KEKER & VAN NEST LLP
ROBERT A. VAN NEST - #84065
rvannest@kvn.com
CHRISTA M. ANDERSON - #184325
canderson@kvn.com
633 Battery Street
San Francisco, CA 94111-1809
Telephone: 415.391.5400
Facsimile: 415.397.7188

KING & SPALDING LLP
DONALD F. ZIMMER, JR. - #112279
fzimmer@kslaw.com
CHERYL A. SABNIS - #224323
csabnis@kslaw.com
101 Second St., Suite 2300
San Francisco, CA 94105
Telephone: 415.318.1200
Facsimile: 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
Telephone: 212.556.2100
Facsimile: 212.556.2222

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

EXHIBIT 3 TO REPLY DECLARATION OF OWEN ASTRACHAN IN SUPPORT OF DEFENDANT GOOGLE INC.'S MOTION FOR SUMMARY JUDGMENT ON COUNT VIII OF PLAINTIFF ORACLE AMERICA'S AMENDED COMPLAINT

Judge: Hon. William Alsup

Hearing: 2:00 p.m., September 15, 2011

PUBLICLY FILED VERSION
REDACTED

EXHIBIT 3

ROBERT A. VAN NEST — #84065
rvannest@kvn.com
CHRISTA M. ANDERSON — #184325
canderson@kvn.com
KEKER & VAN NEST LLP
710 Sansome Street
San Francisco, CA 94111-1704
Telephone: (415) 391-5400
Facsimile: (415) 397-7188

DONALD F. ZIMMER, JR. (SBN 112279)
fzimmer@kslaw.com
CHERYL A. SABNIS (SBN 224323)
csabnis@kslaw.com
KING & SPALDING LLP
101 Second Street – Suite 2300
San Francisco, CA 94105
Telephone: (415) 318-1200
Facsimile: (415) 318-1300

Attorneys for Defendant
GOOGLE INC.

SCOTT T. WEINGAERTNER (*Pro Hac Vice*)
sweingaertner@kslaw.com
ROBERT F. PERRY
rperry@kslaw.com
BRUCE W. BABER (*Pro Hac Vice*)
bbaber@kslaw.com
KING & SPALDING LLP
1185 Avenue of the Americas
New York, NY 10036-4003
Telephone: (212) 556-2100
Facsimile: (212) 556-2222

IAN C. BALLON (SBN 141819)
ballon@gtlaw.com
VALERIE HO (SBN 200505)
hov@gtlaw.com
HEATHER MEEKER (SBN 172148)
meekerh@gtlaw.com
GREENBERG TRAURIG, LLP
1900 University Avenue
East Palo Alto, CA 94303
Telephone: (650) 328-8500
Facsimile: (650) 328-8508

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

Honorable Judge William Alsup

REBUTTAL EXPERT REPORT OF DR. OWEN ASTRACHAN

CONFIDENTIAL PURSUANT TO PROTECTIVE ORDER-
HIGHLY CONFIDENTIAL - SOURCE CODE

I. INTRODUCTION 2

II. DOCUMENTS AND INFORMATION CONSIDERED 2

III. BRIEF SUMMARY OF MY OPINIONS 3

IV. THE DISTINCTION BETWEEN AN API AND ITS IMPLEMENTATION 3

V. GOOGLE'S IMPLEMENTATION OF THE APIs AT ISSUE IS NOT VIRTUALLY IDENTICAL OR SUBSTANTIALLY SIMILAR TO ORACLE'S IMPLEMENTATION 14

VI. THE VARIOUS JAVA VERSIONS THAT ORACLE ALLEGES WERE INFRINGED CONTAIN THE SAME APIs AS EARLIER VERSIONS OR VERSIONS FOR OTHER OPERATING SYSTEMS 18

VII. PARAMETER NAMES ARE FUNCTIONAL AND NOT CREATIVE 19

VIII. THE ORGANIZATION OF PACKAGES IS FUNCTIONAL AND DOES NOT CONTAIN CREATIVE EXPRESSION 20

IX. C#, LIKE JAVA, IS UNPROTECTABLE, AND IS ALSO AVAILABLE AS AN OPEN SPECIFICATION AND IMPLEMENTATION 22

X. ORACLE'S ANALYSIS OF THE FILES AT ISSUE DOES NOT DISCUSS THEIR QUALITATIVE OR QUANTITATIVE IMPORTANCE, WITH ONE EXCEPTION THAT IS INCORRECT 23

EXHIBIT F: COMPARISON OF ANDROID AND ORACLE ZIPFILE.GETINPUTSTREAM

EXHIBIT G: PUBLICPRIVATEANALYZER.PY SOURCE CODE

I. INTRODUCTION

1. I have been asked by Google to review the expert reports of John C. Mitchell, Marc Visnick, and Alan Purdy and, in addition to those opinions offered in my July 29, 2011 Opening Expert Report ("Opening Report"), to opine on the conclusions set forth in those reports, and whether Oracle's allegedly copyrighted works relating to the Android platform are virtually identical or substantially similar to the Java platform.
2. My qualifications, set forth in my Opening Report, are incorporated herein by reference.
3. I understand that I may be asked by Google to review further submissions related to copyright issues from Oracle's experts, and to provide my opinions on issues raised by any such submissions.
4. I understand that I may be called upon to testify in this case regarding my opinions and analyses set forth in this report. If called upon to testify, I may use various demonstratives, including tables or drawings, to assist in presenting my testimony.
5. As set forth in my Opening Report, my compensation does not depend in any way on the outcome of this litigation.

II. DOCUMENTS AND INFORMATION CONSIDERED

6. My opinions are based on my relevant knowledge and experience, the documents identified in Exhibit B to my Opening Report, as well as review of the following documents and information:
   a. Opening Expert Report of John C. Mitchell Regarding Copyright, Opening Expert Report of Alan Purdy Regarding Copyright, and Opening Expert Report of Marc Visnick Regarding Copyright, all dated July 29, 2011.
   b. "Design Patterns: Elements of Reusable Object-Oriented Software," by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
   c. Mono Website page on ECMA, *available at* http://www.mono-project.com/ECMA; Microsoft Open Specifications, *available at* http://www.microsoft.com/openspecifications/en/us/programs/community-promise/covered-specifications/default.aspx

d. "Q&A with Tim Bray," November 13, 2006, *available at* http://www.zdnet.com/blog/burnette/q-a-with-tim-bray/200?pg=3

III. BRIEF SUMMARY OF MY OPINIONS

7. Based upon my review of the material set forth in Section II, I disagree with Prof. Mitchell's conclusion regarding whether elements of the Java API specifications contain copyrightable expression. I also disagree with Prof. Mitchell's conclusion that the Android source code is substantially similar to Oracle's copyrighted source code. It is my opinion that Google's implementation of the APIs at issue is neither virtually identical nor substantially similar to Oracle's implementation.

IV. THE DISTINCTION BETWEEN AN API AND ITS IMPLEMENTATION

8. As discussed in paragraph 52 of my Opening Report, every API, including the Java APIs at issue in this case, exists in two forms: the method declaration of the API (comprised of those elements — name, arguments, and return — described in paragraphs 40-47 of my Opening Report) and the implementation of the API. The implementation is the actual underlying source code that implements the API and allows the API to function. Any two implementations of the same API will contain some similar portions, because each implementation must include exactly the same method declaration, including all the elements of the declaration, such as the arguments and return values, in order to be compatible. However, the overall source code may — and indeed does — differ significantly from implementation to implementation. Even if only a small fraction of the source code of two implementations is identical, the remaining code may appear similar to the untrained eye, both because certain key lines (the method, package, and class declarations) must be the same, and because practical considerations will constrain the expression of the code implementing the functionality. For example, there may be both efficient and inefficient ways to implement a given method, but programmers will typically choose the most efficient way. Similarly, coding standards relating to indentation, punctuation, and formatting will also constrain how code is written. In addition, because many programmers have learned by studying and reading source code

written by others, they typically write code in a similar style. Returning to the car analogy that is set forth in my Opening Report, there may be unusual ways to power a car (hydrogen, rotary engines, etc.), but in most cases the solutions will end up looking similar to other implementations for practical reasons due to standard design practices, and not because the car manufacturers were copying from each other.

9. An API implementation that uses only the necessary API components, but does not repeat the underlying implementation, is an "independent" implementation. A Ford and a Chevy are, in this sense, independent implementations of a car — while they both provide drivers with the same gas pedal and steering interface to the underlying functionality, Chevy engineers likely did not photocopy Ford blueprints in order to build the Chevy's engine and steering mechanism. Similarly, the fact that virtually every modern computer application supports common keyboard commands like Ctl+C, Ctl+V, and Ctl+P does not prove that the programmers used each other's implementation source code. Instead, they have each re-implemented the functionality in a way that makes sense for their circumstances, reusing only the "interface" of the keyboard commands.

