KEKER & VAN NEST LLP
ROBERT A. VAN NEST - #84065
rvannest@kvn.com
CHRISTA M. ANDERSON - #184325
canderson@kvn.com
MICHAEL S. KWUN - #198945
mkwun@kvn.com
633 Battery Street
San Francisco, CA 94111-1809
Tel: 415.391.5400
Fax: 415.397.7188

KING & SPALDING LLP
DONALD F. ZIMMER, JR. - #112279
fzimmer@kslaw.com
CHERYL A. SABNIS - #224323
csabnis@kslaw.com
101 Second Street, Suite 2300
San Francisco, CA 94105
Tel: 415.318.1200
Fax: 415.318.1300

KING & SPALDING LLP
SCOTT T. WEINGAERTNER
(*Pro Hac Vice*)
sweingaertner@kslaw.com
ROBERT F. PERRY
rperry@kslaw.com
BRUCE W. BABER *(Pro Hac Vice)*
1185 Avenue of the Americas
New York, NY 10036
Tel: 212.556.2100
Fax: 212.556.2222

IAN C. BALLON - #141819
ballon@gtlaw.com
HEATHER MEEKER - #172148
meekerh@gtlaw.com
GREENBERG TRAURIG, LLP
1900 University Avenue
East Palo Alto, CA 94303
Tel: 650.328.8500
Fax: 650.328.8508

Attorneys for Defendant
GOOGLE INC.

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

SAN FRANCISCO DIVISION

ORACLE AMERICA, INC.,

Plaintiff,

v.

GOOGLE INC.,

Defendant.

Case No. 3:10-cv-03561 WHA

GOOGLE'S MAY 23, 2012 COPYRIGHT LIABILITY TRIAL BRIEF

Dept.: Courtroom 8, 19th Floor
Judge: Hon. William Alsup

Google hereby responds to the Court's request for more briefing regarding interfaces, exceptions and interoperability. *See* Dkt. 1181. Like the other aspects of the SSO of the 37 API packages, interfaces and exceptions are functional requirements for compatibility with the APIs in those packages, and therefore are not copyrightable. *Sega Enters. Ltd. v. Accolade, Inc.*, 977 F.2d 1510, 1522 (9th Cir. 1992) (citing 17 U.S.C. § 102(b)). By implementing the SSO of the 37 API packages, Google increased the extent to which source code is compatible with both the Android and J2SE platforms.

I. The interfaces and exceptions that are publicly declared in the 37 API packages are functional requirements for compatibility with the APIs in those packages, and therefore are not copyrightable.

A. With respect to the 37 API packages, J2SE and Android declare substantially the same number of interfaces, and throw exactly the same number of exceptions.

In J2SE 5.0, the 37 API packages at issue include declarations for 171 interfaces, while in Android 2.2 ("Froyo"), the 37 API packages include declarations for 158 interfaces. These 158 interfaces in Android 2.2 are a subset of the 171 interfaces in J2SE 5.0, i.e., source code referencing, implementing or extending these 158 interfaces in Android 2.2 will also be compatible with J2SE 5.0. Exhibit A, attached hereto, shows the number of interface declarations on a package-by-package basis.[1] For most of the 37 packages, the number of interface declarations is the same in J2SE and Android. *See* Ex. A.

In both J2SE 5.0 and Android 2.2, the public methods in the 37 API packages throw 1,257 exceptions. Exhibit B, attached hereto, shows the number of exceptions thrown by the public methods on a package-by-package basis. For each of the 37 packages, the number of exceptions thrown is the same in J2SE and Android. *See* Ex. B.

B. Example: the Comparable interface.

Interfaces are a listing of methods and fields that "capture what is common across . . . very different things," with the purpose of allowing standardized, simplified interaction with those

[1] Exhibits A and B were both created by programmatic analysis of compiled versions of J2SE 5.0 and Android 2.2—i.e., programmatic analysis of compiled versions of the source code that was admitted into evidence as TX 623 (J2SE 5.0) and TX 46 (Android 2.2).

common features. RT 590:9-11 (Reinhold). Like classes and methods, interfaces have a declaration, and are part of the APIs at issue. TX 984 (*The Java Language Specification*, 3d ed.) at 114; RT 590:1-3 (Reinhold) ("The term Application Programming Interface includes these interfaces in the classes and methods and everything else.").

For example, the java.lang package includes a declaration for the "Comparable" interface. This interface has a single method, called compareTo. The source code declaration of the Comparable interface, without comments, is:

```
public interface Comparable<T> {
        . . .
        public int compareTo(T o);
}
```

Ex. C (excerpt from TX 623),[2] lines 82, 121-22. This means that any class that "implements" the Comparable interface must declare a method called "compareTo" that returns an integer and accepts a single argument that has the same "type" as the class being declared. The documentation for the compareTo method provides that if there are two objects called "x" and "y" that are instantiated from the same class, and that class implements the Comparable interface, the source code expression "x.compareTo(y)" will return a negative integer if x is less than y, a zero if x and y are equal, and a positive integer if x is greater than y. *See* Ex. C, lines 114-16. The Comparable interface includes only one method, but an interface can have additional methods or fields.

The Comparable interface allows a developer to declare a method that relies on the presence of the compareTo method that is promised for all classes that implement the Comparable interface. For example, the ComparableTimSort.java file, written by Josh Bloch, includes a method that sorts arrays of objects that implement the Comparable interface. *See* Ex. D (TX 45.2), lines 20-22. At various points in ComparableTimSort.java, the source code generically refers to a "Comparable" object, e.g.:

```
if (((Comparable) a[runHi++]).compareTo(a[lo]) < 0) {
```

[2] Exhibit C is a printed version of the file licenseebundles/source-bundles/tmp/j2se/src/share/classes/java/lang/Comparable.java, from TX 623.

Ex. D, line 286. The "(Comparable)" syntax indicates that the object "a" must be an object instantiated from a class that implements the Comparable interface. By making use of the interface construct, Josh Bloch was able to write the ComparableTimSort method in a manner that works for *any* array of Comparable objects. Indeed, if tomorrow a developer were to create a new class that implemented the Comparable interface, Josh Bloch's ComparableTimSort method would sort an array of objects instantiated from that new class, even though Josh Bloch could not have known about that developer's new class when he wrote the source code for the ComparableTimSort method.

Had Google not implemented the publicly declared interfaces that are in the 37 API packages, code that depends on them would not work. For example, if Android did not declare the Comparable interface, then a class that includes "implements Comparable" as part of its declaration would not compile. Moreover, had Android omitted the Comparable interface, methods that depend on it, like ComparableTimSort, would not function on the Android platform. Thus, because the public interfaces in the 37 API packages are functionally required for compatibility with the APIs in those packages, those interface declarations are not copyrightable. *Sega*, 977 F.2d at 1522 (citing 17 U.S.C. § 102(b)).

C. Example: the FileNotFoundException exception.

Exceptions are a type of class used by the Java language to communicate to a program that a particular error has occurred. TX 984 at 297. An exception can signal a problem internal to the program, such as having a beginning index that is greater than the ending index when sorting an array. An exception can also signal an external problem, such as a missing file.

