CSCI 136
Data Structures & Advanced Programming

Making Sorting Generic

Making Sorting Generic

Sorting Class-Based Objects

How can we sort items of a class-based type?

- Need to provide a mechanism for making comparisons
- Unlike equality testing, the Object class doesn't define a "compare()" method 😟
- But provides two mechanisms
  - Both based on implementing an interface
    - The *comparable* interface
    - The *comparator* interface
  - We introduce both mechanisms here

Comparing Objects

Assumes that an ordering exists, denoted by, say, $\precsim$, such that for any pair of items x and y, either

- $x \precsim y$ or $y \precsim x$
  - if both are true we say that x and y are *equal in the ordering:* $x \cong y$
- More precisely, the ordering needs these properties
  - For all x: $x \precsim x$ (*reflexive*)
  - For all x, y: if $x \precsim y$ and $y \precsim x$ then $x \cong y$ (*anti-symmetric)*
  - For all x, y: $x \precsim y$ or $y \precsim x$ (*comparability*)
  - For all x,y,z: if $x \precsim y$ and $y \precsim z$ then $x \precsim z$ (*transitivity*)

Searching & Sorting
The Comparable Interface

- Java provides an interface for comparisons between objects
  - Provides a replacement for "<" and ">" in recBinarySearch
- Java provides the *Comparable* interface, which specifies a method *compareTo()*
  - Any class that implements Comparable must provide compareTo()

```
public interface Comparable<T> {
    //post: return < 0 if this smaller than other
            return 0 if this equal to other
            return > 0 if this greater than other
      int compareTo(T other);
}
```

Comparable Interface

- Many Java-provided classes implement Comparable
  - String (alphabetical order)
  - Wrapper classes: Integer, Character, Boolean
  - All Enum classes
- The *magnitude* of the values returned by `compareTo()` are not important.
  - We only care if the return value is positive, negative, or 0!
    - Often we see -1, 0, 1, but it is up to the implementer
  - For example, in one implementation of java I use
    - "smaller".compareTo("larger") returns the value 7 !

Notes on compareTo()

- `compareTo()` defines a "*natural ordering*" of Objects
  - There's nothing "*natural*" about it…
- We can use `compareTo()` to implement sorting algorithms on anyt generic `List` data structures!
- We can write methods that work on any type that implements Comparable
  - Let's See some examples
    - RecBinSearch.java
    - BinSearchComparable.java

Recursive Binary Search

- Given an array a[] of positive integers in increasing order, and an integer x, find location of x in a[].
  - Take "indexOf" approach: return -1 if x is not in a[]

```
protected static int recBinarySearch(int a[], int value,
              int low, int high) {
   if (low > high) return -1;
   else {
       int mid = (low + high) / 2;       //find midpoint
       if (a[mid] == value) return mid;  //first comparison
                                         //second comparison
       else if (a[mid] < value)          //search upper half
       return recBinarySearch(a, value, mid + 1, high);
        else                             //search lower half
              return recBinarySearch(a, value, low, mid - 1);
   }
```

Comparable Recursive Binary Search

```
protected static <E extends Comparable<E>> int
  recBinarySearch(E a[], E value, int low, int high) {

  if (low > high) return -1;

  int mid = (low + high) / 2; //find middle of array
  int result = a[mid].compareTo(value);

  if (result == 0) {
      return mid;          //we're done!
  } else if (result < 0) {
      //recurse on upper half
      return recBinarySearch(a, value, mid + 1, high);
  } else {
      //recurse on bottom half
      return recBinarySearch(a, value, low, mid - 1);
  }
}
```

Comparable & compareTo

- The Comparable interface (Comparable<T>) is part of the java.lang (not structure5) package.
- Other Java-provided structures can take advantage of objects that implement Comparable
  - See the Arrays class in java.util
  - Example JavaArraysBinSearch
- Users of Comparable are urged to ensure that *compareTo()* and *equals()* are *consistent*. That is,
  - x.compareTo(y) == 0 exactly when x.equals(y) == true
- Note that Comparable limits user to a *single ordering*
- The syntax can get kind of dense
  - See BinSearchComparable.java : a generic binary search method
  - And even more cumbersome….

ComparableAssociation

- Suppose we want an *ordered* Dictionary, so that we can use binary search instead of linear
- Structure5 provides a ComparableAssociation class that implements Comparable.
- The class declaration for ComparableAssociation is

…wait for it...

```
public class ComparableAssociation<K extends Comparable<K>, V>
        Extends Association<K,V> implements
        Comparable<ComparableAssociation<K,V>>
```

(Yikes!)

- Example: Since Integer implements Comparable, we can write
  - ComparableAssociation<Integer, String> myAssoc =
    new ComparableAssociation(new Integer(567), "Bob");
- We could then use Arrays.sort on an array of these

Comparators

- Limitations with Comparable interface?
  - `Comparable` permits `1` order between objects
  - What if `compareTo()` isn't the desired ordering?
  - What if `Comparable` isn't implemented?
- Solution: Comparators

Comparators (Ch 6.8)

- A comparator is an object that contains a method that is capable of comparing two objects
- Sorting methods can be written to apply a Comparator to two objects when a comparison is to be performed
- Different comparators can be applied to the same data to sort in different orders or on different keys

```
public interface Comparator <E> {
     // pre: a and b are valid objects
     // post: returns a value <, =, or > than 0 determined by
     // whether a is less than, equal to, or greater than b
     public int compare(E a, E b);
}
```

Example

Note that Patient does not implement Comparable or Comparator!

```
class Patient {
    protected int age;
    protected String name;
    public Patient (String n, int a) { name = n; age = a; }
    public String getName() { return name; }
    public int getAge() { return age; }
}
```

```
class NameComparator implements Comparator <Patient>{
    public int compare(Patient a, Patient b) {
       return a.getName().compareTo(b.getName());
    }
   // Note: No constructor; a “do-nothing” constructor is added by Java
}
```

```
public void <T> sort(T a[], Comparator<T> c) {
   …
   if (c.compare(a[i], a[max]) > 0) {…}
}
```

```
sort(patients, new NameComparator());
```

Selection Sort with Comparator

```
public static <E> int findPosOfMax(E[] a, int last,
                                   Comparator<E> c) {
      int maxPos = 0      // A wild guess
      for(int i = 1; i <= last; i++)
            if (c.compare(a[maxPos], a[i]) < 0)
                  maxPos = i;
      return maxPos;
}
public static <E> void selectionSort(E[] a, Comparator<E> c) {
      for(int i = a.length - 1; i>0; i--) {
          int big= findPosOfMin(a,i,c);
          swap(a, i, big);
      }
}
```

- The same array can be sorted in multiple ways by passing different `Comparator<E>` values to the sort method;

Comparable vs Comparator

- `Comparable` Interface for class `X`
  - Permits just one order between objects of class `X`
  - Class `X` must implement a `compareTo` method
  - Changing order requires rewriting `compareTo`
    - And then recompiling class `X`
- `Comparator` Interface
  - Allows creation of "compator classes" for class `X`
  - Class `X` isn't changed or recompiled
  - Multiple Comparators for `X` can be developed
    - Ex: Sort Strings by length (alphabetically for same-length)
    - Ex: Sort names by last name instead of first name