance");
Name jack2 = new Name("Jack", "Nance");

System.out.println(kyle.equals(jack));
System.out.println(jack.equals(jack2));
System.out.println(kyle.hashCode());
System.out.println(jack.hashCode());
System.out.println(jack2.hashCode());
```

⇒ false, true, 6718604, 7122755, 14718739

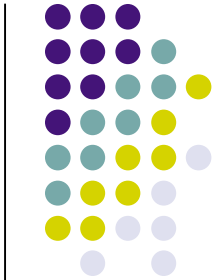
- Objects are equal, hashCodes aren’t

Who cares about hashCode?



- Name code seems to work
- Is this really a problem?
- If we don’t use hashCode(), why bother writing it?

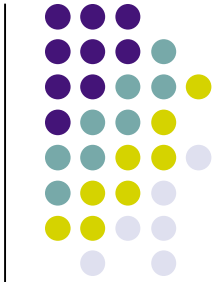
ANSWER: JAVA CARES!



- We have violated the Object contract

- We have embarked upon a path filled with Bad, Strange Things

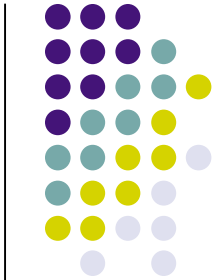
Bad, Strange Thing #1



```
Set<String> strings = new HashSet<String>();
Set<Name> names = new HashSet<Name>()

strings.add("jack");
names.add(new Name("Jack", "Nance"));

System.out.println(strings.contains("jack"));
System.out.println(names.contains(
                        new Name("Jack", "Nance"));
```

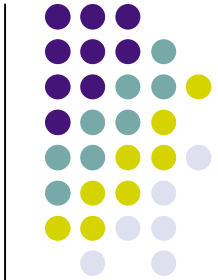
=> true, false

Solution? make .hashCode()



- Remember our requirements:
  - hashCode() must obey equality
  - hashCode() must be consistent
  - hashCode() must generate int
  - hashCode() should recognize inequality

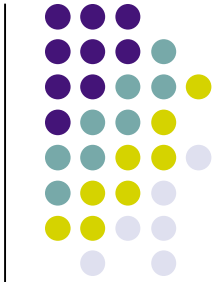
Possible Implementation



```
public class Name {
    …
    public int hashCode() {
        return first.hashCode()
                        + last.hashCode();
    }
}
```

- Does this work?

Good, Normal Thing #1

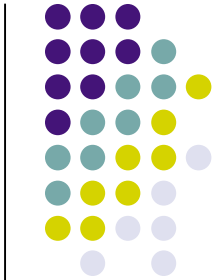


```
Set<Name> names = new HashSet<Name>()

names.add(jack);