When an error occurs, the method in which the error occurs is said to "throw" the relevant exception. *Id.* The method can then either address ("catch") the exception itself, or pass the exception on to the code that called the method.[3] In the latter case, the declaration of the method generally must include the word "throws" followed by the type of exception that is thrown. *Id.* at

[3] Because a method that throws an exception passes that exception to the code that calls the method, the exception can be thought of as part of the "input-output" schema for a method, although it is not the same as the "return" for the method.

394 (discussing inclusion of throw in method and constructor declarations); *see also id.* at 301-02, 222 (explaining why not all types of exceptions must be listed in the method declaration). Methods may throw more than one exception.[4] *Id.* at 221.

As an example, the method java.io.FileReader FileReader(File file) can throw the java.io.FileNotFoundException. This informs the code that called the FileReader method that the requested file could not be found, and thus could not be read. The declaration of the java.io.InputStream method indicates the type of exception that may be thrown so that developers invoking FileReader know that their code needs to "catch" that type of exception:

public FileReader(File file) *throws FileNotFoundException*

Ex. E (excerpt from TX 610.2[5]) (emphasis added).

The exceptions named in the throws clause are "part of the contract between the implementor [of the method] and the user [of the method—i.e., the developer writing source code that invokes the method]." TX 984 at 299. Because of this, the Java language specification requires the compiler to check to ensure that exceptions are properly handled. *Id.* at 299. For example, if "throws FileNotFoundException" is removed from the declaration of a method that throws that exception in the implementation of that method (such as the FileReader method), the class that contains the method will not compile. *Id.* at 301. In addition, if an application catches that exception, but the method that throws the exception does not have a throw clause with that exception in its declaration, the application that calls the method will fail to compile. *Id.*

As a result, maintaining the correct information about thrown exceptions in the method declaration is necessary for compatibility. Indeed, the thrown exceptions are *part* of the method declaration. *See* TX 984 at 210 (defining "*Throws*$_{opt}$" as part of the "*MethodHeader*"), 221 (discussing the "throws" clause). Because the thrown exceptions are part of the functional requirements for compatibility with the 37 API packages, they are not copyrightable. *Sega*, 977

[4] The table in Ex. B counts the total number of exceptions thrown by methods in the public API. Because more than one exception may be thrown by a method, this is larger than the number of methods that throw exceptions.

[5] Exhibit D is a printed version of the file /java/lang/Comparable.html, from TX 610.2. This file is also available on the web at http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Comparable.html.

F.2d at 1522 (citing 17 U.S.C. § 102(b)).

II. Because the SSO of the 37 API packages is functionally required for compatibility with the APIs in those packages, it is not copyrightable.

A. Because Android implements the SSO of the 37 API packages, code written using the APIs in those packages is interoperable between Android and J2SE.

There is no quantitative data in the trial record that demonstrates the extent to which J2SE applications written before Android was released are able to run on the Android platform. Nor is there any quantitative data in the trial record that demonstrates the extent to which post-Android applications are able to run both on the Android and J2SE platforms.

The record does demonstrate, however, that code that relies on APIs that are common to the two platforms will compile and execute on both platforms. RT 2172:6-11 (Astrachan) ("Q. Do you have an opinion, professor, whether, from a computer science perspective, Android and Java are compatible with respect to the methods and other constructors and other items in the classes of the 37 accused packages? A. Yes. For those 37 packages, the code that I write on one platform will run on the other platform."); *see also* RT 2171:24-2172:11 (Astrachan); RT 2287:1-8 (Mitchell) ("I think the point that was illustrated by this code and Dr. Astrachan's description of it is that, for a given piece of code such as this class that he wrote with a marker, it may run on both platforms if the only things it requires are things that are common to the two"); RT 2292:25-2293:14 (Mitchell) (agreeing that Dr. Astrachan's code would work both on the Android and J2SE platforms, and that calling the two platforms "compatible" in this sense is using "a great definition of 'compatible'"). Professor Astrachan explained this during trial as follows:

> Q. Have you formed an opinion, Professor, regarding what, if anything, accounts for the fact that the 37 packages in both have the same structure, organization, and use the same names?
>
> A. Those same names that we have in Android and in Java are needed so that the code inter-operates, so that code I write can be reused in another situation. So for the functionality of using those APIs, the method signatures need to be the same so that the code will inter-operate and meet programmer expectations.

RT 2183:2-11. Ensuring that the signatures used by Android for the APIs in the 37 packages match the signatures used in J2SE "is what allows me to use the libraries on both—use the code I write, like that code up there, on both platforms. Because I'm using those method signatures, my

code will function the same on both platforms." RT 2183:17-20; *see also* RT 2185:5-9 ("that structure of the names of the classes, packages, and methods needs to be the same so that the code will work on both platforms, be compatible, inter-operate, so that I can call the methods. Those need to be the same.").

Moreover, even to the extent that a J2SE application relies on J2SE APIs that are not supported on the Android platform—or to the extent that an Android application relies on Android APIs that are not supported on the J2SE platform—the portion of the source code that relies on APIs that are common to the two platforms will not need to be rewritten. Thus, even for applications that rely on APIs that are not common to both platforms, the Android and J2SE platforms are still *partially* compatible. Indeed, Professor Mitchell testified that one reason why he believes Google wanted to use the APIs in the 37 packages is because those APIs "are known *and used in existing code*." RT 2289:21-13 (emphasis added); *see also* RT 1787:23-1788:4 (Bornstein) ("And, actually, not even just a matter of comfort, but there's a lot of source code out there that wasn't—you know, wasn't written by—well, that was written by lots of people that already existed that could potentially work just fine on Android. And if we went and changed all the names of things, then that source code wouldn't just work—").

By implementing these core APIs, Google reduced the effort required to "port" an application from one platform (e.g., the J2SE platform) to another (e.g., the Android platform), thus promoting increased interoperability. This effort is similar to what is necessary to "port" applications from, for example, the J2SE platform to a given profile of the J2ME platform, or from one profile of the J2ME platform to another. As Dr. Reinhold testified:

> Write once, run anywhere was never a promise that if you wrote code for one Java platform that it would automatically/magically work on another.
>
> The write once, run anywhere promise is relative to a one of the Java platforms. If you write an application that uses Java SE 5, then you can run it on Sun's implementation, on Oracle's implementation, on IBM's implementation, and on others.
>
> Will that same code run on a particular configuration of Java ME? Well, it depends. *It might. It might not. It depends which APIs it uses.*

RT 725:10-20 (emphasis added).

In addition, because the Android platform shares a common core set of APIs with the J2SE platform, developers are able to use experience they gain from working with one platform when developing applications for the other platform. Developers expect these core APIs when they write code in the Java language. RT 2202:6-11, 2203:11-15 (Astrachan); RT 2291:1-8 (Mitchell); RT 364:17-21 (Kurian); RT 519:16-520:6 (Screven). In this sense, Android is compatible with the skills and expectations of Java language programmers.