10. To illustrate how an API must be identical across Java implementations, even while the implementations differ, I will use three examples. Before doing that, it is first useful to provide an analogy that will help to explain the source code being discussed here. In particular, the different implementations of APIs are similar to different sets of driving directions that take someone from point A to point B. In this analogy, the starting point, A, is like an argument, and the ending point, B, is like a return value. Like an API implementation that is constrained by the method declaration, every set of directions that goes from point A to point B will begin and end the same way ("leave the parking lot at point A," "enter the parking lot at point B"); however, there may be many other variations between the directions. For example, one set of directions might take the highway, while another might take back roads. One set of directions might prioritize giving directions in the fewest number of turns, while another set of directions might take more turns, but use those extra steps to avoid an area of high traffic. Another pair of

directions might be identical, except that one adds special steps to be taken during rush hour.

11. Of course, directions, like computer programs, are subject to practical constraints because they are process-driven expressions. You could write directions from San Jose to San Francisco that go by way of New York, but those directions would be so inefficient that, while possible, they are not a realistic option in practice. And in some cases, there will be so few options for how to get from point A to point B that in fact there is only one way to write the directions.

12. The source code discussed in the following examples is similar. Each implementation tells the underlying computer how to get to a particular result, but as I will explain, the Android "directions" generally are different from the Oracle "directions." Although they get the same result — starting from the inputs and ending at the return values — they take different steps to get there.

BEGIN ORACLE SOURCE CODE - HIGHLY CONFIDENTIAL

13. The first example is one I have used earlier: the Math.abs function. As discussed in paragraphs 57-60 of my Opening Report, the absolute value of an integer is essentially the magnitude of the integer, *i.e.*, the distance of the integer from zero. Similarly, paragraph 60 of my Opening Report states that the *declaration* of the method (the function name, return type, and parameter type) is specified as part of the Math.abs API and must be the same in any compatible implementation of the Math.abs API. The following chart (from paragraph 61 of my Opening Report) shows the various identical method declarations for abs from the various implementations of Java:

Java:	public static int abs(int a)
Harmony:	public static int abs(int i)
GNU Classpath:	public static int abs(int i)
Android:	public static int abs(int i)

(As I explain in paragraph 15, below, the variable name chosen for the parameter in the parentheses need not be the same, and, in fact, the variable name in the Android implementation is different than in Oracle's implementation.)

14. Not surprisingly, because the concept is so simple ("if the number is negative, give the positive version of it") the implementations are very brief — all it takes is one line for the declaration, and one line for the actual functionality. Despite this simplicity and brevity, Oracle and Android's implementations of Java are different. The table below shows the Android source code that implements the Math.abs function in the java.lang.Math class compared to the source code that implements JDK1.5 code.

Android Math.abs	**Oracle JDK 1.5 Math.abs**

`public static int abs(int i) {`
`    return i >= 0 ? i : -i;`
`}`
	`public static int abs(int a) {`
	`    return (a < 0) ? -a : a;`
	`}`

15. As required by the API, the first line of the method — the function name, return type, and parameter type — are essentially identical in both implementations. The name of the parameter — *a* for the JDK1.5 implementation and *i* in the Android implementation — is the only thing different. The parameter name can be different because the name of the parameter is not part of the API. The parameter type, *int*, on the other hand, must be the same if the two implementations are to be compatible.

16. The actual implementation of the method — the second line, shown in blue — is how the absolute value is calculated. Each of these lines of code is different, but nevertheless correct. Put into English, the line of code from the Android implementation translates to "if the parameter *i* is greater than or equal to zero, return *i*, otherwise return *i*'s negation." In the JDK1.5 implementation the code translates to English as "if the parameter *a* is less than zero, return *a*'s negation, otherwise return *a*." While these implementations must capture the same functionality, and bear some similarity because of the requirement that the method name and arguments be the same, they capture the functionality with different implementations.

17. The second example I will use to illustrate how the functionality expressed by an API is implemented differently is the java.lang.String class method String.compareTo. In

programming, a "string" is a sequence of characters, such as a word or sentence. The compareTo method compares two strings, in order to determine whether one string is less than, equal to, or greater than another string. In programming, a string that is "less than" another string is alphabetized first. For example, if "compareTo" was used to compare "apple" to "cat," the method would indicate that "apple" is less than "cat."

18. Below is the Android and Oracle JDK 1.5 source code that implements the compareTo method.

#	Android String.compareTo	Oracle JDK 1.5 String.compareTo
1	`public int compareTo(String string) {`	`public int compareTo(String anotherString) {`
2	`  // Code adapted from K&R, pg 101`	`  int len1 = count;`
3	`  int o1 = offset, o2 = string.offset, result;`	`  int len2 = anotherString.count;`
4	`  int end = offset + (count < string.count ? count : string.count);`	`  int n = Math.min(len1, len2);`
5	`  char[] target = string.value;`	`  char v1[] = value;`
6	`  while (o1 < end) {`	`  char v2[] = anotherString.value;`
7	`    if ((result = value[o1++] - target[o2++]) != 0) {`	`  int i = offset;`
8	`      return result;`	`  int j = anotherString.offset;`
9	`    }`	
10	`  }`	`  if (i == j) {`
11	`  return count - string.count;`	`    int k = i;`
12	`}`	`    int lim = n + i;`
13		`    while (k < lim) {`
14		`      char c1 = v1[k];`
15		`      char c2 = v2[k];`
16		`      if (c1 != c2) {`
17		`        return c1 - c2;`
18		`      }`
19		`      k++;`
20		`    }`
21		`  } else {`
22		`    while (n-- != 0) {`
23		`      char c1 = v1[i++];`
24		`      char c2 = v2[j++];`
25		`      if (c1 != c2) {`
26		`        return c1 - c2;`
27		`      }`
28		`    }`
29		`  }`
30		`  return len1 - len2;`
31		`}`

19. As noted in a comment on line 2 of the Android implementation (on the left), the Android implementation of compareTo is adapted and based on code from "K&R," a reference to "The C Programming Language," a book written by the C language's principal authors,

Brian Kernighan and Dennis Ritchie. The K&R book, and the code contained within it, were published long before the Java language existed. The body of the function — that is the code between and including the function's curly braces (*i.e.*, the "{" and "}" that mark the beginning and end of the source code for a function) — is 11 lines long.

20. In the Oracle JDK 1.5 implementation on the right, the first part of the first line of the implementation is the same as the Android implementation on the left — "`public int compareTo(String`". Again, this similarity is required for compatibility. Use of the same parameter name, however, is not required for compatibility, and so the parameter named *string* in the Android implementation is instead *anotherString* in the Oracle JDK 1.5 implementation. The Oracle implementation is also 31 lines, instead of the Android implementation's 15, indicating again that different algorithms and language features were used to reach the same result. The longer Oracle implementation is like a set of driving directions that takes complicated, twisty back roads in hopes of avoiding traffic on the big intersections, making it longer in miles, but possibly more scenic or shorter in time — in other words, possibly more efficient in other ways.

21. These two implementations are functionally identical — they compare the corresponding characters of two strings — but the actual code is very different. For example, in comparing the string "catastrophe" to "catalog" the code scans the first four characters, and finds that they are the same. It then determines the relative order of the strings by comparing the fifth characters — s in catastrophe and l in catalog. In the Android implementation the two characters compared are captured by the expressions `value[o1++]` and `target[o2++]` whereas in the JDK1.5 implementation these characters are stored in variables `c1` and `c2` and are captured by the expressions `v1[i++]` and `v2[j++]` in one part of the code and `v1[k]` and `v2[k]` in a different part of the code. In both versions of the code, once a difference in characters is detected (*i.e.*, s and l in the catastrophe and catalog example), the code need not compare further characters to determine the relative order of the strings. For example, in comparing "ant" and "bee" comparisons stop after the first characters have been examined, but when

comparing "distance" and "distant" the function can only determine the relative order after examining the seventh character of each string (c and t). Despite the similar functionality, the code that performs these comparisons and looks at the corresponding characters of each string is very different.

22. To further illustrate how the same compareTo API can be implemented in various ways, the GNU Classpath implementation of the String.compareTo method is shown in the following table, and is different from both the Android and Oracle JDK 1.5 implementations. Again, all of these sets of source code implement the same underlying functionality — they compare two strings of characters by examining each individual character until corresponding characters are different. The method name, return type, and parameter type ("`public int compareTo(String`") are again identical, as they must be for compatibility and interoperability. However, the way these sets of source code actually achieve this functionality differs significantly. For example, the Android implementation uses variable names `o1` and `o2` whereas the Classpath implementation uses variables `x` and `y`. The Android and Classpath implementations (unlike the Oracle implementation) both use a concept called a "while" loop that repeats a given operation "while" a particular condition is true, but the loop in the Android implementation uses the condition `while (o1 < end)` whereas the loop in the Classpath implementation uses the condition `while (--i > 0)`. And again, like the Android and Oracle implementations, these implementations are of different length, though the difference is much smaller. Although the logic used in the Android and Classpath implementations is the same, the implementations are very different.