System.out.println(names.contains(
                        new Name("Jack", "Nance"));
```

⇒ true

- Could it be better?

A Better Implementation



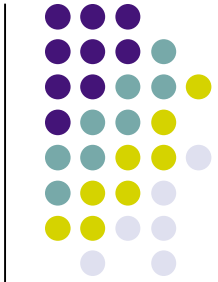
```
public class Name {
    …
    public int hashCode() {
        return first.hashCode() * 37
                + last.hashCode();
    }
}
```

- Why is it better? (remember contract)

hashCode Object Contract

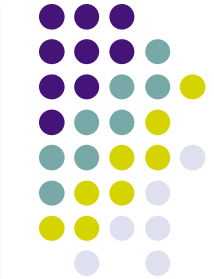
- An object's hashcode cannot change until it is no longer equal to what it was
- Two equal objects must have an equal hashCode
- It is good if two unequal objects have distinct hashes
  - Ex: Jack Nance will be different from Nance Jack

Before We Switch Topics



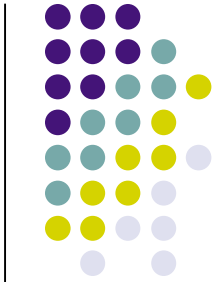
- Any questions about hashCode, please ask!
- It will be an important point later today
- It will cause bizarre problems if you don't understand it

What Collections Do



- “Framework” of Interfaces and Classes to handle:
  - Collecting objects
  - Storing objects
  - Sorting objects
  - Retrieving objects
- Provides common syntax across variety of different Collection implementations

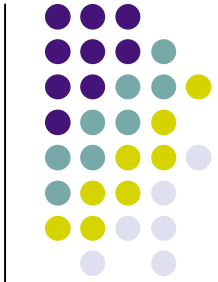
How to use Collections



- add import java.util.*; to the top of every java file

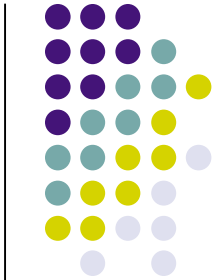
```
package lab2;
import java.util.*;
public class CollectionUser {
        List<String> list = new ArrayList<String>();
        … //rest of class
}
```

Basic Collection<Foo> Syntax



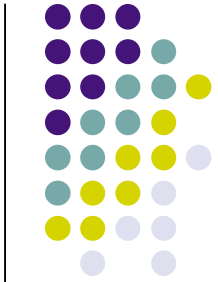
- boolean add(Foo o);
- boolean contains(Object o);
- boolean remove(Foo o);
- int size();

Example Usage



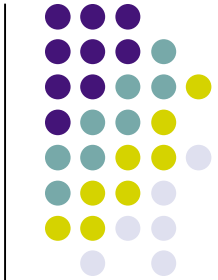
```
List<Name> iapjava = new ArrayList<Name>();

iapjava.add(new Name(“Laura”, “Dern”);
iapjava.add(new Name(“Toby”, “Keeler”);
System.out.println(iapjava.size()); => 2

iapjava.remove(new Name(“Toby”, “Keeler”);
System.out.println(iapjava.size()); => 1

List<Name> iapruby = new ArrayList<Name>();
Iapruby.add(new Name(“Scott”, “Ostler”));
iapjava.addAll(iapruby);

System.out.println(iapjava.size()); => 2
```

Generic Collections



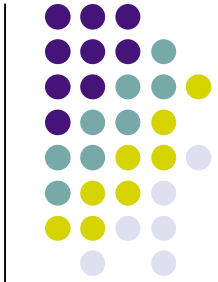
- We can specify the **type** of object that a collection will hold
- Ex: List<String> *strings*
- We are reasonably sure that *strings* contains **only** String objects
- Is optional, but very useful

Why Use Generics?



```
List untyped = new ArrayList();
List<String> typed = new ArrayList<String>();

Object obj = untyped.get(0);
String sillyString = (String) obj;

String smartString = typed.get(0);
```

Retrieving objects



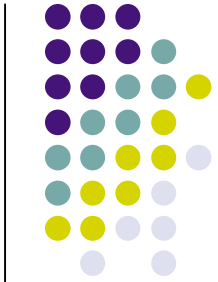
- Given Collection<Foo> coll
- Iterator:

```
Iterator<Foo> it = coll.iterator();
while (it.hasNext) {
   Foo obj = it.next();
   // do something with obj
}
```

- For each:

```
for (Foo obj : coll) {
    // do something with obj
}
```

Object Removing Caveat



- Can't remove objects from a Collection while iterating over it

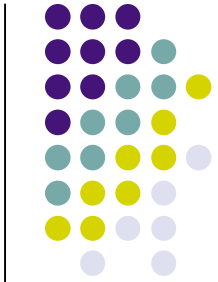
```
for (Foo obj : coll)
    coll.remove(obj) // ConcurrentModificationException
}
```

- Only the Iterator can remove an object it's iterating over