Finally, the record establishes that interoperability was a motive of Google at the time it made the decision to implement the 37 API packages. Google chose the 37 API packages precisely *because* Java language developers expect them to be present when they write code in the Java language. RT 1782:6-1783:10 (Bornstein). "The goal of the project was to provide something that was familiar to developers." RT 1783:19-21. And in hiring the contractor Noser to help write source code implementing the 37 API packages, Google explained in its Statement of Work detailing "the responsibilities of Noser and the Project Services to be provided by Noser" that Google was "interested in *compatibility* with J2SE 1.5 . . . ." TX 2765 at 9, 12 (emphasis added).

B. Under *Sega*, elements that are functionally required for compatibility are not copyrightable, regardless of how they are used.

In *Sega*, the Ninth Circuit held that Accolade's copying and disassembly of Sega's firmware code was a fair use, because Accolade's purpose in copying was "for studying or examining the *unprotected aspects* of a copyrighted computer program . . . ." 977 F.2d at 1520 (emphasis added). Those unprotected aspects were "the functional requirements for compatibility with the Genesis console—aspects of Sega's programs that *are not protected by copyright.* 17 U.S.C. § 102(b)." *Id.* at 1522 (emphasis added).

The logical order of the Ninth Circuit's reasoning is important. The Ninth Circuit did *not* hold that the fair use doctrine allowed Accolade to copy aspects of Sega's programs that were required for compatibility. Instead, the Ninth Circuit held that functional requirements for compatibility are not protected by the Copyright Act in the first instance. That is, Accolade did not need to rely on the fair use doctrine to establish that it was entitled to copy functional

requirements for compatibility. Instead, it relied on the fair use doctrine to establish that it was allowed Accolade to copy and disassemble *all* of Sega's code to the extent necessary to determine what was functionally required for compatibility.

Because aspects of a computer program that are functionally required for compatibility are not copyrightable, it *does not matter what the defendant does with them.* Even if the defendant's product *is not compatible* with the plaintiff's product, the plaintiff still cannot assert infringement based only on the copying of unprotected elements. "The protection established by the Copyright Act for original works of authorship does not extend to the ideas underlying a work *or to the functional or factual aspects of the work.*" *Sega*, 977 F.2d at 1524 (emphasis added) (citing 17 U.S.C. § 102(b)). A "party claiming infringement may place *no* reliance upon any similarity in expression resulting from unprotectable elements." *Apple Computer, Inc. v. Microsoft Corp.*, 35 F.3d 1435, 1446 (9th Cir. 1994) (quotation marks and citation omitted; emphasis in original). Thus, because the SSO of the 37 API packages is functionally required for compatibility, Google was entitled to use that SSO, regardless of whether its use made Android fully compatible or not.

However, as noted, by implementing the SSO of the 37 API packages, Google intended to promote interoperability. And by implementing that SSO, Google increased the extent to which source code written for one platform will operate on the other.

Dated: May 23, 2012

KEKER & VAN NEST LLP

/s/ Robert A. Van Nest
By: ROBERT A. VAN NEST

Attorneys for Defendant
GOOGLE INC.

EXHIBIT A

Ex. A: Interfaces in the 37 Java API Packages

Package Name	J2SE 5 Interfaces	Android Froyo Interfaces
java.awt.font	2	0
java.beans	9	1
java.io	12	12
java.lang	8	8
java.lang.annotation	1	1
java.lang.ref	0	0
java.lang.reflect	9	9
java.net	6	6
java.nio	0	0
java.nio.channels	7	7
java.nio.channels.spi	0	0
java.nio.charset	0	0
java.nio.charset.spi	0	0
java.security	12	12
java.security.acl	5	5
java.security.cert	8	8
java.security.interfaces	13	13
java.security.spec	3	3
java.sql	18	18
java.text	2	2
java.util	16	16
java.util.jar	2	2
java.util.logging	2	2
java.util.prefs	3	3
java.util.regex	1	1
java.util.zip	1	1
javax.crypto	1	1
javax.crypto.interfaces	4	4
javax.crypto.spec	0	0
javax.net	0	0
javax.net.ssl	10	10
javax.security.auth	2	1
javax.security.auth.callback	2	2
javax.security.auth.login	0	0
javax.security.auth.x500	0	0
javax.security.cert	0	0
javax.sql	12	10
TOTAL	**171**	**158**

EXHIBIT B

Ex. B: Thrown Exceptions in the 37 Java API Packages

Package Name	J2SE 5 Exceptions Thrown	Android Froyo Exceptions Thrown
java.awt.font	1	1
java.beans	15	15
java.io	76	76
java.lang	45	45
java.lang.annotation	0	0
java.lang.ref	2	2
java.lang.reflect	50	50
java.net	86	86
java.nio	3	3
java.nio.channels	25	25
java.nio.channels.spi	13	13
java.nio.charset	4	4
java.nio.charset.spi	1	1
java.security	93	93
java.security.acl	6	6
java.security.cert	57	57
java.security.interfaces	1	1
java.security.spec	1	1
java.sql	400	400
java.text	3	3
java.util	10	10
java.util.jar	18	18
java.util.logging	18	18
java.util.prefs	14	14
java.util.regex	1	1
java.util.zip	14	14
javax.crypto	38	38
javax.crypto.interfaces	0	0
javax.crypto.spec	3	3
javax.net	3	3
javax.net.ssl	54	54
javax.security.auth	6	6
javax.security.auth.callback	3	3
javax.security.auth.login	3	3
javax.security.auth.x500	1	1
javax.security.cert	10	10
javax.sql	179	179
TOTAL	**1257**	**1257**

