This story appeared on JavaWorld at
http://www.javaworld.com/javaworld/jw-12-2002/jw-1227-sort.html

Sort it out

Use the Comparable and Comparator interfaces to sort data in Java

By Alex Blewitt, JavaWorld.com, 12/27/02

Have you ever wondered how to sort elements in a `List`? Java's collections classes and interfaces (in the `java.util` package) provide the ability to sort arbitrary datasets. In this article, we look at how to sort elements stored in `Lists`, how to provide sorting orders for data classes, and how to use a generic mechanism for sorting JavaBeans.

This article specifically covers:

- How sorting algorithms work
- How the `Comparable` interface provides a natural sorting order and how to sort a list of `Strings` into lexicographical order
- How additional sorting orders can be used with existing classes by using the `Comparator` interface and how a generic JavaBeans sorting mechanism sorts arbitrary JavaBeans with `BeanPropertyComparator`
- How multiple sorts apply to a single collection, as demonstrated by `CompositeComparator`

Download the source code associated with this article from Resources.

How sorting works

We start with how sorting algorithms work. Although you don't need to implement sorting algorithms yourself, it's useful to know how they work internally. You can skip to the next section if you just want to jump straight to the code.

It is much easier to find an item from a sorted dataset rather than an unsorted dataset. For example, if you want to look up *liberty*'s definition in a dictionary, you begin by locating the *L* section and then scan for words starting with *li* until you finally find *liberty*. This process would take much longer if the dictionary was a randomly ordered jumble of words.

When you look up a word in the dictionary, you compare a random word with the target word. If the target word you're searching for is "greater" than the random word you've just found, then you know the target word follows the other word; conversely, if the target word is "smaller" than the one you've just found, then you know that the target word precedes it. You then move in the right direction, and the process repeats until you find the target word.

Finding data in Java is easy using a sorted data structure. There are two data structures in the standard Java collections package that sort automatically: the interfaces `SortedSet` and `SortedMap`, which are implemented by `TreeSet` and `TreeMap`, respectively.

How do these collections know how to sort arbitrary classes? The sorting algorithms use a process similar to the dictionary example—repeated pair-wise data comparisons.

Several well-known sorting algorithms exist; the exact implementations and differences between them are outside this article's scope. However, one Java demo, which you may have downloaded with your Java SDK, called the Sorting Algorithm Demo, graphically shows the speed differences between various sorting algorithms. (Click on each applet to see the sorting algorithms work.)

The simplest sorting algorithm—the bubble sort—picks a dataset's first element. It then runs through the remainder of the list until it finds one "smaller" than the one it already has. It locates the list's smallest element and puts it at the top of the list. The process then repeats, starting at the second element, to find the next smallest, and so on until the process reaches the end.

Although more efficient sorting algorithms, such as the appropriately named quick sort, exist, the common behavior in all sorting algorithms is repeated comparison between different elements. Elements are compared to determine which is smaller or greater at each step of the process.

Sort collections

The `Collections` class in Java provides a `sort()` method that allows a (modifiable) list to be sorted. The following code sample shows how a `List` can be created from an array, and then sorted using the `Collections.sort()` method:

```
// Needs to import java.util.*;
Object[] data = {"Kiwi","Banana","Mango","Aubergine","Strawberry"};
List list = Arrays.asList(data);
Collections.sort(list);
System.out.println(list);
// Displays [Aubergine, Banana, Kiwi, Mango, Strawberry]
```

The first line creates an in-line array of `Objects` that can be used to sort strings. The `Arrays` utility class then converts the array into a `List`. This is an efficient way to create a `List`, which can then sort data.

Once we have the array, we then use the `Collections.sort()` method. It modifies the target list and arranges the elements in their *natural* order. The resulting list then displays to the console.

Compare collection elements

So how do the generic sorting algorithms in Java compare elements in a type-independent way? The algorithms rely on the `Comparable` interface, which provides a single method, `compareTo()`:

`public int` **`compareTo`**`(Object obj)` returns:

- A negative integer if `this` is less than `obj`
- Zero if `this` and `obj` are equivalent
- A positive integer if `this` is greater than `obj`

Classes that implement the `Comparable` interface can be compared with one another; this allows the sorting algorithms to arrange such elements in a list. As with the `equals()` method, you should only compare like-typed classes, otherwise a `ClassCastException` may be thrown. The standard Java data types (e.g., `String`, `Integer`) all implement the `Comparable` interface to provide a natural sorting order.

To enable sorting, the `String` class implements the `Comparable` interface. To compare `Strings` in a language-independent way, the `String` class implements the `compareTo()` method to provide a *lexicographic ordering* between strings. In other words, the strings are compared character by character, and the Unicode values determine whether the two strings are different:

```
"Aubergine".compareTo("Banana")     <  0
"Banana"    .compareTo("Aubergine") >  0
"Aubergine".compareTo("Aubergine")  ==  0
```

But beware—you might not always get exactly what you expect using natural ordering. The character-by-character comparison implementation is case sensitive and behaves oddly for accented characters. The Unicode character for `A` is `0x0041`, whereas the code for `a` is `0x0061`. There are also other accented variants for both, such as `&#x00C0;`, `&#x00C1;`, `&#x00E0;`, and `&#x00E1;`, each with their own Unicode character value. Strings containing these characters can sort into different positions; because the character code for `&#x00C0;` is `0x00C0`, `A` < `Z`, while `&#x00C0;` > `Z`.

