Comparable and Comparator

Nuts and Bolts

Four methods underlie Collection types: `equals`, `compare` and `compareTo`, and `hashCode`

- Need to ensure that these methods are defined properly for your own objects
- Collection with *membership test* uses `equals` (defaults to `==`)
- Collection that depends on *sorting* requires larger/equal/smaller comparisons (`compare` or `compareTo`)
- 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`

Comparing Objects

- The `Object` class provides `public boolean equals(Object obj)` and `public int hashCode()` methods
  - If we override `equals`, we *should* override `hashCode`
  - If we override `hashCode`, we *must* override `equals`
- The `Object` class does not provide any methods for "less" or "greater"—however,
  - There is a `Comparable` interface in `java.lang`
  - There is a `Comparator` interface in `java.util`

Outline of a Student Class

```
public class Student implements Comparable
{

   public Student(String name, int score) {...}

   public int compareTo(Object o) {...}

   public static void main(String args[]) {...}
}
```

Constructor for Student

- Nothing special here:

```
public Student(String name, int score)
{
        this.name = name;
        this.score = score;
}
```

- Sort students according to score
- Comparisons happen between two *objects*, whatever kind of collection they may or may not be in

The main Method, Version 1

```
public static void main(String args[])
{
    TreeSet<Student> set = new TreeSet<Student>();

    set.add(new Student("Ann", 87));
    set.add(new Student("Bob", 83));
    set.add(new Student("Cat", 99));
    set.add(new Student("Dan", 25));
    set.add(new Student("Eve", 76));

    Iterator<Student> iter = set.iterator();
    while (iter.hasNext())
    {
        Student s = iter.next();
        System.out.println(s.name + "  " + s.score);
    }
}
```

Using the TreeSet

- In the main method we have the line

```
TreeSet set = new TreeSet();
```

- Later we use an iterator to print out the values in order, and get the following result:

```
Dan   25
Eve   76
Bob   83
Ann   87
Cat   99
```

- How did the iterator know that it should sort Students by score, rather than, say, by name?

Implementing Comparable<T>

public class Student implements Comparable

- This means it must implement the method

  public int compareTo(Object o)

- The method **`compareTo`** must return
  - A negative number if the calling object "comes before" the parameter
  - A zero if the calling object "equals" the parameter other
  - A positive number if the calling object "comes after" the parameter other

Implementing Comparable<T>

- Notice that the parameter is an *Object*
- In order to implement this interface, our parameter must also be an *Object*, even if that's not what we want.

```
public int compareTo(Object o) throws ClassCastException
{
   if (o instanceof Student)
      return score - ((Student)o).score;
   else
      throw new ClassCastException("Not a Student!");
}
```

- A ClassCastException should be thrown if we are given a non-Student parameter

An Improved Method

- Since casting an arbitrary Object to a Student may throw a ClassCastException for us, we don't need to throw it explicitly:

```
public int compareTo(Object o) throws
                                    ClassCastException
{
        return score - ((Student)o).score;
}
```

- Moreover, since classCastException is a sub-class of RuntimeException, we don't even need to declare that we might throw one:

```
public int compareTo(Object o) {
        return score - ((Student)o).score;
}
```

Using a Separate Comparator

- Above, Student implemented Comparable
  - 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
- Instead, must put the comparison method in a *separate class* that implements Comparator instead of Comparable
  - This is more flexible, but also clumsier
  - Comparable requires a definition of compareTo but Comparator requires a definition of compare
  - Comparator also (sort of) requires equals

Outline of StudentComparator

```
import java.util.*;

public class StudentComparator
                implements Comparator<Student> {

    public int compare(Student s1, Student s2) {...}

    public boolean equals(Object o1) {...}
}
```

- Note: When we are using this Comparator, we don't need the `compareTo` method in the `Student` class
- Because of generics, our `compare` method can take `Student` arguments instead of just `Object` arguments

The compare Method

```
public int compare(Student  s1, Student s2)
{
        return s1.score – s2.score;
}
```

This differs from compareTo(Object o) in Comparable in these ways:

- The name is different
- It takes both objects as parameters, not just one
- We have to either use generics, or check the type of both objects
- If our parameters are Objects, they have to be cast to Students

The *someComparator*.equals Method

Ignore this method!

- This method is *not* used to compare two Students—it is used to compare two Comparators
- Even though it's part of the Comparator interface, you don't actually need to override it
  - Definition inherited from Object !
- In fact, it's *always* safe to ignore this method
- The purpose is efficiency—you can replace one Comparator with an equal but faster one

The main Method, Version 2

The main method is just like before, except that instead of

```
TreeSet<Student> set = new TreeSet<Student>();
```

We have

```
Comparator<Student> comp = new StudentComparator();
TreeSet<Student> set = new TreeSet<Student>(comp);
```

When to Use Each

- The Comparable interface is simpler and less work
  - Your class implements Comparable
  - You provide a public int compareTo(Object o) method
  - Use no argument in your TreeSet or TreeMap constructor
  - 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 the TreeSet or TreeMap differently with each
    - Construct TreeSet or TreeMap using the comparator you want
  - For example, sort Students by score *or* by name

Sorting Differently

- Suppose you have students sorted by *score*, in a TreeSet you call studentsByScore
- Now you want to sort them again, this time by *name*

```
Comparator<Student> myStudentNameComparator =
        new MyStudentNameComparator();

TreeSet studentsByName =
        new TreeSet(myStudentNameComparator);

studentsByName.addAll(studentsByScore);
```