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   }
```

EXHIBIT E

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```
 1 /*
 2  * Copyright (C) 2008 The Android Open Source Project
 3  *
 4  * Licensed under the Apache License, Version 2.0 (the "License");
 5  * you may not use this file except in compliance with the License.
 6  * You may obtain a copy of the License at
 7  *
 8  *      http://www.apache.org/licenses/LICENSE-2.0
 9  *
10  * Unless required by applicable law or agreed to in writing, software
11  * distributed under the License is distributed on an "AS IS" BASIS,
12  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
13  * See the License for the specific language governing permissions and
14  * limitations under the License.
15  */
16
17 package java.util;
18
19 /**
20  * This is a near duplicate of {@link TimSort}, modified for use with
21  * arrays of objects that implement {@link Comparable}, instead of using
22  * explicit comparators.
23  *
24  * <p>If you are using an optimizing VM, you may find that
    ComparableTimSort
25  * offers no performance benefit over TimSort in conjunction with a
26  * comparator that simply returns {@code
    ((Comparable)first).compareTo(Second)}.
27  * If this is the case, you are better off deleting ComparableTimSort to
28  * eliminate the code duplication.  (See Arrays.java for details.)
29  */
30 class ComparableTimSort {
31     /**
32      * This is the minimum sized sequence that will be merged.  Shorter
33      * sequences will be lengthened by calling binarySort.  If the
        entire
34      * array is less than this length, no merges will be performed.
35      *
36      * This constant should be a power of two.  It was 64 in Tim Peter's
        C
37      * implementation, but 32 was empirically determined to work better
        in
38      * this implementation.  In the unlikely event that you set this
        constant
39      * to be a number that's not a power of two, you'll need to change
        the
40      * {@link #minRunLength} computation.
41      *
42      * If you decrease this constant, you must change the stackLen
43      * computation in the TimSort constructor, or you risk an
44      * ArrayOutOfBounds exception.  See listsort.txt for a discussion
45      * of the minimum stack length required as a function of the length
46      * of the array being sorted and the minimum merge sequence length.
47      */
48     private static final int MIN_MERGE = 32;
49
50     /**
51      * The array being sorted.
52      */
```

```
    private final Object[] a;

    /**
     * When we get into galloping mode, we stay there until both runs win less
     * often than MIN_GALLOP consecutive times.
     */
    private static final int  MIN_GALLOP = 7;

    /**
     * This controls when we get *into* galloping mode.  It is initialized
     * to MIN_GALLOP.  The mergeLo and mergeHi methods nudge it higher for
     * random data, and lower for highly structured data.
     */
    private int minGallop = MIN_GALLOP;

    /**
     * Maximum initial size of tmp array, which is used for merging. The array
     * can grow to accommodate demand.
     *
     * Unlike Tim's original C version, we do not allocate this much storage
     * when sorting smaller arrays.  This change was required for performance.
     */
    private static final int INITIAL_TMP_STORAGE_LENGTH = 256;

    /**
     * Temp storage for merges.
     */
    private Object[] tmp;

    /**
     * A stack of pending runs yet to be merged.  Run i starts at
     * address base[i] and extends for len[i] elements.  It's always
     * true (so long as the indices are in bounds) that:
     *
     *     runBase[i] + runLen[i] == runBase[i + 1]
     *
     * so we could cut the storage for this, but it's a minor amount,
     * and keeping all the info explicit simplifies the code.
     */
    private int stackSize = 0;  // Number of pending runs on stack
    private final int[] runBase;
    private final int[] runLen;

    /**
     * Asserts have been placed in if-statements for performace. To enable them,
     * set this field to true and enable them in VM with a command line flag.
     * If you modify this class, please do test the asserts!
     */
    private static final boolean DEBUG = false;

    /**
     * Creates a TimSort instance to maintain the state of an ongoing
```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```
         sort.
105      *
106      * @param a the array to be sorted
107      */
108     private ComparableTimSort(Object[] a) {
109         this.a = a;
110
111         // Allocate temp storage (which may be increased later if
            necessary)
112         int len = a.length;
113         @SuppressWarnings({"unchecked", "UnnecessaryLocalVariable"})
114         Object[] newArray = new Object[len < 2 *
            INITIAL_TMP_STORAGE_LENGTH ?
115                                        len >>> 1 :
                                           INITIAL_TMP_STORAGE_LENGTH];
116         tmp = newArray;
117
118         /*
119          * Allocate runs-to-be-merged stack (which cannot be expanded).
             The
120          * stack length requirements are described in listsort.txt.  The
             C
121          * version always uses the same stack length (85), but this was
122          * measured to be too expensive when sorting "mid-sized" arrays
             (e.g.,
123          * 100 elements) in Java.  Therefore, we use smaller (but
             sufficiently
124          * large) stack lengths for smaller arrays.  The "magic numbers"
             in the
125          * computation below must be changed if MIN_MERGE is decreased.
             See
126          * the MIN_MERGE declaration above for more information.
127          */
128         int stackLen = (len <    120  ?  5 :
129                         len <   1542  ? 10 :
130                         len < 119151  ? 19 : 40);
131         runBase = new int[stackLen];
132         runLen = new int[stackLen];
133     }
134
135     /*
136      * The next two methods (which are package private and static)
         constitute
137      * the entire API of this class.  Each of these methods obeys the
         contract
138      * of the public method with the same signature in java.util.Arrays.
139      */
140
141     static void sort(Object[] a) {
142          sort(a, 0, a.length);
143     }
144
145     static void sort(Object[] a, int lo, int hi) {
146         rangeCheck(a.length, lo, hi);
147         int nRemaining  = hi - lo;
148         if (nRemaining < 2)
149             return;  // Arrays of size 0 and 1 are always sorted
150
151         // If array is small, do a "mini-TimSort" with no merges
152         if (nRemaining < MIN_MERGE) {
```