```
Iterator<Foo> it = coll.iterator();
while (it.hasNext) {
   Foo obj = it.next();
   it.remove(Obj); // NOT coll.remove(Obj);
}
```

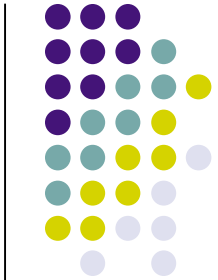
- Note that iter.remove is optional, and not all Iterator objects will support it

General Collection Types



- List
  - ArrayList
- Set
  - HashSet
  - TreeSet
- Map
  - HashMap

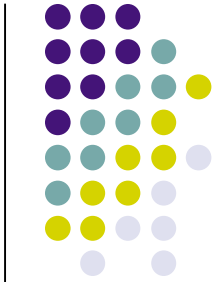
List Overview



- Ordered list of objects, similar to Array
- Unlike Array, no set size
- List order generally equals insert order

```
List<String> strings = new ArrayList<String>();
strings.add(“one”);
strings.add(“two”);
strings.add(“three”);
// strings = [ “one”, “two”, “three”]
```

Other Ways



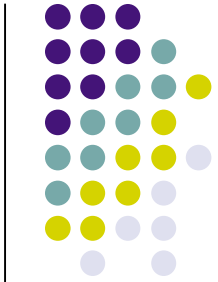
- Insert at an index

```
List<String> strings = new ArrayList<String>();
strings.add("one");
strings.add("three");
strings.add(1, "two");
// strings = [ "one", "two", "three"]
```

- Retrieve objects with an index:

```
s.o.print(strings.get(0))          // => "one"
s.o.print(strings.indexOf("one"))  // => 0
```

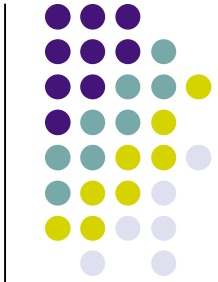
Set Overview



- No set size, no set order
- No duplicate objects allowed!

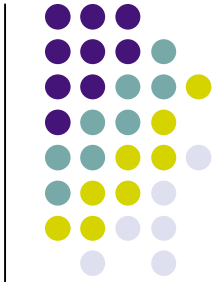
```
Set<Name> names = new HashSet<Name>();
names.add(new Name("Jack", "Nance"));
names.add(new Name("Jack", "Nance"));

System.out.println(names.size()); => 1
```

Set Contract



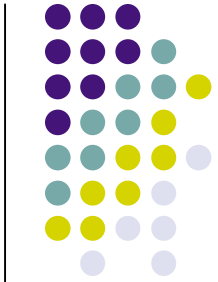
- A set element cannot be changed in a way that affects its equality
- This is a danger of object mutability
- If you don't obey the contract, prepare for Bad, Strange Things

Bad, Strange Thing #2



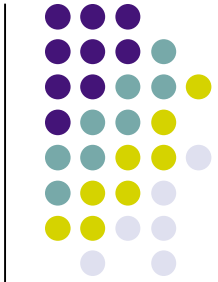
```
Set<Name> names = new HashSet<Name>();
Name jack = new Name(“Jack”, “Nance”);
names.add(jack);
System.out.println(names.size());

System.out.println(names.contains(jack)); => true;

jack.last = "Vance";

System.out.println(names.contains(jack)); => false
System.out.println(names.size()); => 1
```

Solutions to the Problem?



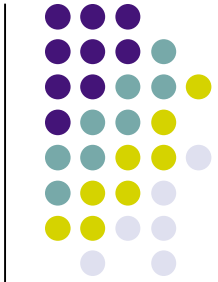
- None.
- So don't do it.
- If at all possible, use immutable set elements
- Otherwise, be careful

Map Overview



- Mapping between a set of “Key-Value Pairs”
- That is, for every Key object, there is a Value object
- Essentially a “lookup service”
- Keys must be unique, but values don’t have to be

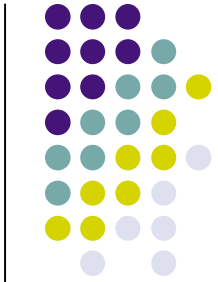
Note: Map is not a Collection



- Map doesn't support:
  - boolean add(Foo obj);
  - boolean contains(Object obj);

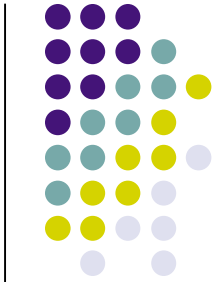
- Rather, it supports:
  - boolean put(Foo key, Bar value);
  - boolean containsKey(Foo key);
  - boolean containsValue(Bar value);

Sample Map Usage



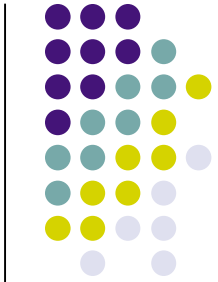
```
Map<String, String> dns = new HashMap<String, String>();

dns.put("scotty.mit.edu", "18.227.0.87");