#	**Android String.compareTo**	**GNU Classpath String.compareTo**
1	`public int compareTo(String string) {`	`public int compareTo(String anotherString)`
2	`// Code adapted from K&R, pg 101`	`{`
3	`int o1 = offset, o2 = string.offset, result;`	`int i = Math.min(count, anotherString.count);`
4	`int end = offset + (count < string.count ? count : string.count);`	`int x = offset;`
5	`char[] target = string.value;`	`int y = anotherString.offset;`
6	`while (o1 < end) {`	`while (--i >= 0)`
7	`if ((result = value[o1++] - target[o2++]) != 0) {`	`{`

```
        return result;

      }
    }
    return count - string.count;
}
```

```
        int result = value[x++] -
        anotherString.value[y++];
        if (result != 0)
            return result;
    }
  return count - anotherString.count;
}
```

23. The final example that I will use to compare implementations is the class ZipFile from the package java.util.zip. This class manipulates "zip" files, which are files that contain one or more other files, so that those files can be easily emailed, stored, and otherwise moved around. Because zip files are archival, they allow many files or folders to be packaged together as a single zip file. In addition, zip files are "compressed" — that is to say, a zip file is usually smaller than the sum of the sizes of the files contained in the zip file. Each of the files stored in a zip file is referred to as an "entry" in the zip file.

24. The Java API package java.util.zip contains several classes for creating, reading, writing, and manipulating zip files and the files ("entries") stored within them. In particular, I will focus on the class ZipFile and the method getInputStream from that class in order to compare and contrast an API with its implementation.

25. Among the public methods in ZipFile is one called getInputStream, which is used to "read" a zip file — *i.e.*, to access the archived and compressed contents stored in a given zip file. The getInputStream method does this by creating an "InputStream," which is a standard way for Java programmers to access files and other data sources. An InputStream is essentially a representation of a steady stream of information. Programs written in the Java language can act on these streams in a variety of ways, such as reading the next piece of data in the stream, skipping ahead to another part of the stream, and finding out how much of the stream is still available to be read. When a program written in the Java language opens, closes, and reads documents or other files, the program is using an input stream.

26. This functionality — both the ZipFile class generally and the getInputStream method specifically — can be implemented in a variety of ways. As I will discuss in more detail in paragraph 35, the implementation of a class can contain both "public" methods — or

methods that can be used by any programmer when writing programs — and “private” methods — or methods that can only be used by the code implementing the class, and used only for the purpose of implementing other parts of the class. “Public” and “private” methods can also be thought of as “external” and “internal” methods, respectively — public methods can be used from outside of the program, while private methods are “internal” to the program and can only be used by that program, not by other programs. For one class to be compatible and interoperable with another class, both must have the same public methods, but they may have different private methods and still be compatible. The Android implementation of ZipFile contains two private methods used to help implement the public methods. The Oracle JDK 1.5 implementation of ZipFile, in contrast, contains 20 private methods. The GNU Classpath ZipFile.java implementation contains seven private methods. This significant difference in the number of private methods illustrates that although the public methods of the API are similar, as they must be, the internal implementations of these methods and the class ZipFile are very different. It might be helpful to think of the Oracle implementation, which contains many private methods, as a pasta recipe that, in turn, refers to 20 other recipes — the pasta dough recipe, the pasta sauce recipe, a salad recipe to be served alongside, etc. The Android “recipe” for ZipFile, in contrast, refers only to two other recipes, incorporating the other components into the main recipe. Both the Android and Oracle recipes, in the end, create pasta, but use different processes to get there.

27. Just as the ZipFile classes in these two implementations as a whole are different, the getInputStream method in each is also different. Both the Oracle and Android implementations of the getInputStream method accomplish the same task: when given a “ZipEntry” object (*i.e.*, a reference to one of the files or directories in a zip file), return an input stream that allows the program to read that entry. However, the source code that implements Oracle JDK 1.5 method ZipFile.getInputStream, including the private helper methods and classes it uses, is 275 lines of code. Android’s implementation of the same method, including its private classes and methods, is 120 lines of code. (Because of their

length, the table with this code is attached as Exhibit F.)[1] This is a very large difference in how the methods are implemented.

28. However, it is not just the length of the two implementations that distinguish them. They are also structurally different, which can be seen by analyzing the "private" methods and classes used in the implementations. Both the Android and JDK 1.5 methods use private classes to represent the input stream that corresponds to the file or directory being read. Android's implementation uses two internal classes, named RAFstream and ZipInflaterInputStream.[2] These classes "extend" (*i.e.*, are based on and add new functionality to) other classes — InputStream and InflaterInputStream, respectively. The Oracle JDK 1.5 implementation of ZipFile.getInputStream . [redacted]. In the Java code there are three private methods (highlighted in the table below in blue) whereas there are none in the Android implementation. Again, the usage of structurally different private methods and classes indicates, in my opinion, that the implementation of these specific methods are very different, and more generally, shows how analysis of private methods can be used to help understand whether or not two given implementations are similar.

29. The methods in the source code that implements the complex task of creating the InputStream differs, but that is not the only difference — a more detailed analysis shows that even the relatively simple programming task of ensuring that the ZipFile has a name is implemented differently. The fragment of the ZipFile.getInputStream source code that implements this simple functionality is shown in the table below. The Oracle JDK 1.5 implementation [redacted]

[1] For ease of reference, in this rebuttal report I will not reuse exhibit labels used in my Opening Report.

[2] Technically these are not private - they can be used by other parts of the API package. However, the classes are only used within the ZipFile.java file, and can't be used by external programs, so they are effectively private.

[redacted].

30. The Android version has several key differences. First, it does not use a helper function — it does the work itself. Second, if the FileEntry has no name, the Android code simply returns "null" — *i.e.*, an empty value — [redacted]. Third, the Android source code finds the name of the Entry in a different way from the Oracle code — [redacted] — represented in the Android code by `entry.getName`. While this difference may look subtle (only three characters), the approach used by the Oracle code is generally considered bad style; [redacted].

Android ZipFile.getInputStream [fragment]	**Oracle JDK 1.5 ZipFile.getInputStream [fragment]**
`public InputStream getInputStream(ZipEntry entry) throws IOException {`	`public InputStream getInputStream(ZipEntry entry) throws IOException {`
`entry = getEntry(entry.getName());`	[redacted]
`if (entry == null) {`	`}`
`return null;`	[redacted]
`}`	
`...`	
	`...`

31. By looking closely at ZipFile.getInputStream, I have shown that the same, compatible,

interoperable functionality can differ in many ways — overall, by simply comparing the length of the two implementations; at an intermediate level, by showing that there are different names and numbers of private methods and classes used to implement the functionality; and at a granular level, by showing that one particular subtask is implemented in different ways.

END ORACLE SOURCE CODE - HIGHLY CONFIDENTIAL

32. In each of these three methods examined in this section, I have shown that the programmatic logic used to implement a particular method can be very different, with only one small portion — the method name and argument types — being the same. These files are typical of all Android and Oracle JDK 1.5 files that I have inspected — one small portion, which is required to be the same for purposes of compatibility and interoperability, is the same, and the rest of the file is different. As a result, I disagree with Prof. Mitchell's conclusion that the Android source code is substantially similar to Oracle's copyrighted source code. Instead, it is my opinion that Google's implementation of the APIs at issue is not virtually identical or substantially similar to Oracle's implementation.

V. GOOGLE'S IMPLEMENTATION OF THE APIS AT ISSUE IS NOT VIRTUALLY IDENTICAL OR SUBSTANTIALLY SIMILAR TO ORACLE'S IMPLEMENTATION

33. I understand from the Visnick and Purdy reports that, with the exception of portions of a dozen files, Oracle does not allege that Google has copied Oracle's implementation of the Java APIs. Instead, Oracle only alleges that the classes, interfaces (including fields, constructors, and method signatures), and exceptions are similar in both platforms. In other words, except for 12 files identified by Visnick out of the 9,479 files in Oracle's implementation of Java 1.5, Oracle does not allege that Google copied source code from Oracle. As explained in Section V.Q (paragraph 129) of my Opening Report, the names and parameters of the APIs must be the same for interoperability and efficiency reasons.

While the Android software is compatible with and provided the functionality of the Java language APIs at issue, and necessarily uses the same API names and organization in order to do so, my opinion, after my review of the Android and Oracle source code, is that Android's underlying implementation (or source code) of the APIs is substantially different from Oracle's implementation. Put another way, Android is written in the Java language and compatible with programs that use the Java language APIs at issue, so that developers can reuse their existing code in the Java language on both the Android and Java platforms, but the Android source code was not copied from the source code in Oracle's Java platform. Rather, leaving aside the 12 files identified by Mr. Visnick and addressed in paragraphs 150-177 of my Opening Report, Android includes an independent implementation of the Java language APIs at issue, created without copying the Java platform's source code.