Implement Comparable

The `Comparable` interface provides a natural (i.e., default) sorting order for a class. We use an example `Date` class to demonstrate how sort order works:

```
public class Date implements Comparable {
  private int year;
  private int month;
  private int day;
  public Date(int year, int month, int day) {
    this.year = year;
    this.month = month;
    this.day = day;
  }
  public int getYear()  { return year; }
  public int getMonth() { return month; }
  public int getDay()   { return day; }
  public String toString() {
    return year + "-" + month + "-" + day;
  }
  public int compareTo(Object o) throws ClassCastException {
    Date d = (Date)o; // If this doesn't work, ClassCastException is thrown
    int yd = year  - d.year;
    int md = month - d.month;
    int dd = day   - d.day;
    if (yd != 0) return yd;
    else if (md != 0) return md;
    else return dd;
  }
}
```

This class defines a data structure that contains three integers: year, month, and day. In the `compareTo()` method, we calculate the difference in year, month, and day. If the years are not the same (`yd!=0`), then the method returns the difference in years. It will be negative if `o < this`, or positive otherwise. The same happens with the month. If the months are the same, then the method returns the difference in days.

Note: The return value does not have to be `-1`, `0`, or `1`. Only the sign is important, not the value. In this case, the code just returns the difference in years, without worrying about the magnitude.

Create additional sort orderings

While the `Comparable` interface allows natural sorting order for a class, it is often desirable to sort data in a different order (such as reverse sorting or case-insensitive sorting). For example, sorting the list of `Strings` `[Aubergine, banana, aubergine, Banana]` results in `[Aubergine, Banana, aubergine, banana]`, because the natural sorting order is a character-by-character comparison.

Although the `Strings`' natural sorting order can't change, you can define an external sort on an existing class using the `Comparator` interface.

The `Comparator` interface defines the `public int` **`compare`**`(Object o1, Object o2)` method that returns:

- A negative integer if `o1` is less than `o2`
- Zero if `o1` and `o2` are considered equivalent
- A positive integer if `o1` is greater than `o2`

To define a case-insensitive sorting operation for strings, we define the following class:

```
public class CaseInsensitiveComparator
 implements java.util.Comparator {
  public int compare(Object o1, Object o2) {
     String s1 = o1.toString().toUpperCase();
     String s2 = o2.toString().toUpperCase();
     return s1.compareTo(s2);
  }
}
```

To use the comparator, we pass an instance as the second argument to the `Collections.sort()`:

```
   // Needs to import java.util.*;
   Object[] data = {"Aubergine","banana","aubergine","Banana"};
   List list = Arrays.asList(data);
   Collections.sort(list, new CaseInsensitiveComparator());
   System.out.println(list);
   // Displays [Aubergine, aubergine, banana, Banana]
```

The only difference between this code and the previous `Collections.sort()` code is the second argument to `Collections.sort()`. In this case, an instance of `CaseInsensitiveComparator()` is passed, which allows the comparison of `List`'s elements using `CaseInsensitiveComparator()` instead of the natural ordering provided by the `String` class's `Comparable` implementation. (Often, a comparator instance will be stored using the Singleton design pattern. This approach is shown in the downloadable code examples.)

Note: Because Aubergine and aubergine are considered equivalent (as are banana and Banana), they remain in the same relative order as they were before the sort. Sorting algorithms that preserve the order for otherwise equal elements are *stable*, and the one implemented by the `Collections` class is stable. If you want capitalized words ahead of their lower-case counterparts, two sorting operations are required: a case-sensitive sort and a case-insensitive sort.

Also note: Sorting a collection of assorted classes might cause problems. Often, a `Comparator` may generate a `ClassCastException` when trying to compare two incompatible types.

Sort JavaBeans

So far, we have seen how to create simple sorts using the `Comparable` and `Comparator` interfaces. However, sorting several kinds of classes could potentially involve the creation of numerous `Comparators`, which would be inefficient.

Instead, we can use the dynamic JavaBean `Introspector` to sort any object written in accordance with the JavaBeans specification. (See Resources for more information about the `Introspector` class.)

JavaBean properties

A JavaBean *property* exposes itself through *accessor* methods. Normally, a `getName()` method (or an optional `setName()` method) exposes a JavaBean property `name`. In addition to providing a common coding convention, this allows a program to access a JavaBean's properties programmatically. (See "Sidebar 1: JavaBeans Properties.")

We use this feature to sort a JavaBean based on one of its properties. We implement a `Comparator` that dynamically accesses a JavaBean's property and then uses those values to implement the sorting order.