System.out.println(dns.get("scotty.mit.edu"));
System.out.println(dns.containsKey("scotty.mit.edu"));
System.out.println(dns.containsValue("18.227.0.87"));

dns.remove("scotty.mit.edu");

System.out.println(dns.containsValue("18.227.0.87"));

// => "18.227.0.87", true, true, false
```

Other Useful Methods



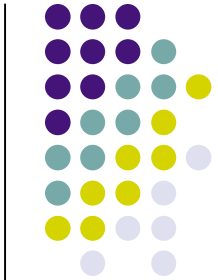
- keySet() - returns a Set of all the keys
- values() - returns a Collection of all the values
- entrySet() - returns a Set of Key,Value Pairs
  - Each pair is a Map.Entry object
  - Map.Entry supports getKey, getValue, setValue

Dangers of Key Mutability



- A key must always be equal to what it was
- This is a restatement of the Set discussion
- If a key chanages, it and its value will be “lost”

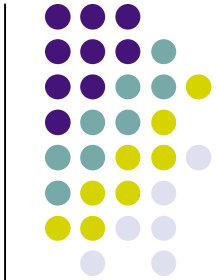
Bad, Strange Thing #3



```
Name isabella = new Name("Isabella", "Rosellini")
Map<Name, String> directory = new HashMap<Name, String>();
directory.put(isabella, "123-456-7890");

System.out.println(directory.get(isabella));

isabella.first = "Dennis";

System.out.println(directory.get(isabella));

directory.put(new Name("Isabella", "Rosellini"), "555-555-1234")
isabella.first = "Isabella";

System.out.println(directory.get(isabella));
```

- What happens?

Two Answers

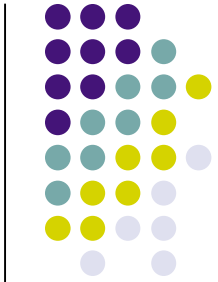


- Right Answer:

```
// => 123-456-7890, null, 555-555-1234
```

- Righter Answer:
  - Doesn't matter because we shouldn't be doing it
  - Unspecified behavior

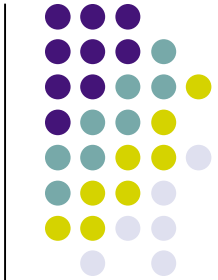
How to Fix Mutable Keys?



- We want to be able to use any object to stand in for another

- But mutable objects are dangerous

Copy the Key

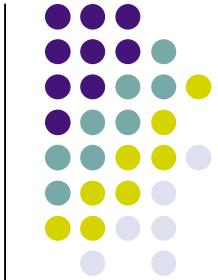


```
Name dennis = new Name(“Dennis”, “Hopper”);
Name copy = new Name(dennis.first, dennis.last);

map.put(copy, “555-555-1234”);
```

- Now changes to dennis don’t mess up map
- But the keys themselves can still be changed

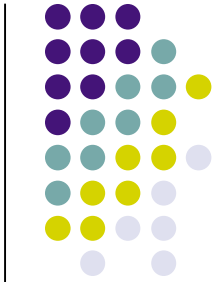
```
For (Name name : map.keySet()) {
   name.first = “u r wrecked”; // uh oh
}
```

Make Immutable Keys



```
public class Name {
    public final String first;
    public final String last;

    public Name(String first, String last) {
        this.first = first;
        this.last = last;
    }

    public boolean equals(Object o) {
        return (o instanceof Name &&
                ((Name) o).first.equals(this.first) &&
                ((Name) o).last.equals(this.last));
    }}
```

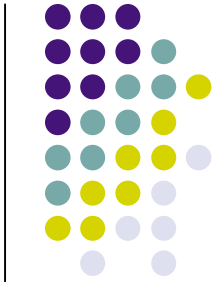
Immutable Proxy for Keys