34. Besides the kind of line-by-line analysis done from paragraphs 12-15, we can analyze the differences in the implementations of the APIs by examining the names of the private methods of each implementation. In my opinion, the different names for these private methods show that the Android source code was not copied from the Oracle JDK 1.5 source code.

35. As explained above in paragraph 24, "public methods" are the methods that are made available for use by programmers who use an API to write applications. These must be the same if the two implementations are to be compatible. In contrast, "private methods" help to implement the API but are not visible or available for use by software developers building their own software. The classes that are at issue in this case have public methods that must be implemented in order to be compatible with the API, *e.g.*, Math.abs and Math.sqrt in the java.lang package. However, the API does not dictate how the methods are implemented. I demonstrated in paragraph 24's analysis of getInputStream that private, helper functions are often used in implementing the public methods required by the APIs. Differences in the private methods reflect differences in the implementations. For example, a simple way to see the differences in the

implementations above is to list the names of the private methods, and compare the two. If the names and quantity of the private methods in the two implementations are different, then the implementations themselves are also different. For example, the getInputStream method is implemented using different private methods and private classes in the different implementations — the Android implementation uses three private methods and two private classes, whereas the Oracle JDK 1.5 implementation uses two private classes but no private methods. This difference in the number of the private methods and classes (and in many places, also the type and name of the internal structures) indicates that the two implementations have very different underlying structures and therefore are not similar. This is akin to two very different tables of contents for two books that are on the same topic — differences between the two tables strongly suggests that the underlying content will also be different.

36. Using software I developed to analyze the classes examined in this report, I detected large differences in how public and private methods are used across the Android, GNU Classpath, and Oracle JDK 1.5 implementations. I used the program (attached as Exhibit G) to examine the accused packages, and created the table below to summarize the data for the 740 public classes and interfaces in common between the Android and Java implementations of the 37 accused packages. For comparison, I have also provided information on the GNU Classpath implementation of the same materials.

37. The column labeled "Total Methods" provides the total number of methods (including constructors) found across all classes. The column labeled "Total Private Methods" shows how many of these methods are labeled as private, and hence not accessible to programmers but used to implement the public methods. As I discussed in the example of the getInputStream method in the java.util.zip class, sometimes private methods are used to implement the public methods, but they are not part of a class's API because programmers using the class cannot access the private methods. The column labeled "Percent Private" provides one estimate of how often private methods are used across all classes. Each of these classes contributes a percentage between zero and one hundred to

a running total. If all methods in a class are private, the percent private for that class is 100%. If all methods are public and none are private, the percent private is 0%. The percentage shown in the column is the average of these per-class percentages across all classes. The significant difference between the Android and Oracle implementations in this metric shows that the Android classes use, on average, fewer private methods than the other Java implementations. In my opinion, this indicates that the implementations are significantly structurally different. The structural difference between the implementations is also indicated by the total number of methods that differ across the implementations. Methods can be public, private, or package access, and it is possible to add public methods that are not part of the API. The differences between the total number of methods across the implementations is a further indication that the implementations of the APIs are very different.

	Packages	**Total Methods**	**Total Private Methods**	**Percent Private**
Android	37	8994	970	5.92%
GNU Classpath	37	7365	576	4.11%
JDK 1.5	37	8190	1369	7.17%

38. The substantially different numbers of classes and methods, and the different ratio of public to private methods, strongly suggests that each of the implementations measured is substantially different from the other. In particular, recall from paragraphs 24 and 35 that to achieve compatibility and interoperability, private methods, unlike public methods, are not required to be the same. As a result, the very different number of total private methods in the implementations of the allegedly infringed packages leads me to conclude that, when the authors of the three pieces of software were not constrained by compatibility, they took very different routes to implement the functionality. My direct inspection of a cross-section of the files at issue confirms the results of this numerical approach. As expected from a review of the overall numbers, in the individual classes, the number of private methods and classes, and their underlying implementation, also

vary substantially between the two implementations.

39. As a result of this analysis, it is my opinion that the Android and Oracle JDK implementations are not virtually identical or substantially similar. The only meaningful similarities I have observed are between elements that — as discussed in my Opening Report (section V.J to V.R, paragraphs 90-139) — are necessary for compatibility and interoperability.

VI. THE VARIOUS JAVA VERSIONS THAT ORACLE ALLEGES WERE INFRINGED CONTAIN THE SAME APIS AS EARLIER VERSIONS OR VERSIONS FOR OTHER OPERATING SYSTEMS

40. It is my understanding that Oracle first asserted on July 29, 2011 that Google allegedly infringed its copyright in Java 6. Java 6, like the other allegedly infringed Java versions, contains all the APIs that were contained in previous versions of Java. This is because it is Java's stated policy, for purposes of compatibility, to keep versions of Java as similar as possible to previous versions. When new versions are released, API elements are essentially never changed or removed, only added. This is known as "upwards" compatibility, as referenced in the Java SE Compatibility Policy (*available at* http://java.sun.com/j2se/1.5.0/compatibility.html.) As a result of this policy, the APIs in Java 1.1 are also present, in their entirety, in Java 1.2; all Java 1.1 and any new APIs added in Java 1.2 are present in Java 1.3; all Java 1.2 APIs and any new APIs added in Java 1.3 are present in Java 1.4; and so on.

41. Similarly, it is my understanding that some of the allegedly copied works are Java 1.2 for Windows, Java 1.2 for Linux, Java 1.2 for Mac, Java 1.2 for Solaris, and the same set of platforms for Java 1.3. These works contain deliberately contain the same APIs and API packages. If their APIs were different, it would defeat Java's stated purpose of "write once, run anywhere." The API implementations for each operating system differ, however, so that they will work with the specific operating system. For example, the lastModified method in the java.io.File class asks the underlying operating system when a file was last modified, and returns that time to the program. This method's name,

parameters, and return value (in other words, its API) are the same in Java 1.2 for Windows, Java 1.2 for Mac, as well as Android. The source code that implements the lastModified functionality for Java 1.2 for Windows (the function Java_java_io_Win32FileSystem_getLastModifiedTime contained in the file Win32FileSyste_md.c) is different from the source code for lastModified in Java 1.2 for Solaris (the function Java_java_io_UnixFileSystem_getLastModifiedTime contained in the file UnixFileSystem_md.c). This is necessary, because the different operating systems, and their file systems, tell time differently, and so this source code must "translate" the underlying operating system's time information into the standard Java time system. In fact, because Java's time-keeping system is heavily inspired by Solaris's system, the Unix code for this purpose is roughly 1/3rd the length of the Windows code — less "translation" work is required. Despite these differences in the underlying implementation, as a result of this deliberate goal of making APIs available and compatible across different operating systems, these different works necessarily contain the same groups of APIs.

VII. PARAMETER NAMES ARE FUNCTIONAL AND NOT CREATIVE

42. Prof. Mitchell's report asserts that parameter names are particularly creative, purportedly because they are not reused by programmers. It is correct that the parameter names need not be reused by programmers, who choose their own names when interacting with a method. However, these parameter names still play a functional role because they serve to inform programmers what kind of information the method expects. Like the other components discussed in Section V.L (paragraph 102) of my Opening Report, this functional requirement creates practical restraints on the developer's choice of how to convey information. So, for example, the creators of an API do have the flexibility to call the integer value used by the "abs" function "a," "i," "x," or "Steve." However, if the value is named "Steve," that will still make the documentation and specification of the method unnecessarily confusing to developers who are trying to understand the API.

43. It may be helpful to think about the "creativity" involved in choosing parameter names

(and other named elements in an API) as analogous to the creation of a recipe. In writing down a recipe for cooking a steak, there are a variety of different choices a cook could make in describing a given ingredient. The main ingredient could be called a "steak," the "beef," or even something more unusual like the "cut of cow." That said, practical constraints (such as consumer expectations about ingredient names in recipes) will limit the reasonable choices for the ingredient name. As one extreme example, a cook certainly could choose to call the steak "flubber," and explain to the reader that "flubber" is meant to refer to the cut of meat being cooked, but this would make it difficult for the typical reader to process the instructions in the recipe. Calling the steak "flubber" is thus, as a practical matter, not a reasonable option.

44. A stated in paragraph 112 of my Opening Report, it is my opinion that there is no meaningful *expressive* creativity in short, fragmentary words and phrases. All the parameters in the Java APIs at issue are names and fragmentary phrases, and so they similarly lack expressive creativity. For example, many methods use parameters that are single letters (such as *a*) that reflect the parameter's roots in algebra. Others are simply abbreviations; for example, at least 41 parameters in Oracle's implementation of Java 1.5 are integers called "i" ("i" being a commonly used abbreviation by programmers for integer variables since long before the Java programming language was created) and at least 23 are characters called "c" (again, "c" being a well-known abbreviation of character). Many others are simple names that reflect the underlying idea being manipulated; *e.g.*, the single parameter name for the method JarEntry is named, simply, "name," and the single parameter taken by the method "setSize" is called, appropriately, "size."