```
153             int initRunLen = countRunAndMakeAscending(a, lo, hi);
154             binarySort(a, lo, hi, lo + initRunLen);
155             return;
156         }
157
158         /**
159          * March over the array once, left to right, finding natural
             runs,
160          * extending short natural runs to minRun elements, and merging
             runs
161          * to maintain stack invariant.
162          */
163         ComparableTimSort ts = new ComparableTimSort(a);
164         int minRun = minRunLength(nRemaining);
165         do {
166             // Identify next run
167             int runLen = countRunAndMakeAscending(a, lo, hi);
168
169             // If run is short, extend to min(minRun, nRemaining)
170             if (runLen < minRun) {
171                 int force = nRemaining <= minRun ? nRemaining : minRun;
172                 binarySort(a, lo, lo + force, lo + runLen);
173                 runLen = force;
174             }
175
176             // Push run onto pending-run stack, and maybe merge
177             ts.pushRun(lo, runLen);
178             ts.mergeCollapse();
179
180             // Advance to find next run
181             lo += runLen;
182             nRemaining -= runLen;
183         } while (nRemaining != 0);
184
185         // Merge all remaining runs to complete sort
186         if (DEBUG) assert lo == hi;
187         ts.mergeForceCollapse();
188         if (DEBUG) assert ts.stackSize == 1;
189     }
190
191     /**
192      * Sorts the specified portion of the specified array using a binary
193      * insertion sort.  This is the best method for sorting small
         numbers
194      * of elements.  It requires O(n log n) compares, but O(n^2) data
195      * movement (worst case).
196      *
197      * If the initial part of the specified range is already sorted,
198      * this method can take advantage of it: the method assumes that the
199      * elements from index {@code lo}, inclusive, to {@code start},
200      * exclusive are already sorted.
201      *
202      * @param a the array in which a range is to be sorted
203      * @param lo the index of the first element in the range to be
         sorted
204      * @param hi the index after the last element in the range to be
         sorted
205      * @param start the index of the first element in the range that is
206      *        not already known to be sorted (@code lo <= start <= hi}
207      */
```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```
208     @SuppressWarnings("fallthrough")
209     private static void binarySort(Object[] a, int lo, int hi, int
        start) {
210         if (DEBUG) assert lo <= start && start <= hi;
211         if (start == lo)
212             start++;
213         for ( ; start < hi; start++) {
214             @SuppressWarnings("unchecked")
215             Comparable<Object> pivot = (Comparable) a[start];
216
217             // Set left (and right) to the index where a[start] (pivot)
                belongs
218             int left = lo;
219             int right = start;
220             if (DEBUG) assert left <= right;
221             /*
222              * Invariants:
223              *   pivot >= all in [lo, left).
224              *   pivot <  all in [right, start).
225              */
226             while (left < right) {
227                 int mid = (left + right) >>> 1;
228                 if (pivot.compareTo(a[mid]) < 0)
229                     right = mid;
230                 else
231                     left = mid + 1;
232             }
233             if (DEBUG) assert left == right;
234
235             /*
236              * The invariants still hold: pivot >= all in [lo, left) and
237              * pivot < all in [left, start), so pivot belongs at left.
                 Note
238              * that if there are elements equal to pivot, left points to
                 the
239              * first slot after them -- that's why this sort is stable.
240              * Slide elements over to make room to make room for pivot.
241              */
242             int n = start - left;  // The number of elements to move
243             // Switch is just an optimization for arraycopy in default
                case
244             switch(n) {
245                 case 2:  a[left + 2] = a[left + 1];
246                 case 1:  a[left + 1] = a[left];
247                          break;
248                 default: System.arraycopy(a, left, a, left + 1, n);
249             }
250             a[left] = pivot;
251         }
252     }
253
254     /**
255      * Returns the length of the run beginning at the specified position
         in
256      * the specified array and reverses the run if it is descending
         (ensuring
257      * that the run will always be ascending when the method returns).
258      *
259      * A run is the longest ascending sequence with:
260      *
```

```
261       *    a[lo] <= a[lo + 1] <= a[lo + 2] <= ...
262       *
263       * or the longest descending sequence with:
264       *
265       *    a[lo] >  a[lo + 1] >  a[lo + 2] >  ...
266       *
267       * For its intended use in a stable mergesort, the strictness of the
268       * definition of "descending" is needed so that the call can safely
269       * reverse a descending sequence without violating stability.
270       *
271       * @param a the array in which a run is to be counted and possibly
          reversed
272       * @param lo index of the first element in the run
273       * @param hi index after the last element that may be contained in
          the run.
274                 It is required that @code{lo < hi}.
275       * @return  the length of the run beginning at the specified
          position in
276       *          the specified array
277       */
278      @SuppressWarnings("unchecked")
279      private static int countRunAndMakeAscending(Object[] a, int lo, int
          hi) {
280          if (DEBUG) assert lo < hi;
281          int runHi = lo + 1;
282          if (runHi == hi)
283              return 1;
284
285          // Find end of run, and reverse range if descending
286          if (((Comparable) a[runHi++]).compareTo(a[lo]) < 0) { //
             Descending
287              while(runHi < hi && ((Comparable)
                 a[runHi]).compareTo(a[runHi - 1]) < 0)
288                  runHi++;
289              reverseRange(a, lo, runHi);
290          } else {                              // Ascending
291              while (runHi < hi && ((Comparable)
                 a[runHi]).compareTo(a[runHi - 1]) >= 0)
292                  runHi++;
293          }
294
295          return runHi - lo;
296      }
297
298      /**
299       * Reverse the specified range of the specified array.
300       *
301       * @param a the array in which a range is to be reversed
302       * @param lo the index of the first element in the range to be
          reversed
303       * @param hi the index after the last element in the range to be
          reversed
304       */
305      private static void reverseRange(Object[] a, int lo, int hi) {
306          hi--;
307          while (lo < hi) {
308              Object t = a[lo];
309              a[lo++] = a[hi];
310              a[hi--] = t;
311          }
```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```
312     }
313
314     /**
315      * Returns the minimum acceptable run length for an array of the specified
316      * length. Natural runs shorter than this will be extended with
317      * {@link #binarySort}.
318      *
319      * Roughly speaking, the computation is:
320      *
321      *  If n < MIN_MERGE, return n (it's too small to bother with fancy stuff).
322      *  Else if n is an exact power of 2, return MIN_MERGE/2.
323      *  Else return an int k, MIN_MERGE/2 <= k <= MIN_MERGE, such that n/k
324      *   is close to, but strictly less than, an exact power of 2.
325      *
326      * For the rationale, see listsort.txt.
327      *
328      * @param n the length of the array to be sorted
329      * @return the length of the minimum run to be merged
330      */
331     private static int minRunLength(int n) {
332         if (DEBUG) assert n >= 0;
333         int r = 0;      // Becomes 1 if any 1 bits are shifted off
334         while (n >= MIN_MERGE) {
335             r |= (n & 1);
336             n >>= 1;
337         }
338         return n + r;
339     }
340
341     /**
342      * Pushes the specified run onto the pending-run stack.
343      *
344      * @param runBase index of the first element in the run
345      * @param runLen  the number of elements in the run
346      */
347     private void pushRun(int runBase, int runLen) {
348         this.runBase[stackSize] = runBase;
349         this.runLen[stackSize] = runLen;
350         stackSize++;
351     }
352
353     /**
354      * Examines the stack of runs waiting to be merged and merges adjacent runs
355      * until the stack invariants are reestablished:
356      *
357      *     1. runLen[i - 3] > runLen[i - 2] + runLen[i - 1]
358      *     2. runLen[i - 2] > runLen[i - 1]
359      *
360      * This method is called each time a new run is pushed onto the stack,
361      * so the invariants are guaranteed to hold for i < stackSize upon
362      * entry to the method.
363      */
364     private void mergeCollapse() {
365         while (stackSize > 1) {
366             int n = stackSize - 2;
```

```
367             if (n > 0 && runLen[n-1] <= runLen[n] + runLen[n+1]) {
368                 if (runLen[n - 1] < runLen[n + 1])
369                     n--;
370                 mergeAt(n);
371             } else if (runLen[n] <= runLen[n + 1]) {
372                 mergeAt(n);
373             } else {
374                 break; // Invariant is established
375             }
376         }
377     }
378
379     /**
380      * Merges all runs on the stack until only one remains.  This method
         is
381      * called once, to complete the sort.
382      */
383     private void mergeForceCollapse() {
384         while (stackSize > 1) {
385             int n = stackSize - 2;
386             if (n > 0 && runLen[n - 1] < runLen[n + 1])
387                 n--;
388             mergeAt(n);
389         }
390     }
391
392     /**
393      * Merges the two runs at stack indices i and i+1.  Run i must be
394      * the penultimate or antepenultimate run on the stack.  In other
         words,
395      * i must be equal to stackSize-2 or stackSize-3.
396      *
397      * @param i stack index of the first of the two runs to merge
398      */
399     @SuppressWarnings("unchecked")
400     private void mergeAt(int i) {
401         if (DEBUG) assert stackSize >= 2;
402         if (DEBUG) assert i >= 0;
403         if (DEBUG) assert i == stackSize - 2 || i == stackSize - 3;
404
405         int base1 = runBase[i];
406         int len1 = runLen[i];
407         int base2 = runBase[i + 1];
408         int len2 = runLen[i + 1];
409         if (DEBUG) assert len1 > 0 && len2 > 0;
410         if (DEBUG) assert base1 + len1 == base2;
411
412         /*
413          * Record the length of the combined runs; if i is the 3rd-last
414          * run now, also slide over the last run (which isn't involved
415          * in this merge).  The current run (i+1) goes away in any case.
416          */
417         runLen[i] = len1 + len2;
418         if (i == stackSize - 3) {
419             runBase[i + 1] = runBase[i + 2];
420             runLen[i + 1] = runLen[i + 2];
421         }
422         stackSize--;
423
424         /*
```

