EXHIBIT C

```
 1  /*
 2   * @(#)Comparable.java  1.22 03/12/19
 3   *
 4   * Copyright 2004 Sun Microsystems, Inc. All rights reserved.
 5   * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 6   */
 7
 8  package java.lang;
 9
10  /**
11   * This interface imposes a total ordering on the objects of each class that
12   * implements it.  This ordering is referred to as the class's <i>natural
13   * ordering</i>, and the class's <tt>compareTo</tt> method is referred to as
14   * its <i>natural comparison method</i>.<p>
15   *
16   * Lists (and arrays) of objects that implement this interface can be sorted
17   * automatically by <tt>Collections.sort</tt> (and <tt>Arrays.sort</tt>).
18   * Objects that implement this interface can be used as keys in a sorted map
19   * or elements in a sorted set, without the need to specify a comparator.<p>
20   *
21   * The natural ordering for a class <tt>C</tt> is said to be <i>consistent
22   * with equals</i> if and only if <tt>(e1.compareTo((Object)e2) == 0)</tt> has
23   * the same boolean value as <tt>e1.equals((Object)e2)</tt> for every
24   * <tt>e1</tt> and <tt>e2</tt> of class <tt>C</tt>.  Note that <tt>null</tt>
25   * is not an instance of any class, and <tt>e.compareTo(null)</tt> should
26   * throw a <tt>NullPointerException</tt> even though <tt>e.equals(null)</tt>
27   * returns <tt>false</tt>.<p>
28   *
29   * It is strongly recommended (though not required) that natural orderings be
30   * consistent with equals.  This is so because sorted sets (and sorted maps)
31   * without explicit comparators behave "strangely" when they are used with
32   * elements (or keys) whose natural ordering is inconsistent with equals.  In
33   * particular, such a sorted set (or sorted map) violates the general contract
34   * for set (or map), which is defined in terms of the <tt>equals</tt>
35   * method.<p>
36   *
37   * For example, if one adds two keys <tt>a</tt> and <tt>b</tt> such that
38   * <tt>(!a.equals((Object)b) && a.compareTo((Object)b) == 0)</tt> to a sorted
39   * set that does not use an explicit comparator, the second <tt>add</tt>
40   * operation returns false (and the size of the sorted set does not increase)
41   * because <tt>a</tt> and <tt>b</tt> are equivalent from the sorted set's
42   * perspective.<p>
43   *
44   * Virtually all Java core classes that implement comparable have natural
45   * orderings that are consistent with equals.  One exception is
46   * <tt>java.math.BigDecimal</tt>, whose natural ordering equates
47   * <tt>BigDecimal</tt> objects with equal values and different precisions
48   * (such as 4.0 and 4.00).<p>
49   *
50   * For the mathematically inclined, the <i>relation</i> that defines
51   * the natural ordering on a given class C is:<pre>
52   *       {(x, y) such that x.compareTo((Object)y) &lt;= 0}.
53   * </pre> The <i>quotient</i> for this total order is: <pre>
54   *       {(x, y) such that x.compareTo((Object)y) == 0}.
55   * </pre>
56   *
57   * It follows immediately from the contract for <tt>compareTo</tt> that the
58   * quotient is an <i>equivalence relation</i> on <tt>C</tt>, and that the
59   * natural ordering is a <i>total order</i> on <tt>C</tt>.  When we say that a
60   * class's natural ordering is <i>consistent with equals</i>, we mean that the
61   * quotient for the natural ordering is the equivalence relation defined by
62   * the class's <tt>equals(Object)</tt> method:<pre>
63   *     {(x, y) such that x.equals((Object)y)}.
64   * </pre><p>
65   *
66   * This interface is a member of the
67   * <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   }
```