VIII. THE ORGANIZATION OF PACKAGES IS FUNCTIONAL AND DOES NOT CONTAIN CREATIVE EXPRESSION

45. As I discussed in section V.N, paragraph 118 of my Opening Report, the organization of packages in Java is not creative expression. Professor Mitchell also addresses this point, but I disagree with his conclusions. For example, in paragraph 180, Prof. Mitchell states

that the streams "ByteArray-," "File-," "Filter-," and "Piped" could have been grouped together and then divided into Input and Output classes without affecting the functionality of the classes. This is incorrect. In fact, the organization of the base classes InputStream and OutputStream, the hierarchy shown in Professor Mitchell's report, and the Reader classes and subclasses he does not mention, are all based on the "Decorator" design pattern from the classic computer science textbook "Design Patterns," by Gamma, Helm, Johnson, and Vlissides. This book is so commonly assigned to undergraduate computer science students that it has a nickname in the computer science profession — the "Gang of Four" book. The "design patterns" described in the textbook are common methods of organizing computer code, and are widely used in the industry as templates — *i.e.*, "patterns" — that sophisticated professional developers should use when organizing their own code. Use of these patterns is not merely a good idea; the patterns help dictate how APIs are designed, because in order for APIs to be accepted and used by developers, it is important to use design rules and guidelines (like the patterns in *Design Patterns*) that the developer community views as accepted and well-understood. Prof. Mitchell's focus on a design that is simply appealing aesthetically is not necessarily a good indication that the design is good from a functional perspective. Instead, reliance on established patterns of organization — like Decorator — is usually a more reliable way of building software.

46. In this case, use of the Decorator design pattern helps to ensure that new types of InputStreams or OutputStreams can be easily added to the hierarchy. Use of the Decorator pattern also facilitates interactions between InputStreams and Reader classes, an important aspect of the java.io package that helps move between streams and files of characters (*e.g.*, the characters of various alphabets) and streams and files of bytes (a lower level kind of data than a character). Although it may be true that a different design could yield the same functionality in terms of reading files or other streams, an API designer must also, for example, ensure that new classes can be added to solve problems that were not anticipated when the API is designed, and the Decorator design pattern used

here is designed to do that. A different design — one using a different design pattern, or not using an established design pattern at all — might make it difficult to add new functionality, or use existing classes together in novel ways. Use of the vetted and established Decorator pattern from the *Design Patterns* text helps to avoid these problems. In this way, the choices in the design of java.io referenced by Professor Mitchell are still highly constrained by the software's functionality. This is not to say that the resulting functionality is not aesthetically pleasing, but Prof. Mitchell, unfortunately, has made the mistake of confusing an aesthetically pleasing outcome with creative expression. In this case, creative expression was not required; like a knife that has been well-sharpened by skillful hands, logical application of consistent, basic design rules created a beautiful outcome without necessarily implying significant creative expression.

IX. C#, LIKE JAVA, IS UNPROTECTABLE, AND IS ALSO AVAILABLE AS AN OPEN SPECIFICATION AND IMPLEMENTATION

47. In paragraph 121, Prof. Mitchell claims that "C# and .Net are *proprietary products* of Microsoft Corporation and Google Android would have had to negotiate terms with Microsoft." (emphasis mine). Prof. Mitchell does not define "proprietary" or otherwise substantiate this claim. It is my opinion that C# and .Net have very similar characteristics to Java, and so Prof. Mitchell's implicit claim that use of C# and .Net would have imposed a different or more significant legal burden than Java because they are purportedly proprietary is incorrect.

48. C# is a programming language, and .Net is the collection of libraries that form C#'s platform, similar to the role the Java Class Libraries play in the Java platform ecosystem. C# and .Net have APIs. Like the Java APIs, the C# and .Net APIs are functional methods of operations that are constrained by a variety of requirements. As explained in my Opening Report, APIs with these characteristics may not be protectable under copyright law, so it is incorrect to refer to C# and .Net as "proprietary" without detailed analysis of the C# and .Net APIs. Certain aspects of C# and .Net may be protectable, but (as with

Java) other aspects may not be, and it would appear premature to characterize C# as "proprietary" or assume that Google could not use it without doing more analysis than Prof. Mitchell appears to have done.

49. More concretely, C# and .Net are also not proprietary (as the word is commonly used) in at least two significant respects. First, significant components of C# and .Net have been made available by Microsoft through the international standards body ECMA as open standards that can be implemented by anyone. (*See, e.g.,* http://www.ecma-international.org/publications/standards/Ecma-334.htm *and* http://www.mono-project.com/ECMA.) The patents associated with these standards have been made available to the public for anyone to implement under Microsoft's "Community Promise" for specifications. (*See* http://www.microsoft.com/openspecifications/en/us/programs/community-promise/covered-specifications/default.aspx.) Second, a third-party version of C# and .Net, called "Mono," is available in part under a permissive license that allows anyone (including Google and Android, should it so desire) to reuse the code. (*See* http://www.mono-project.com/FAQ:_Licensing.) Again, these two facts (Microsoft's publication of a standard, and the existence of a permissively licensed implementation not authored by Microsoft) suggest that Prof. Mitchell's claim that C# and .Net are proprietary is not correct.

X. ORACLE'S ANALYSIS OF THE FILES AT ISSUE DOES NOT DISCUSS THEIR QUALITATIVE OR QUANTITATIVE IMPORTANCE, WITH ONE EXCEPTION THAT IS INCORRECT

50. The Mitchell and Visnick reports discuss the dozen files which I also address in my Opening Report. However, they do not address the qualitative or quantitative importance of these files, glossing over the fact that (as I discussed at length in my Opening Report) these files constitute an incredibly small percentage of the two works at issue — less than 0.13% of Oracle's implementation of Java 1.5 when measured by number of files, less than 0.03% of Oracle's implementation of Java 1.5 when measured by lines of code, and

less than 0.02% of Android by number of files and less than 0.005% of Android by lines of code.

51. Visnick's report states that 12 Android source code files are copied. These are the same 12 files that I discussed in my opening report. I have not confirmed his methodology, but if he is correct, he admits that at most 12 files out of 57,076 files in Android (0.02%) and 9,479 files in Oracle's implementation of Java 1.5 (0.13%) were copied. When the lines of code that Mr. Visnick alleges are similar are compared, the numbers are even smaller — 0.03% of Oracle's implementation and 0.005% of Android. Thus, assuming that his methodology is correct, all Mr. Visnick's report does is confirm that a very small number and percentage of allegedly copied files are at issue, and Mr. Visnick in fact proves my point in paragraph 150 of my Opening Report that these files represent a quantitatively very small portion of the works at issue.

52. Mr. Visnick's report makes no attempt at explaining why these 12 files might be qualitatively important to Java or Android.

53. In comparing the Android APIs to the Java APIs in paragraphs 200-208, outside of the names and organization that is necessary for compatibility and interoperability, Prof. Mitchell never identifies any Android source code that implements these APIs and is identical or even substantially similar to any Oracle source code. Similarly, when discussing use of the method signatures in paragraphs 212-213, he again focuses on one line in each method (the signature) and does not discuss or analyze the source code that implements these methods. As I have shown in paragraphs 13-32 and 34-39, the source code that implements these methods in Android is not substantially similar to any Oracle source code. In fact the method signatures are a tiny percentage of the works at issue; each method signature is typically one line of source code, so the 8190 public methods in the 37 packages at issue constitute less than 0.3% of the 2.8 million lines of code in Java 1.5. Prof. Mitchell glosses over this by saying that there are "hundreds" of files which contain these method signatures, but neither his discussions nor Exhibit Copyright-G actually compare the Oracle implementation to the Google implementation. Actually

doing this comparison, as I have done, shows that the signatures are a very small part of the source code, and that the other components of the source code are not substantially similar.

54. Prof. Mitchell's comparison of the Android source code files to the APIs, without doing an analysis of the Oracle source code, is at odds with public statements made by Sun. In 2006, Tim Bray, who was then Director of Web Technologies at Sun, stated that in Sun's view, an alternative implementation of the Java APIs would only infringe Sun's rights if there was "a direct and substantial copying of code." He also stated that in Sun's view there was "no issue" with GNU Classpath's implementation of the Java APIs. (*See* "Q&A with Tim Bray," *available at* http://www.zdnet.com/blog/burnette/q-a-with-tim-bray/200?pg=3.) As I have shown, GNU Classpath, like Android, is an independent implementation of the Java APIs, with no "direct and substantial copying of code," so if GNU Classpath raises no issues, then Android's use of the Java language API specifications should also raise no issues.

55. Prof. Mitchell's report does state briefly in paragraph 235 that, despite constituting only 0.28% by lines of code of the file Arrays.java, "[n]evertheless, rangeCheck is qualitatively significant to arrays.java, as it is called nine times by other methods in the class." Prof. Mitchell's reliance on frequency of use to assess qualitative significance is misplaced, for several reasons.