```
425          * Find where the first element of run2 goes in run1. Prior
             elements
426          * in run1 can be ignored (because they're already in place).
427          */
428         int k = gallopRight((Comparable<Object>) a[base2], a, base1,
             len1, 0);
429         if (DEBUG) assert k >= 0;
430         base1 += k;
431         len1 -= k;
432         if (len1 == 0)
433             return;
434
435         /*
436          * Find where the last element of run1 goes in run2. Subsequent
             elements
437          * in run2 can be ignored (because they're already in place).
438          */
439         len2 = gallopLeft((Comparable<Object>) a[base1 + len1 - 1], a,
440                 base2, len2, len2 - 1);
441         if (DEBUG) assert len2 >= 0;
442         if (len2 == 0)
443             return;
444
445         // Merge remaining runs, using tmp array with min(len1, len2)
             elements
446         if (len1 <= len2)
447             mergeLo(base1, len1, base2, len2);
448         else
449             mergeHi(base1, len1, base2, len2);
450     }
451
452     /**
453      * Locates the position at which to insert the specified key into
         the
454      * specified sorted range; if the range contains an element equal to
         key,
455      * returns the index of the leftmost equal element.
456      *
457      * @param key the key whose insertion point to search for
458      * @param a the array in which to search
459      * @param base the index of the first element in the range
460      * @param len the length of the range; must be > 0
461      * @param hint the index at which to begin the search, 0 <= hint <
         n.
462      *     The closer hint is to the result, the faster this method will
         run.
463      * @return the int k,  0 <= k <= n such that a[b + k - 1] < key <=
         a[b + k],
464      *    pretending that a[b - 1] is minus infinity and a[b + n] is
         infinity.
465      *    In other words, key belongs at index b + k; or in other words,
466      *    the first k elements of a should precede key, and the last n -
         k
467      *    should follow it.
468      */
469     private static int gallopLeft(Comparable<Object> key, Object[] a,
470             int base, int len, int hint) {
471         if (DEBUG) assert len > 0 && hint >= 0 && hint < len;
472
473         int lastOfs = 0;
```