```
Map<String, String> dir = new HashMap<String, String>();
Name naomi = new Name("Naomi", "Watts");
String key = naomi.first + "," + naomi.last;

dir.put(key, "888-444-1212");
```

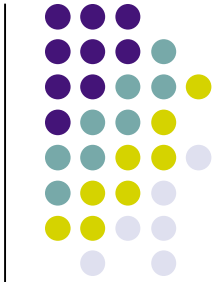
- Strings are immutable, so our Maps will be safe

“Freeze” Keys



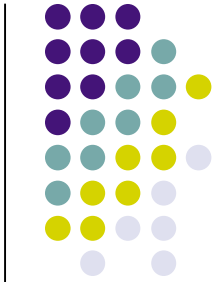
```
public class Name {
    private String first;
    private String last;
    private boolean frozen = false;
    …

    public void setFirst(String s) {
        if (!frozen) first = s;
    }
    … // do same with setLast

    public void freeze() {
        frozen = true;
    }
}
```

Summary: Mutable Keys



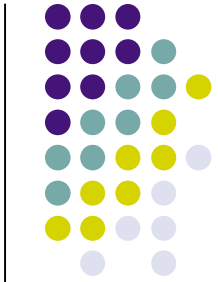
- Each approach has tradeoffs
- But where appropriate, choose the simplest, strongest solution
- If a key cannot ever be changed, there will never be problems
- “Put and Pray” only as a lost resort

Collection Wrap-up



- Common problems
  - Sharing obects between Collections
  - Trying to remove an Object during iteration
  - Mutable Keys, Sets
- Any questions?

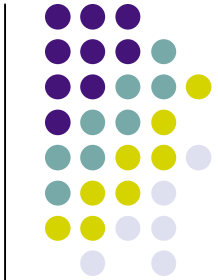
Comparing and Sorting



- Used to decide, between two objects, if one is bigger or they are equal

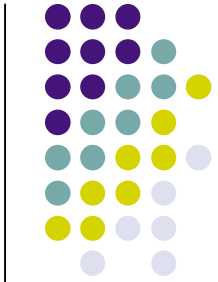
- (a.compareTo(b)) should result in:
  - < 0 if a < b
  - = 0 if a = b
  - > 0 if a > b

Comparison Example



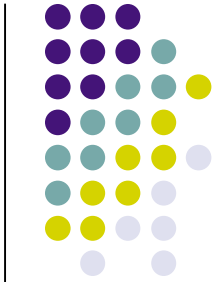
```
Integer one = 1;
System.out.println(one.compareTo(3));
System.out.println(one.compareTo(-50));

String frank = “Frank”;
System.out.println(frank.compareTo(“Booth”));
System.out.println(frank.compareTo(“Hopper”));

// => -1 , 1, 4, -2
```

Sorting a List Alphabetically



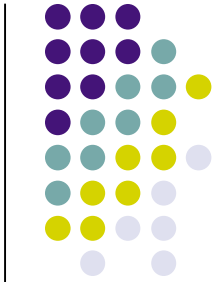
```
List<String> names = new ArrayList<String>();
names.add(“Sailor”);
names.add(“Lula”);
names.add(“Bobby”);
names.add(“Santos”);
names.add(“Dell”);
Collections.sort(names);
// names => [ “Bobby”, “Dell”, “Lula”, “Sailor”,
    “Santos” ]
```

Comparable Interface



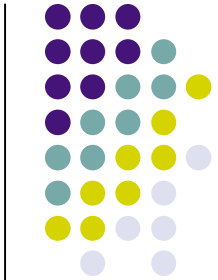
- We can sort Strings because they implement Comparable
- That is, they have a “Natural Ordering”.
- To make Foo class Comparable, we have to implement:
  - int compareTo(Foo obj);

A Sortable Name



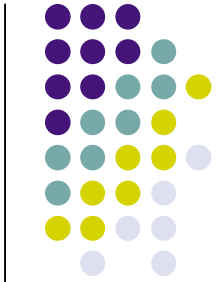
```
public class Name implements Comparable<Name> {
    …
    public int compareTo(Name o) {
        int compare = this.last.compareTo(o.last)
        if (compare != 0)
            return compare;
            else return this.first.compareTo(o.first);
    }
}
```

Sorting Names in Action