56. First, frequency of use is a poor proxy for qualitative significance. For example, in building a car, one designer might choose to use hundreds of 9 mm bolts, while another might choose 3/8 inch bolts. The fact that hundreds of these bolts were used does not mean that the decision to use 9 mm bolts was qualitatively significant to the car's design. Just as the 9 mm bolts perform a mundane function, so too does the rangeCheck method, for the reasons I explained in my Opening Report in paragraphs 153-156.

57. Second, as a general matter, reuse of a function may or may not be indicative of its qualitative importance; it may indicate simply that something is simple and frequently reused, or perhaps that it is used inefficiently. In fact, while rangeCheck is used nine

times in Oracle's Arrays.java, it is used *only once* in Android's TimSort.java, and *only once* in Android's Comparable TimSort.java.

58. Third, in this specific case, the function is reused multiple times in the Oracle code largely because the programming of Arrays.java is inefficient as a result of constraints imposed by the Java language. A comment in the file indicates that:

```
/*
 * The code for each of the seven primitive types is largely
identical.
 * C'est la vie.
 */
```

This repetition of identical code is often a sign that code has been repeated needlessly, and in this case, the "c'est la vie" comment from the original programmer seems to perhaps acknowledge that he regretted the "largely identical" code. The code is identical, and reused seven times, because the Java language does not support a feature called "generic functions for primitive types." If the Arrays.java functionality were implemented in a different language that supported this feature, such as C++ or C#, there would be only one copy of rangeCheck, not seven. Thus the metric of number of calls is not a measure of the importance of rangeCheck, but rather of the inadequacies imposed by the Java language. These seven sets of "largely identical" code explain seven of the nine uses of rangeCheck. The other two uses are similar in that they are also called prior to sorting arrays, but for sorting arrays of Objects rather than primitive types. As a result, it is incorrect to say that the mere numerical use of rangeCheck makes the function qualitatively significant; instead, a more plausible interpretation is that the nine uses of rangeCheck in Arrays.java justify a conclusion that the file was written to cope with inadequacies of the Java language, incorrectly inflating any alleged importance of rangeCheck. (TimSort.java and ComparableTimSort.java do not have to cope with this inadequacy because they do not operate on the so-called primitive types.)

59. Finally, it should be noted that Arrays.java, TimSort.java, and ComparableTimSort.java all provide the functionality of sorting arrays. As noted in my Opening Report, at the time Oracle was first made aware of TimSort.java and ComparableTimSort.java, Oracle's

reaction was not to complain of any alleged "copying," but rather to accept TimSort.java and ComparableTimSort.java *as contributions to Java to be distributed to every single user of Java*, and to praise the author's contribution as significantly increasing the speed and performance of Java. That this one, very brief segment of these two files is similar to code in Arrays.java should strongly suggest (even to someone untrained in programming) that the important part of the TimSort.java and ComparableTimSort.java files are the over 900 lines that are completely different (as opposed to the allegedly similar 9 lines of code), since it is this different part that had such a significant impact on the functionality and efficiency of the software. As a result of these four points, and in agreement with the analysis in my Opening Report, it is my opinion that this method is not qualitatively significant, either to the file Arrays.java or to the infringed work as a whole.

60. I reserve the right to update and refine my opinions and analyses based on any additional materials or information that may come to my attention in the future, including additional contentions by Oracle as well as any rulings issued by the Court in this case. I also reserve the right to supplement my opinions and analyses as set forth in this report in light of any expert reports submitted by Oracle and in light of any deposition or trial testimony of Oracle's experts.

DATED: August 12, 2011

Owen Astrachan, Ph.D.

Exhibit F: Comparison of Android and Oracle ZipFile.getInputStream

BEGIN ORACLE SOURCE CODE - HIGHLY CONFIDENTIAL

Android ZipFile.getInputStream	Oracle JDK 1.5 ZipFile.getInputStream
public InputStream getInputStream(ZipEntry entry) throws IOException {	public InputStream getInputStream(ZipEntry entry) throws IOException {
/*	[redacted]
* Make sure this ZipEntry is in this Zip file. We run it through	}
* the name lookup.	
*/	[redacted]
entry = getEntry(entry.getName());	[redacted]
if (entry == null) {	[redacted]
return null;	[redacted]
}	[redacted]
	[redacted]
/*	[redacted]
* Create a ZipInputStream at the right part of the file.	[redacted]
*/	[redacted]
RandomAccessFile raf = mRaf;	[redacted]
synchronized (raf) {	[redacted]
// We don't know the entry data's start position. All we have is the	[redacted]
// position of the entry's local header. At position 28 we find the	[redacted]
// length of the extra data. In some cases this length differs from	[redacted]
// the one coming in the central header.	[redacted]
RAFStream rafstrm = new RAFStream(raf,	[redacted]
entry.mLocalHeaderRelOffset + 28);	[redacted]
int localExtraLenOrWhatever = ler.readShortLE(rafstrm);	[redacted]
// Skip the name and this "extra" data or whatever it is:	[redacted]
rafstrm.skip(entry.nameLen + localExtraLenOrWhatever);	[redacted]
rafstrm.mLength = rafstrm.mOffset + entry.compressedSize;	[redacted]
if (entry.compressionMethod == ZipEntry.DEFLATED) {	[redacted]
int bufSize = Math.max(1024, (int)Math.min(entry.getSize(), 65535L));	[redacted]
return new ZipInflaterInputStream(rafstrm, new Inflater(true), bufSize, entry);	[redacted]
} else {	[redacted]
return rafstrm;	[redacted]
}	
}	[redacted]
}	[redacted]
//--	[redacted]
static class RAFStream extends InputStream {	[redacted]
	[redacted]
RandomAccessFile mSharedRaf;	[redacted]
long mOffset;	[redacted]

#	Android ZipFile.getInputStream	Oracle JDK 1.5 ZipFile.getInputStream
39	long mLength;	[redacted]
40		[redacted]
41	public RAFStream(RandomAccessFile raf, long pos) throws IOException {	[redacted]
42	mSharedRaf = raf;	[redacted]
43	mOffset = pos;	[redacted]
44	mLength = raf.length();	[redacted]
45	}	[redacted]
46		[redacted]
47	@Override	[redacted]
48	public int available() throws IOException {	[redacted]
49	return (mOffset < mLength ? 1 : 0);	[redacted]
50	}	[redacted]
51		[redacted]
52	@Override	[redacted]
53	public int read() throws IOException {	[redacted]
54	byte[] singleByteBuf = new byte[1];	[redacted]
55	if (read(singleByteBuf, 0, 1) == 1) {	[redacted]
56	return singleByteBuf[0] & 0XFF;	
57	} else {	[redacted]
58	return -1;	[redacted]
59	}	[redacted]
60	}	[redacted]
61		[redacted]
62	@Override	[redacted]
63	public int read(byte[] b, int off, int len) throws IOException {	[redacted]
64	synchronized (mSharedRaf) {	[redacted]
65	mSharedRaf.seek(mOffset);	[redacted]
66	if (len > mLength - mOffset) {	[redacted]
67	len = (int) (mLength - mOffset);	[redacted]
68	}	[redacted]
69	int count = mSharedRaf.read(b, off, len);	
70	if (count > 0) {	[redacted]
71	mOffset += count;	[redacted]
72	return count;	[redacted]
73	} else {	[redacted]
74	return -1;	[redacted]
75	}	
76	}	[redacted]
77	}	[redacted]
78		[redacted]
79	@Override	[redacted]
80	public long skip(long n) throws IOException {	[redacted]
81	if (n > mLength - mOffset) {	

#	Android ZipFile.getInputStream	Oracle JDK 1.5 ZipFile.getInputStream
82	`n = mLength - mOffset;`	
83	`}`	[redacted]
84	`mOffset += n;`	[redacted]
85	`return n;`	[redacted]
86	`}`	[redacted]
87	`}`	[redacted]
88	`//--`	[redacted]
89		[redacted]
90	`static class ZipInflaterInputStream extends InflaterInputStream {`	[redacted]
91		[redacted]
92	`ZipEntry entry;`	[redacted]
93	`long bytesRead = 0;`	[redacted]
94		[redacted]
95	`public ZipInflaterInputStream(InputStream is, Inflater inf, int bsize, ZipEntry entry) {`	
96	`super(is, inf, bsize);`	[redacted]
97	`this.entry = entry;`	
98	`}`	[redacted]
99		[redacted]
100	`@Override`	
101	`public int read(byte[] buffer, int off, int nbytes) throws IOException {`	[redacted]
102	`int i = super.read(buffer, off, nbytes);`	[redacted]
103	`if (i != -1) {`	
104	`bytesRead += i;`	[redacted]
105	`}`	[redacted]
106	`return i;`	[redacted]
107	`}`	[redacted]
108		[redacted]
109	`@Override`	[redacted]
110	`public int available() throws IOException {`	[redacted]
111	`if (closed) {`	[redacted]
112	`// Our superclass will throw an exception, but there's a jtreg test that`	
113	`// explicitly checks that the InputStream returned from ZipFile.getInputStream`	[redacted]
114	`// returns 0 even when closed.`	
115	`return 0;`	[redacted]
116	`}`	[redacted]
117	`return super.available() == 0 ? 0 : (int) (entry.getSize() - bytesRead);`	[redacted]
118	`}`	[redacted]
119	`}`	
120	`}`	[redacted]
121		[redacted]
122		[redacted]