```
474         int ofs = 1;
475         if (key.compareTo(a[base + hint]) > 0) {
476             // Gallop right until a[base+hint+lastOfs] < key <=
                a[base+hint+ofs]
477             int maxOfs = len - hint;
478             while (ofs < maxOfs && key.compareTo(a[base + hint + ofs]) >
                0) {
479                 lastOfs = ofs;
480                 ofs = (ofs << 1) + 1;
481                 if (ofs <= 0)   // int overflow
482                     ofs = maxOfs;
483             }
484             if (ofs > maxOfs)
485                 ofs = maxOfs;
486
487             // Make offsets relative to base
488             lastOfs += hint;
489             ofs += hint;
490         } else { // key <= a[base + hint]
491             // Gallop left until a[base+hint-ofs] < key <=
                a[base+hint-lastOfs]
492             final int maxOfs = hint + 1;
493             while (ofs < maxOfs && key.compareTo(a[base + hint - ofs])
                <= 0) {
494                 lastOfs = ofs;
495                 ofs = (ofs << 1) + 1;
496                 if (ofs <= 0)   // int overflow
497                     ofs = maxOfs;
498             }
499             if (ofs > maxOfs)
500                 ofs = maxOfs;
501
502             // Make offsets relative to base
503             int tmp = lastOfs;
504             lastOfs = hint - ofs;
505             ofs = hint - tmp;
506         }
507         if (DEBUG) assert -1 <= lastOfs && lastOfs < ofs && ofs <= len;
508
509         /*
510          * Now a[base+lastOfs] < key <= a[base+ofs], so key belongs
                somewhere
511          * to the right of lastOfs but no farther right than ofs.  Do a
                binary
512          * search, with invariant a[base + lastOfs - 1] < key <= a[base
                + ofs].
513          */
514         lastOfs++;
515         while (lastOfs < ofs) {
516             int m = lastOfs + ((ofs - lastOfs) >>> 1);
517
518             if (key.compareTo(a[base + m]) > 0)
519                 lastOfs = m + 1;  // a[base + m] < key
520             else
521                 ofs = m;          // key <= a[base + m]
522         }
523         if (DEBUG) assert lastOfs == ofs;    // so a[base + ofs - 1] <
                key <= a[base + ofs]
524         return ofs;
525     }
```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```
526
527     /**
528      * Like gallopLeft, except that if the range contains an element equal to
529      * key, gallopRight returns the index after the rightmost equal element.
530      *
531      * @param key the key whose insertion point to search for
532      * @param a the array in which to search
533      * @param base the index of the first element in the range
534      * @param len the length of the range; must be > 0
535      * @param hint the index at which to begin the search, 0 <= hint < n.
536      *     The closer hint is to the result, the faster this method will run.
537      * @return the int k,  0 <= k <= n such that a[b + k - 1] <= key < a[b + k]
538      */
539     private static int gallopRight(Comparable<Object> key, Object[] a,
540             int base, int len, int hint) {
541         if (DEBUG) assert len > 0 && hint >= 0 && hint < len;
542
543         int ofs = 1;
544         int lastOfs = 0;
545         if (key.compareTo(a[base + hint]) < 0) {
546             // Gallop left until a[b+hint - ofs] <= key < a[b+hint - lastOfs]
547             int maxOfs = hint + 1;
548             while (ofs < maxOfs && key.compareTo(a[base + hint - ofs]) < 0) {
549                 lastOfs = ofs;
550                 ofs = (ofs << 1) + 1;
551                 if (ofs <= 0)   // int overflow
552                     ofs = maxOfs;
553             }
554             if (ofs > maxOfs)
555                 ofs = maxOfs;
556
557             // Make offsets relative to b
558             int tmp = lastOfs;
559             lastOfs = hint - ofs;
560             ofs = hint - tmp;
561         } else { // a[b + hint] <= key
562             // Gallop right until a[b+hint + lastOfs] <= key < a[b+hint + ofs]
563             int maxOfs = len - hint;
564             while (ofs < maxOfs && key.compareTo(a[base + hint + ofs]) >= 0) {
565                 lastOfs = ofs;
566                 ofs = (ofs << 1) + 1;
567                 if (ofs <= 0)   // int overflow
568                     ofs = maxOfs;
569             }
570             if (ofs > maxOfs)
571                 ofs = maxOfs;
572
573             // Make offsets relative to b
574             lastOfs += hint;
575             ofs += hint;
576         }
```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```
577         if (DEBUG) assert -1 <= lastOfs && lastOfs < ofs && ofs <= len;
578
579         /*
580          * Now a[b + lastOfs] <= key < a[b + ofs], so key belongs somewhere to
581          * the right of lastOfs but no farther right than ofs.  Do a binary
582          * search, with invariant a[b + lastOfs - 1] <= key < a[b + ofs].
583          */
584         lastOfs++;
585         while (lastOfs < ofs) {
586             int m = lastOfs + ((ofs - lastOfs) >>> 1);
587
588             if (key.compareTo(a[base + m]) < 0)
589                 ofs = m;          // key < a[b + m]
590             else
591                 lastOfs = m + 1;  // a[b + m] <= key
592         }
593         if (DEBUG) assert lastOfs == ofs;    // so a[b + ofs - 1] <= key < a[b + ofs]
594         return ofs;
595     }
596
597     /**
598      * Merges two adjacent runs in place, in a stable fashion.  The first
599      * element of the first run must be greater than the first element of the
600      * second run (a[base1] > a[base2]), and the last element of the first run
601      * (a[base1 + len1-1]) must be greater than all elements of the second run.
602      *
603      * For performance, this method should be called only when len1 <= len2;
604      * its twin, mergeHi should be called if len1 >= len2.  (Either method
605      * may be called if len1 == len2.)
606      *
607      * @param base1 index of first element in first run to be merged
608      * @param len1  length of first run to be merged (must be > 0)
609      * @param base2 index of first element in second run to be merged
610      *        (must be aBase + aLen)
611      * @param len2  length of second run to be merged (must be > 0)
612      */
613     @SuppressWarnings("unchecked")
614     private void mergeLo(int base1, int len1, int base2, int len2) {
615         if (DEBUG) assert len1 > 0 && len2 > 0 && base1 + len1 == base2;
616
617         // Copy first run into temp array
618         Object[] a = this.a; // For performance
619         Object[] tmp = ensureCapacity(len1);
620         System.arraycopy(a, base1, tmp, 0, len1);
621
622         int cursor1 = 0;       // Indexes into tmp array
623         int cursor2 = base2;   // Indexes int a
624         int dest = base1;      // Indexes int a
625
626         // Move first element of second run and deal with degenerate
```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```
            cases
627         a[dest++] = a[cursor2++];
628         if (--len2 == 0) {
629             System.arraycopy(tmp, cursor1, a, dest, len1);
630             return;
631         }
632         if (len1 == 1) {
633             System.arraycopy(a, cursor2, a, dest, len2);
634             a[dest + len2] = tmp[cursor1]; // Last elt of run 1 to end
                of merge
635             return;
636         }
637
638         int minGallop = this.minGallop;  // Use local variable for
            performance
639     outer:
640         while (true) {
641             int count1 = 0; // Number of times in a row that first run
                won
642             int count2 = 0; // Number of times in a row that second run
                won
643
644             /*
645              * Do the straightforward thing until (if ever) one run
                 starts
646              * winning consistently.
647              */
648             do {
649                 if (DEBUG) assert len1 > 1 && len2 > 0;
650                 if (((Comparable) a[cursor2]).compareTo(tmp[cursor1]) <
                    0) {
651                     a[dest++] = a[cursor2++];
652                     count2++;
653                     count1 = 0;
654                     if (--len2 == 0)
655                         break outer;
656                 } else {
657                     a[dest++] = tmp[cursor1++];
658                     count1++;
659                     count2 = 0;
660                     if (--len1 == 1)
661                         break outer;
662                 }
663             } while ((count1 | count2) < minGallop);
664
665             /*
666              * One run is winning so consistently that galloping may be
                 a
667              * huge win. So try that, and continue galloping until (if
                 ever)
668              * neither run appears to be winning consistently anymore.
669              */
670             do {
671                 if (DEBUG) assert len1 > 1 && len2 > 0;
672                 count1 = gallopRight((Comparable) a[cursor2], tmp,
                    cursor1, len1, 0);
673                 if (count1 != 0) {
674                     System.arraycopy(tmp, cursor1, a, dest, count1);
675                     dest += count1;
676                     cursor1 += count1;
```

```
677                     len1 -= count1;
678                     if (len1 <= 1)  // len1 == 1 || len1 == 0
679                         break outer;
680                 }
681                 a[dest++] = a[cursor2++];
682                 if (--len2 == 0)
683                     break outer;
684
685                 count2 = gallopLeft((Comparable) tmp[cursor1], a,
                    cursor2, len2, 0);
686                 if (count2 != 0) {
687                     System.arraycopy(a, cursor2, a, dest, count2);
688                     dest += count2;
689                     cursor2 += count2;
690                     len2 -= count2;
691                     if (len2 == 0)
692                         break outer;
693                 }
694                 a[dest++] = tmp[cursor1++];
695                 if (--len1 == 1)
696                     break outer;
697                 minGallop--;
698             } while (count1 >= MIN_GALLOP | count2 >= MIN_GALLOP);
699             if (minGallop < 0)
700                 minGallop = 0;
701             minGallop += 2;  // Penalize for leaving gallop mode
702         }  // End of "outer" loop
703         this.minGallop = minGallop < 1 ? 1 : minGallop;  // Write back
            to field
704
705         if (len1 == 1) {
706             if (DEBUG) assert len2 > 0;
707             System.arraycopy(a, cursor2, a, dest, len2);
708             a[dest + len2] = tmp[cursor1]; //  Last elt of run 1 to end
                of merge
709         } else if (len1 == 0) {
710             throw new IllegalArgumentException(
711                 "Comparison method violates its general contract!");
712         } else {
713             if (DEBUG) assert len2 == 0;
714             if (DEBUG) assert len1 > 1;
715             System.arraycopy(tmp, cursor1, a, dest, len1);
716         }
717     }
718
719     /**
720      * Like mergeLo, except that this method should be called only if
721      * len1 >= len2; mergeLo should be called if len1 <= len2.  (Either
           method
722      * may be called if len1 == len2.)
723      *
724      * @param base1 index of first element in first run to be merged
725      * @param len1  length of first run to be merged (must be > 0)
726      * @param base2 index of first element in second run to be merged
727      *        (must be aBase + aLen)
728      * @param len2  length of second run to be merged (must be > 0)
729      */
730     @SuppressWarnings("unchecked")
731     private void mergeHi(int base1, int len1, int base2, int len2) {
732         if (DEBUG) assert len1 > 0 && len2 > 0 && base1 + len1 == base2;
```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```
733
734         // Copy second run into temp array
735         Object[] a = this.a; // For performance
736         Object[] tmp = ensureCapacity(len2);
737         System.arraycopy(a, base2, tmp, 0, len2);
738
739         int cursor1 = base1 + len1 - 1;  // Indexes into a
740         int cursor2 = len2 - 1;          // Indexes into tmp array
741         int dest = base2 + len2 - 1;     // Indexes into a
742
743         // Move last element of first run and deal with degenerate cases
744         a[dest--] = a[cursor1--];
745         if (--len1 == 0) {
746             System.arraycopy(tmp, 0, a, dest - (len2 - 1), len2);
747             return;
748         }
749         if (len2 == 1) {
750             dest -= len1;
751             cursor1 -= len1;
752             System.arraycopy(a, cursor1 + 1, a, dest + 1, len1);
753             a[dest] = tmp[cursor2];
754             return;
755         }
756
757         int minGallop = this.minGallop;  // Use local variable for
            performance
758     outer:
759         while (true) {
760             int count1 = 0; // Number of times in a row that first run
                won
761             int count2 = 0; // Number of times in a row that second run
                won
762
763             /*
764              * Do the straightforward thing until (if ever) one run
765              * appears to win consistently.
766              */
767             do {
768                 if (DEBUG) assert len1 > 0 && len2 > 1;
769                 if (((Comparable) tmp[cursor2]).compareTo(a[cursor1]) <
                    0) {
770                     a[dest--] = a[cursor1--];
771                     count1++;
772                     count2 = 0;
773                     if (--len1 == 0)
774                         break outer;
775                 } else {
776                     a[dest--] = tmp[cursor2--];
777                     count2++;
778                     count1 = 0;
779                     if (--len2 == 1)
780                         break outer;
781                 }
782             } while ((count1 | count2) < minGallop);
783
784             /*
785              * One run is winning so consistently that galloping may be
                a
786              * huge win. So try that, and continue galloping until (if
                ever)
```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```
787              * neither run appears to be winning consistently anymore.
788              */
789             do {
790                 if (DEBUG) assert len1 > 0 && len2 > 1;
791                 count1 = len1 - gallopRight((Comparable) tmp[cursor2],
                    a, base1, len1, len1 - 1);
792                 if (count1 != 0) {
793                     dest -= count1;
794                     cursor1 -= count1;
795                     len1 -= count1;
796                     System.arraycopy(a, cursor1 + 1, a, dest + 1,
                        count1);
797                     if (len1 == 0)
798                         break outer;
799                 }
800                 a[dest--] = tmp[cursor2--];
801                 if (--len2 == 1)
802                     break outer;
803
804                 count2 = len2 - gallopLeft((Comparable) a[cursor1], tmp,
                    0, len2, len2 - 1);
805                 if (count2 != 0) {
806                     dest -= count2;
807                     cursor2 -= count2;
808                     len2 -= count2;
809                     System.arraycopy(tmp, cursor2 + 1, a, dest + 1,
                        count2);
810                     if (len2 <= 1)
811                         break outer; // len2 == 1 || len2 == 0
812                 }
813                 a[dest--] = a[cursor1--];
814                 if (--len1 == 0)
815                     break outer;
816                 minGallop--;
817             } while (count1 >= MIN_GALLOP | count2 >= MIN_GALLOP);
818             if (minGallop < 0)
819                 minGallop = 0;
820             minGallop += 2;  // Penalize for leaving gallop mode
821         }  // End of "outer" loop
822         this.minGallop = minGallop < 1 ? 1 : minGallop;  // Write back
            to field
823
824         if (len2 == 1) {
825             if (DEBUG) assert len1 > 0;
826             dest -= len1;
827             cursor1 -= len1;
828             System.arraycopy(a, cursor1 + 1, a, dest + 1, len1);
829             a[dest] = tmp[cursor2];  // Move first elt of run2 to front
                of merge
830         } else if (len2 == 0) {
831             throw new IllegalArgumentException(
832                 "Comparison method violates its general contract!");
833         } else {
834             if (DEBUG) assert len1 == 0;
835             if (DEBUG) assert len2 > 0;
836             System.arraycopy(tmp, 0, a, dest - (len2 - 1), len2);
837         }
838     }
839
840     /**
```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```
841      * Ensures that the external array tmp has at least the specified
842      * number of elements, increasing its size if necessary.  The size
843      * increases exponentially to ensure amortized linear time
         complexity.
844      *
845      * @param minCapacity the minimum required capacity of the tmp array
846      * @return tmp, whether or not it grew
847      */
848     private Object[]  ensureCapacity(int minCapacity) {
849         if (tmp.length < minCapacity) {
850             // Compute smallest power of 2 > minCapacity
851             int newSize = minCapacity;
852             newSize |= newSize >> 1;
853             newSize |= newSize >> 2;
854             newSize |= newSize >> 4;
855             newSize |= newSize >> 8;
856             newSize |= newSize >> 16;
857             newSize++;
858
859             if (newSize < 0) // Not bloody likely!
860                 newSize = minCapacity;
861             else
862                 newSize = Math.min(newSize, a.length >>> 1);
863
864             @SuppressWarnings({"unchecked", "UnnecessaryLocalVariable"})
865             Object[] newArray = new Object[newSize];
866             tmp = newArray;
867         }
868         return tmp;
869     }
870
871     /**
872      * Checks that fromIndex and toIndex are in range, and throws an
873      * appropriate exception if they aren't.
874      *
875      * @param arrayLen the length of the array
876      * @param fromIndex the index of the first element of the range
877      * @param toIndex the index after the last element of the range
878      * @throws IllegalArgumentException if fromIndex > toIndex
879      * @throws ArrayIndexOutOfBoundsException if fromIndex < 0
880      *         or toIndex > arrayLen
881      */
882     private static void rangeCheck(int arrayLen, int fromIndex, int
        toIndex) {
883         if (fromIndex > toIndex)
884             throw new IllegalArgumentException("fromIndex(" + fromIndex
                 +
885                     ") > toIndex(" + toIndex+")");
886         if (fromIndex < 0)
887             throw new ArrayIndexOutOfBoundsException(fromIndex);
888         if (toIndex > arrayLen)
889             throw new ArrayIndexOutOfBoundsException(toIndex);
890     }
891 }
```

EXHIBIT E

java.io

Class FileReader