Note: Accessing the JavaBean property programmatically is beyond this article's scope, but the source code `BeanPropertyUtil` included in the downloadable examples (see Resources) is documented using Javadoc for those who are interested.

The property name must be provided in the constructor when instantiating the comparator. Then we dynamically access the property's value and use that for sorting:

```
import java.util.Comparator;
public class BeanPropertyComparator implements Comparator {
  private String property;
  private Comparator comparator;
  public BeanPropertyComparator(String property, Comparator comparator) {
    this.property = property;
    this.comparator = comparator;
  }
  public int compare(Object bean1, Object bean2) {
    // Get the value of the properties
    Object value1 = BeanPropertyUtil.getProperty(property,bean1);
    Object value2 = BeanPropertyUtil.getProperty(property,bean2);
    return comparator.compare(value1,value2);
  }
}
```

This code sample uses `BeanPropertyUtil`. It accesses the property named `property` from each of the bean instances as an object (primitive Java types are wrapped) and then uses the given comparator to perform the comparison.

We can now use the `BeanPropertyComparator` to sort JavaBeans. As an example, here is a JavaBean that represents a person:

```
public class Person {
  private String first;
  private String last;
  public Person(String first, String last) {
    this.first = first;
    this.last = last;
  }
  public String getFirst() {
    return first;
  }
  public String getLast() {
    return last;
  }
  public String toString() {
    return last + ", " + first;
  }
}
```

This example has two JavaBean properties (both of which are `String`s): `first` and `last`. We display the JavaBeans' sorted lists using the `SortApplet`:

(**Note:** Java Runtime Environment (JRE) Plug-in 1.3+ is required to view this applet; or run `appletviewer http://www.javaworld.com/javaworld/jw-12-2002/jw-1227-sort-p2.html`.)

The applet builds a list of `Person` instances from hard-coded defaults, but other entries can be added into the list using the Add button. The list on the left shows the data ordering (which is randomized each time the Shuffle button is pressed). The other two lists show the sort orders of the JavaBean properties `last` and `first`. (See Resources for the applet's source code.)

Combining sorts

Now that we can sort on generic JavaBean properties, it is desirable to join the sorts together. For example, in a large table of employees, sorting the employees by both last name and then by first name may be desired.

Unfortunately, the `Collections.sort()` method only takes a single `Comparator`; if you sort the data once, then apply a second sort, the first sort is lost.

Instead, we can build a generic mechanism to combine different sorts. We define a generic `CompositeComparator` that combines the logic of two individual `Comparators`.

We refer to the two `Comparators` as the *major* and *minor* comparators. The major comparator has priority over the minor comparator. If the major comparator returns `< 0` or `> 0`, then that result is passed back. The minor comparator's result is used only if the major comparator returns 0:

```
import java.util.Comparator;
public class CompositeComparator implements Comparator {
  private Comparator major;
  private Comparator minor;
  public CompositeComparator(Comparator major, Comparator minor) {
    this.major = major;
    this.minor = minor;
  }
  public int compare(Object o1, Object o2) {
    int result = major.compare(o1,o2);
    if (result != 0) {
      return result;
    } else {
       return minor.compare(o1,o2);
    }
  }
}
```

This code implements a composite comparator; if the major comparator thinks `o1` and `o2` are equivalent, then the minor comparator is consulted. However, if the major comparator produces a discrepancy, then the minor comparator is not consulted.

We can combine the `CompositeComparator` with the `BeanPropertyComparator` to sort a list of persons by both first and last names:

```
    Comparator last = new BeanPropertyComparator("last");
    Comparator first = new BeanPropertyComparator("first");
    Comparator lastFirst = new CompositeComparator(last,first);
```

This applet shows items with both first/last sorting and last/first sorting:

(JRE Plug-in 1.3+ is required to view this applet; or run `appletviewer http://www.javaworld.com/javaworld/jw-12-2002/jw-1227-sort-p2.html`.)

Note: Because the `CompositeComparator` implements the `Comparator` interface, the `CompositeComparator` can nest within itself. This is called the *Composite* pattern and is exhibited within other parts of Java, such as with the Java Abstract Windowing Toolkit (AWT) `Component` and `Container` classes.

All sorted out

The standard data structures provided in Java's collections classes allow data to be easily manipulated and sorted. If you need to sort JavaBeans, then using `BeanPropertyComparator` can ease the sorting process. It is also possible to simultaneously sort on multiple properties using `CompositeComparator` to nest arbitrary `Comparators` together.

If you need to define a natural ordering for your classes, you should implement the `Comparable` interface. However, if you want to define a new sort for an existing class (including ones for which you don't have source access), then you can implement a standalone `Comparator`.

For more information on the collections classes, see Resources below. For a discussion about whether to sort on Java or databases, see "Sidebar 2: To Sort or Not to Sort (in Java)."

Author Bio

Alex Blewitt is CEO and founder of International Object Solutions Limited. He has been developing with Java since its first release and currently provides consultancy and development of J2EE applications across Europe. When not traveling, Alex enjoys staying at home in Milton Keynes with his wife Amy and their two dogs Milly and Kea.