#	Android ZipFile.getInputStream	Oracle JDK 1.5 ZipFile.getInputStream
123		[redacted]
124		[redacted]
125		[redacted]
126		[redacted]
127		[redacted]
128		[redacted]
129		[redacted]
130		[redacted]
131		
132		[redacted]
133		[redacted]
134		[redacted]
135		[redacted]
136		[redacted]
137		[redacted]
138		[redacted]
139		[redacted]
140		[redacted]
141		[redacted]
142		[redacted]
143		[redacted]
144		[redacted]
145		
146		[redacted]
147		[redacted]
148		
149		[redacted]
150		[redacted]
151		[redacted]
152		[redacted]
153		[redacted]
154		
155		[redacted]
156		[redacted]
157		[redacted]
158		
159		[redacted]
160		[redacted]
161		[redacted]
162		[redacted]
163		[redacted]
164		[redacted]
165		[redacted]
166		
167		[redacted]
168		[redacted]
169		[redacted]
170		[redacted]
171		[redacted]
172		[redacted]

#	Android ZipFile.getInputStream	Oracle JDK 1.5 ZipFile.getInputStream
173		[redacted]
174		[redacted]
175		[redacted]
176		
177		[redacted]
178		[redacted]
179		[redacted]
180		
181		[redacted]
182		[redacted]
183		[redacted]
184		
185		[redacted]
186		[redacted]
187		[redacted]
188		[redacted]
189		[redacted]
190		[redacted]
191		[redacted]
192		[redacted]
193		[redacted]
194		
195		[redacted]
196		[redacted]
197		[redacted]
198		[redacted]
199		[redacted]
200		[redacted]
201		[redacted]
202		
203		[redacted]
204		[redacted]
205		[redacted]
206		[redacted]
207		[redacted]
208		[redacted]
209		
210		[redacted]
211		[redacted]
212		[redacted]
213		[redacted]
214		[redacted]
215		[redacted]
216		[redacted]
217		[redacted]
218		[redacted]
219		[redacted]
220		[redacted]

#	Android ZipFile.getInputStream	Oracle JDK 1.5 ZipFile.getInputStream
221		[redacted]
222		
223		[redacted]
224		[redacted]
225		[redacted]
226		[redacted]
227		[redacted]
228		[redacted]
229		[redacted]
230		[redacted]
231		[redacted]
232		[redacted]
233		[redacted]
234		[redacted]
235		
236		[redacted]
237		[redacted]
238		[redacted]
239		[redacted]
240		[redacted]
241		[redacted]
242		[redacted]
243		[redacted]
244		
245		[redacted]
246		[redacted]
247		[redacted]
248		[redacted]
249		[redacted]
250		[redacted]
251		[redacted]
252		[redacted]
253		[redacted]
254		[redacted]
255		
256		[redacted]
257		[redacted]
258		[redacted]
259		
260		[redacted]
261		[redacted]
262		[redacted]
263		
264		[redacted]
265		[redacted]
266		[redacted]
267		[redacted]
268		[redacted]
269		[redacted]
270		[redacted]
271		[redacted]

