Sorting and Generic Methods

Based on the notes from David Fernandez-Baca and Steve Kautz

Bryn Mawr College
CS206 Intro to Data Structures

Selection Sort on an int Array (Java)

```
public static void selectionSort(int[] arr) {
   for (int i = 0; i < arr.length - 1; ++i) {
      int minIndex = i;
      for (int j = i+1; j < arr.length; ++j) {
         if (arr[j] < arr[minIndex]) {
         minIndex = j;
         }
      }
      int temp = arr[i];
      arr[i] = arr[minIndex];
      arr[minIndex] = temp;
   }
}
```

What if we want to sort Strings alphabetically or Points by their x-coordinates?

Comparing Objects in Java

- Approach 1: If a type T implements Comparable, exploit its **natural ordering** and use compareTo(). That is, to compare x with y, invoke x.compareTo(y).
  - For example, to sort Strings instead of ints, we can use the fact that String has a compareTo() method, inherited from the Comparable interface.
- Approach 2: Explicitly define a Comparator object for T and use its compare method to determine the relative order of two objects.
  - For example, if comp is the comparator, we use comp.compare(x,y).

Using the Comparable Interface

- The notion of a "natural" ordering is captured by the Comparable interface.
- Some familiar classes implement Comparable (read the source code), e.g., String and Integer. In other words, String and Integer have a natural ordering.
- The compareTo method allows us to compare an object of type T to another object of type T:
  - x.compareTo(y)<0 ≈ x<y
  - x.compareTo(y)=0 ≈ x=y
  - x.compareTo(y)>0 ≈ x>y
- For other classes that we wish to be Comparable, we have to write our own compareTo() method.

Selection Sort on a String Array (Java)

```
public static void selectionSort(String[] arr) {
    for (int i = 0; i < arr.length - 1; ++i) {
        int minIndex = i;
        for (int j = i+1; j < arr.length; ++j) {
            if (arr[j].compareTo(arr[minIndex])<0) {
                minIndex = j;
            }
        }
        String temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}
```

Using the Comparator Interface

- The Comparator interface defines a class of objects that have a method to compare two objects of type T.

```
public interface Comparable<T>{
    int compareTo(T rhs)
}

public interface Comparator<T> {
    int compare(T lhs, T rhs);
}
```

Behaviors of compareTo and Compare

Idea	Using a Comparable Type	Using a Comparator Type
lhs < rhs	lhs.compareTo(rhs) < 0	comp.compare(lhs, rhs) < 0
lhs > rhs	lhs.compareTo(rhs) > 0	comp.compare(lhs, rhs) > 0
lhs == rhs	lhs.compareTo(rhs) == 0	comp.compare(lhs, rhs) == 0

Not a generic class

```
class LengthComparator implements Comparator<String> {
    public int compare(String lhs, String rhs) {
        return lns.length() – rhs.length();
    }
}
```

Selection Sort on a String Array (Java)

```
public static void selectionSort(String[] arr,
                                 Comparator<String> comp) {
   for (int i = 0; i < arr.length - 1; ++i) {
      int minIndex = i;
      for (int j = i+1; j < arr.length; ++j) {
         if (comp.compare(arr[j], arr[minIndex])<0) {
            minIndex = j; }
      }
      String temp = arr[i];
      arr[i] = arr[minIndex];
      arr[minIndex] = temp;
   }
}
```

To sort Strings by length, we would invoke:
selectionSort(arr, new LengthComparator());

Generic Sorting Methods

- We can define a **sorting method** itself to be generic, so that it takes a type argument as a parameter and sorts arrays of objects of that type.
- Instead of having different programs to sort different types of objects, we would have **one** program that handles multiple types.
- Like a generic class, a generic method has a **type declaration block** which defines one or more type variables. It occurs directly before the return type

public static <T> void selectionSort(T[] arr . . .

Sorting with the Comparator Interface

- Generalizing what we did for String sorting, we can sort objects in an generic array by passing a generic comparator.
- We need to assure Java that we are passing it a comparator that is defined on any supertype of T.
- We do so by giving a **lower bound** for the argument.

Comparator<? super T>

Selection Sort – a Generic Method

```
public static <T> void selectionSort(T[] arr,
                                     Comparator<? super T> comp) {
   for (int i = 0; i < arr.length - 1; ++i) {
      int minIndex = i;
      for (int j = i+1; j < arr.length; ++j) {
         if (comp.compare(arr[j], arr[minIndex])<0) {
            minIndex = j;
         }
      }
      T temp = arr[i];
      arr[i] = arr[minIndex];
      arr[minIndex] = temp;
   }
}
```

Sorting with compareTo()

- To write a generic sorter that uses the Comparable interface, we must impose an **upper bound** on type T, which states that T is guaranteed to implement the Comparable interface.
- Thus, we can call the compareTo() method on objects of type T.

  <T extends Comparable<? super T>>

- Note: when dealing with type parameters, the keyword "extends" is used for both classes and interfaces.

Selection Sort – a Generic Method

```
public static <T extends Comparable<? super T>>
              void selectionSort(T[] arr) {
   for (int i = 0; i < arr.length - 1; ++i) {
      int minIndex = i;
      for (int j = i+1; j < arr.length; ++j) {
         if (arr[j].compare(arr[minIndex])<0) {
            minIndex = j;
         }
      }
      T temp = arr[i];
      arr[i] = arr[minIndex];
      arr[minIndex] = temp;
   }
}
```

Note

Normally, when invoking generic methods, you don't have to tell Java explicitly what the type of the arguments is. Instead, the type is automatically inferred from the compile-time types of the arguments.