```
java.lang.Object
  └ java.io.Reader
      └ java.io.InputStreamReader
          └ java.io.FileReader
```

All Implemented Interfaces:

Closeable, Readable

```
public class FileReader
extends InputStreamReader
```

Convenience class for reading character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate. To specify these values yourself, construct an InputStreamReader on a FileInputStream.

`FileReader` is meant for reading streams of characters. For reading streams of raw bytes, consider using a `FileInputStream`.

Since:

JDK1.1

See Also:

`InputStreamReader`, `FileInputStream`

Field Summary

Fields inherited from class java.io.Reader
lock

Constructor Summary

FileReader(File file) Creates a new FileReader, given the File to read from.
FileReader(FileDescriptor fd) Creates a new FileReader, given the FileDescriptor to read from.
FileReader(String fileName)

Creates a new `FileReader`, given the name of the file to read from.

Method Summary

Methods inherited from class java.io.InputStreamReader
close, getEncoding, read, read, ready

Methods inherited from class java.io.Reader
mark, markSupported, read, read, reset, skip

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

FileReader

```
public FileReader(String fileName)
           throws FileNotFoundException
```

Creates a new `FileReader`, given the name of the file to read from.

Parameters:

`fileName` - the name of the file to read from

Throws:

`FileNotFoundException` - if the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

FileReader

```
public FileReader(File file)
           throws FileNotFoundException
```

Creates a new `FileReader`, given the `File` to read from.

Parameters:

`file` - the `File` to read from

Throws:

`FileNotFoundException` - if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

FileReader

```
public FileReader(FileDescriptor fd)
```

Creates a new `FileReader`, given the `FileDescriptor` to read from.

Parameters:

`fd` - the FileDescriptor to read from