#	Android ZipFile.getInputStream	Oracle JDK 1.5 ZipFile.getInputStream
272		[redacted]
273		[redacted]

END ORACLE SOURCE CODE - HIGHLY CONFIDENTIAL

Exhibit G: PublicPrivateAnalyzer.py Source Code

Printed by Owen L. Astrachan


```python
'''
Created as part of work on expert report
for Google/Oracle for GreenbergTraurig

@author: ola
@copyright: owen astrachan, compsciconsulting
'''
import os,collections,re

acdict = collections.defaultdict(int)
aperclass = collections.defaultdict(int)
aprivdict = {}
aset = set()
amethnames = []
apubclass = set()

jcdict = collections.defaultdict(int)
jperclass = collections.defaultdict(int)
jprivdict = {}
jset = set()
jmethnames = []
jpubclass = set()

gcdict = collections.defaultdict(int)
gperclass = collections.defaultdict(int)
gprivdict = {}
gset = set()
gmethnames = []
gpubclass = set()

afunclist = []
jfunclist = []
gfunclist = []

methnames = []

public_ids = ["public class",
              "public abstract class",
              "public interface",
              "protected class",
              "protected",
              "public"]

def is_func(line):
    if "new " in line:
        return False
    parts = line.split()
    if line.startswith("public") and line.find(")") >= 0 and line.find("(") >= 0:
        return True
    if line.startswith("private") and line.find(")") >= 0 and line.find("(") >= 0:
        return True
    return False


def getClass(path):
    '''
  path ends with .java, return class name preceding .java including preceding .
  e.g., for java/lang/Arrays, return .Arrays
  '''
    nm = path[:-5]
    index = nm.rfind("/")
    return "."+nm[index+1:]

def pubtrack(fname,pubclass,clname):
    f = open(fname)
    allText = f.read()
    changedText = re.sub(r"\s+"," ",allText)
    contents = changedText.split()
    for i in range(len(contents)-2):
        if contents[i] == "public" and contents[i+1] == "class":
            pubclass.add(clname)
            break
        if contents[i] == "public" and contents[i+1] == "interface":
```



```python
            pubclass.add(clname)
            break
        if contents[i] == "public" and contents[i+1] == "abstract" and contents[i+2]
 == "class":
            pubclass.add(clname)
            break
        if contents[i] == "public" and contents[i+1] == "final" and contents[i+2] =
= "class":
            pubclass.add(clname)
            break

def do_one(packname,onepath,cdict,perclass,cset,funclist,privdict,methnames,pubc
lass):

    if not onepath.endswith(".java"):
        return True
    if onepath.endswith("package-info.java"):
        return True


    class_name = getClass(onepath)
    #pubtrack(onepath,pubclass,packname+class_name)
    fullname = packname+class_name
    if not fullname in pubclass:
        print "rejected",fullname
        return True

    f = open(onepath)

    pcount = 0
    first = True
    public = False
    pubf = 0
    privf = 0
    for line in f:

        line = line.strip()

        if is_func(line):
            methnames.append(line)
            if line.startswith("public"):
                pubf += 1
            else:
                privf += 1
                nm = packname+class_name
                if not nm in privdict:
                    privdict[nm] = []
                privdict[nm].append(line)

        if first and line.startswith("class "):
            #print "class",onepath,line
            base = os.path.basename(onepath)
            cset.add(base)

        pfound = False
        for pub in public_ids:

            if line.startswith(pub):
                if first:
                    first = False
                    if line.find("public") >= 0 or line.find("protected") >= 0:
                        public = True
                    else:
                        print "big problem",onepath,pub,line
                if line.find("protected") < 0:
                    pcount += 1

                cdict[pub] += 1
                pfound = True
                if line.find("class") >= 0 and line.find("extends") >= 0:
                    cdict["extends"] += 1
                elif line.find("interface") >= 0 and line.find("extends") >= 0:
                    cdict["extends"] += 1
```

```
                break

    f.close()

    perclass[pcount] += 1
    if pcount == 0:
        #print "%s = %d" % (onepath,pcount)
        pass

    funclist.append((pubf,privf))
    return public


def pop_one(packname,onepath,pubclass):

    if not onepath.endswith(".java"):
        return True
    if onepath.endswith("package-info.java"):
        return True

    class_name = getClass(onepath)
    pubtrack(onepath,pubclass,packname+class_name)

def populate(basepath,packname,pubclass):
    parts = packname.split(".")
    pathize = '/'.join(parts)
    packagepath = os.path.join(basepath,pathize)
    for top in os.listdir(packagepath):
        top_path = os.path.join(packagepath,top)
        if os.path.isdir(top_path):
            #print "*** %s is a directory in %s" % (top,packagepath)
            pass
        else:
            c = pop_one(packname,top_path,pubclass)


def topcount(basepath,packname,cdict,perclass,cset,funclist,privdict,methnames,p
ubclass):
    parts = packname.split(".")
    pathize = '/'.join(parts)
    packagepath = os.path.join(basepath,pathize)
    for top in os.listdir(packagepath):
        top_path = os.path.join(packagepath,top)
        if os.path.isdir(top_path):
            #print "*** %s is a directory in %s" % (top,packagepath)
            pass
        else:
            c = do_one(packname,top_path,cdict,perclass,cset,funclist,privdict,m
ethnames,pubclass)
            if not c:
                #print "no public",top_path,top
                pass
            #print "%s has %d public" % (top_path,c)

def func_stats(coll):
    low = 0
    word_total = 0
    wt_count = 0
    nonlow = 0
    getter = 0
    setter = 0
    req = 0

    obj_names = ["toString", "hashCode", "notifyAll", "getClass"]

    for nm in coll:
        if nm.islower():
            low += 1
            #print "\t lower",nm
        else:
            wc = 0
            for i,ch in enumerate(nm):
                if ch.isupper() and i > 0 and nm[i-1].islower():
```



```
                    wc += 1

            wc += 1
            #word_total += wc
            nonlow += 1

            if nm.startswith("get"):
                getter += 1
            elif nm.startswith("set"):
                setter += 1
            elif nm in obj_names:
                req += 1
            else:
                word_total += wc
                wt_count += 1

    print "total = %d, one = %d more = %d\n" % (nonlow+low,low,nonlow)
    print "perc = %f avg = %f\n" % (1.0*low/(low+nonlow),1.0*word_total/wt_count)
    print "non simple = %d\n" % (wt_count)

    print "getter = %d, setter = %d, req = %d, total = %d\n" % (getter,setter,req,req+getter+se
tter)

def funcalyze(methnames):
    all_names = set()
    names = []
    for meth in methnames:
        if meth.startswith("public"):
            nameEnd = meth.find("(")
            if nameEnd == -1:
                print "error on ",meth
            else:
                name = meth[:nameEnd]
                space = name.rfind(" ")
                mname = name[space+1:]
                all_names.add(mname)
                names.append(mname)

    print "total = %d, unique = %d\n" % (len(names), len(all_names))
    print "unique"
    func_stats(all_names)
    print "total"
    func_stats(names)

    meth_counts = [(names.count(nm),nm) for nm in all_names]
    smc = sorted(meth_counts, reverse=True)
    print "top func occurrences"
    for pair in smc[:20]:
        print pair


    return all_names



def report(cdict,perclass,funclist,privdict,methnames):

    uset = funcalyze(methnames)


    ctotal = 0
    for key in cdict:
        if key.find("public") < 0:
            continue
        print "%s occurrences = %d" % (key,cdict[key])
        if key.find("class") >= 0 or key.find("interface") >= 0:
            ctotal += cdict[key]
    print "----"
    print "public class/interface total = %d" % (ctotal)

    ctotal = 0
    for key in cdict:
        if key.find("protected") < 0:
```

```
            continue
        print "%s occurrences = %d" % (key,cdict[key])
        if key.find("class") >= 0 or key.find("interface") >= 0:
            ctotal += cdict[key]
    print "----"
    print "protected class/interface total = %d" % (ctotal)

    print "per class method counts"
    print "# methods\t#classes"
    total = 0
    levels = collections.defaultdict(int)
    levlist = [0,1,6,11,16,21,51,101,100001]
    for method_count in sorted(perclass.keys()):
        print "%d\t%d" % (method_count,perclass[method_count])
        total += method_count*perclass[method_count]
        for lev in xrange(1,len(levlist)):
            if levlist[lev-1] <= method_count < levlist[lev]:
                levels[lev] += perclass[method_count]
    print "-----"
    print "total methods = %d" % (total)
    print "\n---summary--"
    total = 0
    for lev in xrange(1,len(levlist)):
        print "perclass from %d to %d = %d" % (levlist[lev-1],levlist[lev]-1,levels[le
v])
        total += levels[lev]
    print "total = %d" % (total)

    print "size of funclist = %d" % (len(funclist))
    total = 0
    totalMeths = 0
    totalPriv = 0
    for x in funclist:
        totalMeths += x[0] + x[1]
        totalPriv += x[1]
        if x[0] != 0 or x[1] != 0:
            total += 100.0*x[0]/(x[1]+x[0])
    print "average = %f" % (total/len(funclist))
    print "total meths = %d" % (totalMeths)
    print "total private = %d" % (totalPriv)
    return uset

def analyze():

    apath = "/Users/ola/expert/google/SOURCE/libcore/luni/src/main/java"
    javapath = "/Users/ola/expert/google/ESOURCE/j2se/src/share/classes"
    gnupath = "/Users/ola/expert/google/source-gnu/classpath-0.98"

    packages = ["java.awt.font",
                "java.beans",
                "java.io",
                "java.lang",
                "java.lang.annotation",
                "java.lang.ref",
                "java.lang.reflect",
                #"java.math",
                "java.net",
                "java.nio",
                "java.nio.channels",
                "java.nio.channels.spi",
                "java.nio.charset",
                "java.nio.charset.spi",
                "java.security",
                "java.security.acl",
                "java.security.cert",
                "java.security.interfaces",
                "java.security.spec",
                "java.sql",
                "java.text",
                "java.util",
                #"java.util.concurrent",
                #"java.util.concurrrent.atomic",
                #"java.util.concurrent.locks",
```



```
                "java.util.jar",
                "java.util.logging",
                "java.util.prefs",
                "java.util.regex",
                "java.util.zip",
                "javax.crypto",
                "javax.crypto.interfaces",
                "javax.crypto.spec",
                "javax.net",
                "javax.net.ssl",
                "javax.security.auth",
                "javax.security.auth.callback",
                "javax.security.auth.login",
                "javax.security.auth.x500",
                "javax.security.cert",
                "javax.sql",
                #"javax.xml",
                #"javax.xml.datatype",
                #"javax.xml.namespace",
                #"javax.xml.parsers",
                #"javax.xml.transform",
                #"javax.xml.transform.dom",
                #"javax.xml.transform.sax",
                #"javax.xml.transform.stream",
                #"javax.xml.validation",
                #"javax.xml.xpath"
                ]

    for pack in packages:
        populate(javapath,pack,jpubclass)
        populate(apath,pack,apubclass)
        populate(gnupath,pack,gpubclass)

    allinter = jpubclass & apubclass
    print "js = %d, as = %d, gs = %d, inter = %d\n" % (len(jpubclass),len(apubclass),len(gpu
bclass),len(allinter))

    #return

    for pack in packages:
        print "java"
        topcount(javapath,pack,jcdict,jperclass,jset,jfunclist,jprivdict,jmethna
mes,jpubclass)
        print "android"
        topcount(apath,pack,acdict,aperclass,aset,afunclist,aprivdict,amethnames
,apubclass)
        print "gnu"
        topcount(gnupath,pack,gcdict,gperclass,gset,gfunclist,gprivdict,gmethnam
es,gpubclass)

    print "%d packages analyzed" % (len(packages))
    print "\nJava Analysis"
    juset = report(jcdict,jperclass,jfunclist,jprivdict,jmethnames)
    print "\nAndroid Analysis"
    auset = report(acdict,aperclass,afunclist,aprivdict,amethnames)
    print "\nGnuClasspath Analysis"
    report(gcdict,gperclass,gfunclist,gprivdict,gmethnames)
    print "\n-----"

    jmset = juset
    amset = auset
    inter = jmset&amset
    aonly = amset-jmset
    jonly = jmset-amset
    print "android only count = ",len(aonly),len(amset)
    print "java only count = ",len(jonly),len(jmset)
    print "android only"
    for i,n in enumerate(sorted(aonly)):
        print i,n
    print "java only"
    for i,n in enumerate(sorted(jonly)):
        print i,n
```

```
    # public classes that are different?
#    japub = jpubclass - apubclass
#    ajpub = apubclass - jpubclass
#    print "javapub = %d, android pub = %d, j-a = %d, a-j = %d\n" % (len(jpubcla
ss),len(apubclass), len(japub),len(ajpub))
#    print "java public not in android"
#    for nm in sorted(japub):
#        print nm
#    print "-----\n"
#    print "android public not in java"
#    for nm in sorted(ajpub):
#        print nm
#    print "-----\n"


    privlog = open("privatelog","w")
    for pack in aprivdict:
        if pack in jprivdict:
            line = "package class private {0!s}\n".format(pack)
            print "package class private %s" % (pack)
            privlog.write(line)
            for priv in aprivdict[pack]:
                line = "\tAndroid {0!s}\n".format(priv)
                privlog.write(line)
                #print "\tAndroid %s" % (priv)
                if priv in jprivdict[pack]:
                    privlog.write("\t\talso in Java\n")
                    #print "\t\talso in Java"
            for priv in jprivdict[pack]:
                if not priv in aprivdict[pack]:
                    privlog.write("\tJava "+priv+"\n")
                    #print "\tJava %s" % (priv)
    privlog.close()




#    print "common package/private"
#    inter = jset&aset
#    for name in inter:
#        print name
#
#    print "\nAndroid\n------"
#    for name in aset:
#        print name
#    print "\nJava\n------"
#    for name in jset:
#        print name




if __name__ == "__main__":
    analyze()
```