

Comparable and Comparator

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 provides `public boolean equals(Object obj)` and `public int hashCode()` methods
 - For objects that we define, the inherited `equals` and `hashCode` methods use the object's address in memory
 - We can override these 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)  
        throws ClassCastException {...}  
  
    public static void main(String args[]) {...}  
}
```

Constructor for Student

- This is the same for both methods—nothing new here
- ```
public Student(String name, int score) {
 this.name = name;
 this.score = score;
}
```
- We will be sorting students according to their score
- This example will use sets, but that's irrelevant—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)
- 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 subclass 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

- In the program we just finished, `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 sort 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`

# 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
    - Implementing an interface requires you to have a definition for *every* method in the interface--so how can this be an exception?
    - Because you *do have* a 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

- 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);
```