```
List<Name> names = new ArrayList<Name>();
names.add(new Name("Nicolas", "Cage"));
names.add(new Name("Laura", "Dern"));
names.add(new Name("Harry", "Stanton"));
names.add(new Name("Diane", "Ladd"));
names.add(new Name("William", "Morgan"));
names.add(new Name(“Dirty”, "Glover"));
names.add(new Name("Johnny", "Cage"));
names.add(new Name("Metal", "Cage"));

System.out.println(names);
Collections.sort(names);
System.out.println(names);

// => [Johnny Cage, Metal Cage, Nicolas Cage, Laura Dern, Crispin Glover,
        Diane Ladd, William Morgan, Harry Stanton]
```

Comparator Objects



- To create multiple sortings for a given Type, we can define Comparator classes
- A Comparator takes in two objects, and determines which is bigger
- For type Foo, a Comparator<Foo> has:

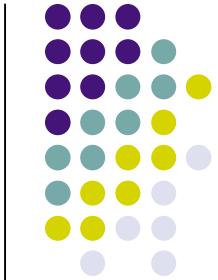
  int compare(Foo o1, Foo o2);

A First-Name-First Comparator

```
public class FirstNameFirst implements
    Comparator<Name> {
    public int compare(Name n1, Name n2) {
        int ret = n1.first.compareTo(n2.first);
        if (ret != 0)
            return ret;
            else return n1.last.compareTo(n2.last);
    }
}
```

- This goes in a separate file, FirstNameFirst.java

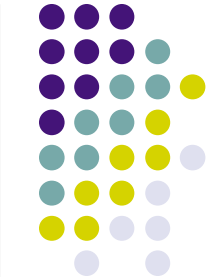
Does it Work?



```
List<Name> names = new ArrayList<Name>();
..
Comparator<Name> first = new FirstNameFirst();
Collections.sort(names, first);
System.out.println(names);

// => [Crispin Glover, Diane Ladd, Harry Stanton, Johnny
    Cage, Laura Dern, Metal Cage, Nicolas Cage, William
    Morgan]
```

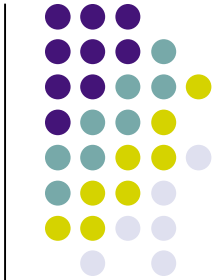
- It works!

Comparison Contract



- Once again, there are rules that we must follow
- Specifically, be careful when
  (compare(e1, e2)==0) != e1.equals(e2)
- With such a sorting, using SortedSet or SortedMap will cause Bad, Strange Things

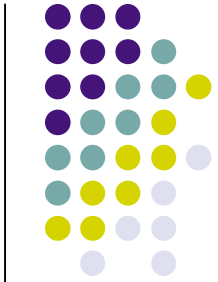
Another Way of Sorting



- Use a TreeSet - automatically kept sorted!
  - Either the Objects in TreeSet must implement Comparable
  - Or give a Comparator Object when making the TreeSet

```
SortedSet<Name> names = new TreeSet<Name>(new
    FirstNameFirst());
names.add(new Name("Laura", "Dern"));
names.add(new Name("Harry", "Stanton"));
names.add(new Name("Diane", "Ladd"));
System.out.println(names);

// => [Diane Ladd, Harry Stanton, Laura Dern]
```

Day 3 Wrap-Up



- Ask questions!
- There was more here than anyone could get or remember
- Think of what you want your code to do, and the best way to express that
- Read Sun's Java Documentation:
  - http://java.sun.com/j2se/1.5.0/docs/api
  - No one can keep Java in their head
  - Everytime you code, have this page open