Comparing Objects

- Objectives - when we have completed this set of notes, you should be familiar with:

  - Defining the natural ordering of objects

  - Implementing the Comparator interface

  - Sorting a list of objects

    - Using the Collections class

  - Sorting an array of objects

    - Using the Arrays class

Natural Ordering

- Often a class has an attribute that defines the order of objects:

    - Strings objects are ordered based on lexographic order

    - Integer objects are ordered based on value

    - An Employee object might be ordered based on ID number or last name

- Implementing the Comparable interface defines the natural order of objects

Natural Ordering

- Recall that when you implement the Comparable interface, you must define the compareTo method

```
obj1.compareTo(obj2)
```

- The compareTo method defines the natural ordering by returning

  - A negative int value if obj1 < obj2
    (i.e., obj1 comes <u>before</u> obj2)

  - Zero if obj1 is equal to obj2

  - A positive int value if obj1 > obj2
    (i.e., obj1 comes <u>after</u> obj2)

Natural Ordering

- Suppose that you would like to use the name of products to define the natural order of Product objects

- Implement the Comparable interface and then define the compareTo method

- In Product.java, the compareTo method orders Product objects based on alphabetical order (ignoring case)

Sorting

- Now that you have defined the way that objects are ordered, you can use classes and methods from the Java API to sort objects

- The Collections class has a static sort method that takes a List of objects

  - Polymorphism via interface: List is an interface implemented by ArrayList, Vector, LinkedList, etc

- When you define the natural ordering of objects, you can use the sort method

- See ProductList.java

Sorting

- Use the static sort method in the Arrays class if you have an array of objects

  - Polymorphism via inheritance: The sort class takes an array of Object types; all classes extend object

- See ProductArray.java

Alternate Ordering

- What if you wanted to define an alternate order to objects?

  - Suppose you built an e-mail system and defined sent date as the natural ordering of Email objects.

  - Some users, however, might want to sort Email objects based on sender's name

- The Comparator interface defines an alternate way of sorting objects.

Alternate Ordering

- Create a separate class that implements java.util.Comparator

- Define the *compare* method in the class: takes two object parameters obj1 and obj2

- The return value for *compare* is similar to *compareTo* in Comparable interface:

  - A negative int value if obj1 < obj2
    (i.e., obj1 comes <u>before</u> obj2)

  - Zero if obj1 is equal to obj2

  - A positive int value if obj1 > obj2
    (i.e., obj1 comes <u>after</u> obj2)

Alternate Ordering

- Suppose that you wanted an alternate ordering for Product objects based on base price

- Create a class that implements Comparator (see BasePriceComparator.java)

- You can create additional classes that implement Comparator

- See TotalPriceComparator.java

Alternate Ordering

- The Collections and Arrays class have overloaded the sort method to take a set of objects AND a Comparator object

- See Java API documentation for Collections and Array

- ProductArraySorts.java