Comparisons
Logistics

• HW10 due yesterday
• Last Recitation this week – just review
• Review on Monday
  – Come with questions
• Final exam next Friday at 3pm in CHEM 102
Nuts and bolts

• Four methods underlie many of Java’s important Collection types: `equals`, `compare` and `compareTo`, and `hashCode`
  – To put your own objects into a Collection, you need to ensure that these methods are defined properly
  – Any collection with some sort of membership test uses `equals` (which, in many cases, defaults to `==`)
  – Any collection that depends on sorting requires larger/equal/smaller comparisons (`compare` or `compareTo`)
  – Any collection that depends on hashing requires both equality testing and hash codes (`equals` and `hashCode`)
  – Any time you implement `hashCode`, you *must* also implement `equals`
• Some of Java’s classes, such as String, already define all of these properly for you
  – For your own objects, you have to do it yourself
Comparing our own objects

- The **Object** class has **equals** and **hashCode** methods for checking if two objects are equal
  - There is nothing in **Object** for “less” or “greater”
- For “less” and “greater”
  - There is a **Comparable** interface in **java.lang**
  - There is a **Comparator** interface in **java.util**
Implementing Comparable<T>

• Any class implementing `Comparable<T>` must implement the method `public int compareTo(T other)`
  – Returns a negative integer if `this` is smaller than `other`, a positive integer if `this` is larger, and 0 if they are equal

• If you store your `Comparable` objects in a collection (say, named `col`), call `Collections.sort(col)` to sort the collection
  – Internally, this uses your `compareTo` method (to build a binary tree)
  – To ensure correctness, `compareTo` must be transitive
    – i.e., if `t1 > t2` and `t2 > t3`, then `t1 > t3`

• Eclipse example: StudentWithComparable.java
Using a separate Comparator

• In the previous example, **Student** implemented **Comparable**
  – Therefore, it had a **compareTo** method
  – We could sort students *only* by their score
  – If we wanted to sort students another way, such as by name, we are out of luck

• Now we will put the comparison method in a *separate class* that implements **Comparator** instead of **Comparable**
  – This is more flexible (you can use a different **Comparator** to order **Students** by name or by score), but it’s also clumsier
  – **Comparator** is in **java.util**, not **java.lang**
  – **Comparable** requires a definition of **compareTo** but **Comparator** requires a definition of **compare**
  – **Comparator** also (sort of) requires **equals**
The compare method

• Any class implementing a `Comparator<T>` needs to implement `public int compare(T t1, T t2)`
  – Returns a negative integer if `t1` is less than `t2`, a positive integer if `t1` is larger than `t2`, and 0 if they are equal

• Unlike `compareTo`, `compare`
  – Takes both objects as parameters
  – Has a different name!
  – If you stored your elements in a collection (named `col`, say), and your `Comparator` object is called `comp`, then you sort your collection using `Collections.sort(col, comp)`

• Eclipse example: `StudentWithComparator.java`, `StudentScoreComparator.java`, `StudentNameNameComparator.java`
When to use each

• The **Comparable** interface is simpler and less work
  – Your class implements **Comparable**
  – You provide a **public int compareTo(T other)** method
  – You will use the same comparison method every time

• The **Comparator** interface is more flexible but slightly more work
  – Create as many different classes that implement **Comparator** as you like
  – You can sort your objects differently with each
  – For example, sort **Students** by score or by name