EXHIBIT C

```
/*
 * @(#)Comparable.java  1.22 03/12/19
 *
 * Copyright 2004 Sun Microsystems, Inc. All rights reserved.
 * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 */

package java.lang;

/**
 * This interface imposes a total ordering on the objects of each class that
 * implements it.  This ordering is referred to as the class's <i>natural
 * ordering</i>, and the class's <tt>compareTo</tt> method is referred to as
 * its <i>natural comparison method</i>.<p>
 *
 * Lists (and arrays) of objects that implement this interface can be sorted
 * automatically by <tt>Collections.sort</tt> (and <tt>Arrays.sort</tt>).
 * Objects that implement this interface can be used as keys in a sorted map
 * or elements in a sorted set, without the need to specify a comparator.<p>
 *
 * The natural ordering for a class <tt>C</tt> is said to be <i>consistent
 * with equals</i> if and only if <tt>(e1.compareTo((Object)e2) == 0)</tt> has
 * the same boolean value as <tt>e1.equals((Object)e2)</tt> for every
 * <tt>e1</tt> and <tt>e2</tt> of class <tt>C</tt>.  Note that <tt>null</tt>
 * is not an instance of any class, and <tt>e.compareTo(null)</tt> should
 * throw a <tt>NullPointerException</tt> even though <tt>e.equals(null)</tt>
 * returns <tt>false</tt>.<p>
 *
 * It is strongly recommended (though not required) that natural orderings be
 * consistent with equals.  This is so because sorted sets (and sorted maps)
 * without explicit comparators behave "strangely" when they are used with
 * elements (or keys) whose natural ordering is inconsistent with equals.  In
 * particular, such a sorted set (or sorted map) violates the general contract
 * for set (or map), which is defined in terms of the <tt>equals</tt>
 * method.<p>
 *
 * For example, if one adds two keys <tt>a</tt> and <tt>b</tt> such that
 * <tt>(!a.equals((Object)b) && a.compareTo((Object)b) == 0)</tt> to a sorted
 * set that does not use an explicit comparator, the second <tt>add</tt>
 * operation returns false (and the size of the sorted set does not increase)
 * because <tt>a</tt> and <tt>b</tt> are equivalent from the sorted set's
 * perspective.<p>
 *
 * Virtually all Java core classes that implement comparable have natural
 * orderings that are consistent with equals.  One exception is
 * <tt>java.math.BigDecimal</tt>, whose natural ordering equates
 * <tt>BigDecimal</tt> objects with equal values and different precisions
 * (such as 4.0 and 4.00).<p>
 *
 * For the mathematically inclined, the <i>relation</i> that defines
 * the natural ordering on a given class C is:<pre>
 *       {(x, y) such that x.compareTo((Object)y) &lt;= 0}.
 * </pre> The <i>quotient</i> for this total order is: <pre>
 *       {(x, y) such that x.compareTo((Object)y) == 0}.
 * </pre>
 *
 * It follows immediately from the contract for <tt>compareTo</tt> that the
 * quotient is an <i>equivalence relation</i> on <tt>C</tt>, and that the
 * natural ordering is a <i>total order</i> on <tt>C</tt>.  When we say that a
 * class's natural ordering is <i>consistent with equals</i>, we mean that the
 * quotient for the natural ordering is the equivalence relation defined by
 * the class's <tt>equals(Object)</tt> method:<pre>
 *     {(x, y) such that x.equals((Object)y)}.
 * </pre><p>
 *
 * This interface is a member of the
 * <a href="{@docRoot}/../guide/collections/index.html">
```

```
 68   * Java Collections Framework</a>.
 69   *
 70   * @author  Josh Bloch
 71   * @version 1.22, 12/19/03
 72   * @see java.util.Comparator
 73   * @see java.util.Collections#sort(java.util.List)
 74   * @see java.util.Arrays#sort(Object[])
 75   * @see java.util.SortedSet
 76   * @see java.util.SortedMap
 77   * @see java.util.TreeSet
 78   * @see java.util.TreeMap
 79   * @since 1.2
 80   */
 81
 82  public interface Comparable<T> {
 83      /**
 84       * Compares this object with the specified object for order.  Returns a
 85       * negative integer, zero, or a positive integer as this object is less
 86       * than, equal to, or greater than the specified object.<p>
 87       *
 88       * In the foregoing description, the notation
 89       * <tt>sgn(</tt><i>expression</i><tt>)</tt> designates the mathematical
 90       * <i>signum</i> function, which is defined to return one of <tt>-1</tt>,
 91       * <tt>0</tt>, or <tt>1</tt> according to whether the value of <i>expression</i>
 92       * is negative, zero or positive.
 93       *
 94       * The implementor must ensure <tt>sgn(x.compareTo(y)) ==
 95       * -sgn(y.compareTo(x))</tt> for all <tt>x</tt> and <tt>y</tt>.  (This
 96       * implies that <tt>x.compareTo(y)</tt> must throw an exception iff
 97       * <tt>y.compareTo(x)</tt> throws an exception.)<p>
 98       *
 99       * The implementor must also ensure that the relation is transitive:
100       * <tt>(x.compareTo(y)&gt;0 &amp;&amp; y.compareTo(z)&gt;0)</tt> implies
101       * <tt>x.compareTo(z)&gt;0</tt>.<p>
102       *
103       * Finally, the implementer must ensure that <tt>x.compareTo(y)==0</tt>
104       * implies that <tt>sgn(x.compareTo(z)) == sgn(y.compareTo(z))</tt>, for
105       * all <tt>z</tt>.<p>
106       *
107       * It is strongly recommended, but <i>not</i> strictly required that
108       * <tt>(x.compareTo(y)==0) == (x.equals(y))</tt>.  Generally speaking, any
109       * class that implements the <tt>Comparable</tt> interface and violates
110       * this condition should clearly indicate this fact.  The recommended
111       * language is "Note: this class has a natural ordering that is
112       * inconsistent with equals."
113       *
114       * @param   o the Object to be compared.
115       * @return  a negative integer, zero, or a positive integer as this object
116       *          is less than, equal to, or greater than the specified object.
117       *
118       * @throws ClassCastException if the specified object's type prevents it
119       *         from being compared to this Object.
120       */
121      public int compareTo(T o);
122  }
```