CS 228: Introduction to Data Structures
Lecture 15

Generics and Sorting

First, a recap. In Lecture 11, we saw the following Java code for selection sort on an `int` array.

```
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;
  }
}
```

We then saw that we can use essentially the same code to sort `Strings` alphabetically, if we do the comparisons using `compareTo()` instead of "<".

```
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;
  }
}
```

This works, because Java implements `Comparable<String>`. `Comparable` is a ***generic*** interface, which takes a ***type argument*** T:

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

If we want more flexibility in sorting `String` objects (e..g., if we want to sort them by length instead of alphabetically), we can provide the sorting method with a *comparator* specifying how comparisons should be resolved. In Java, the idea of a comparator is captured by the `Comparator` interface. Like `Comparable`, `Comparator` is generic. `Comparable` takes a type argument T specifying the type of objects to be compared. The interface has essentially one method: `compare()`.

```
public interface Comparator<T>
{
  int compare(T x, T y);
  . . .
}
```

To sort `String` objects using a comparator, we would write:

```
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;
  }
}
```

The method could be used like this:

```
selectionSort(arr, new LengthComparator());
```

Generic Sorting Methods

The sorting methods we have seen thus far can sort either `int` arrays or `String` arrays, but not both. We will now see how to define a sorting method to be ***generic***, so that it can sort objects of multiple types.

A generic method has *a* ***type declaration block*** that defines one or more type variables. It occurs directly before the return type, which is `void` in the example below:

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

The scope of a type variable, in this case T, extends through the method. We can refer to T in the method declaration; e.g., we can replace `String` by T.
We will see two ways to define generic sorters, depending on whether we use the `Comparator` interface or the `Comparable` interface.

Generic Sorting with the `Comparator` Interface

We can sort objects in an array of some generic type T by passing a generic comparator.

```
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;
  }
}
```

Note the use of the idiom `Comparator<? super T>` when introducing the comparator. This is a ***lower bound***

for the type of arguments to the comparator; it assures the Java compiler that we are passing it a comparator that is defined on some supertype of T (possibly T itself). We need the lower bound because although we might not have defined a comparator for type T explicitly, we intend to use a comparator for a superclass of T. We will see more about lower (and upper) bounds later.

Generic Sorting with the `Comparable` Interface

We can also write a sorter that can handle objects of any type T that has a natural ordering. For this, we must impose an ***upper bound*** on type T, guaranteeing that T implements the `Comparable` interface (so we can use `compareTo()`). The idiom to express this looks rather intimidating:

```
<T extends Comparable<? super T>>
```

Note. When dealing with type parameters, the keyword "`extends`" is used for both classes and interfaces.

```
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].compareTo(arr[minIndex])
          < 0)
      {
        minIndex = j;
      }
    }
    T temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
}
```

Note. Normally, when invoking generic methods, we do not 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. See `GenericSortingExample` for an illustration of this.

Wild Cards and Bounds

As we have seen, generic sorting methods typically come in one of two forms, corresponding to two ways of comparing objects in Java.

1. For an existing class `Foo`, we can create an implementation of `Comparator<Foo>` and override the `compare()` method.

2. If we are starting from scratch to create a class, say T, we can endow T with a "natural" ordering by having T implement the `Comparable<T>` interface, and then overriding the `compareTo()` method.

The idioms

```
Comparator<? super T>
```

and

```
<T extends Comparable<? super T>>
```

correspond to each of the two approaches. We will now see what these idioms mean.

The Meaning of `Comparator<? super T>`

Suppose we need a version of `selectionSort` that works for any type T, by using an external `Comparator` supplied by the caller. Then, if we write `Comparator<T>` instead of `Comparator<? super T>`, as shown below, we may run into trouble.

```
// first attempt, doesn't quite work
public static <T>
void selectionSort(T[] arr,
                   Comparator<T> comp){...}
```

To see the problem with this, assume you have defined a `Insect` class and have created a comparator `InsectComparator` implementing `Comparator<Insect>`, which compares `Insect` objects by size. Assume also that you have created class Bee that extends `Insect`, adding some fields and methods.

Now suppose you have an array `arr` of type `Bee[]` and a `InsectComparator comp`. You might be surprised to find that the following will *not* compile:

```
selectionSort(arr, comp); // Incorrect!
```

This fails because we are passing in an array consisting of Bee objects, so the type T in `selectionSort` is inferred to be `Bee`. But `comp` does not implement `Comparator<Bee>`, it implements `Comparator<Insect>`.

The solution is to use a wildcard to indicate that our `selectionSort` method can use not just a comparator for type T, but also a comparator defined for any supertype of T as well:

```
// Correct approach:
public static <T>
void selectionSort(T[] arr,
                   Comparator<? super T> comp)
{...}
```

"`? super T`" means "an unknown super-type of T" (possibly an interface, possibly T itself). T is the ***lower bound*** for the unknown type. The goal is to allow subclasses to use a comparison function that was defined in a superclass.

The rule of thumb is:

> If T is a type parameter and you write `Comparator<T>`, there is a good chance you actually want `Comparator<? super T>`.

In this class, we'll only use lower bounds in connection with the `Comparator` interface.

The meaning of `<T extends Comparable<? super T>>`

Suppose we want to sort of objects of a `Comparable` type according to the natural ordering for the type, using `compareTo()`. That is, we want a method such as this:

```
// first attempt, will not work
public static <T>
void selectionSort(T[] arr)
```

The problem is that we cannot invoke `compareTo()` on elements of `arr`, because the compiler has no way to know that type T implements the `Comparable` interface.

The solution is to put an ***upper bound*** on the type parameter to indicate that the type T extends `Comparable<T>`. Then, within the method, we can invoke `compareTo()` on elements of type T.

```
// second attempt, still not quite right
public static <T extends Comparable<T>>
void selectionSort(T[] arr)
```

This is *still* not quite right. To see why, suppose we have the following class.

```
public class Insect
implements Comparable<Insect> {...}
```

We then create subclass of `Insect` called Bee. By inheritance, Bee also implements `Comparable<Insect>`. Now suppose `arr` is an array of type `Bee[]`. The following statement will not compile.

```
selectionSort(arr);              // Trouble!!
```

The problem is that Bee does not implement `Comparable<Bee>`, which is what the upper bound "`T extends Comparable<T>`" requires. So, we must again use "`? super T`" to allow for the fact that the `compareTo` method was defined in a superclass:

```
// success!
public static
<T extends Comparable<? super T>>
void selectionSort(T[] arr){...}
```

The rule of thumb is:

> If T is a type parameter and you write "`T extends Comparable<T>`", there is a good chance you want "`T extends Comparable<? super T>`".

In Com S 228, we will only use the "`T extends Comparable<T>`" idiom in connection with the `Comparable` interface.