EXHIBIT 2

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

SAN FRANCISCO DIVISION

ORACLE AMERICA, INC. Plaintiff, v. GOOGLE INC. Defendant.	Case No. CV 10-03561 WHA

DR. JOHN C. MITCHELL'S REPORT IN OPPOSITION TO DR. OWEN ASTRACHAN'S OPENING EXPERT REPORT

SUBMITTED ON BEHALF OF PLAINTIFF ORACLE AMERICA, INC.

CONFIDENTIAL PURSUANT TO PROTECTIVE ORDER
Highly Confidential Attorneys' Eyes Only

TABLE OF CONTENTS

Page

1. I, John C. Mitchell, Ph.D., submit the following expert report ("Opposition Copyright Report") on behalf of plaintiff Oracle America, Inc. ("Oracle"). I submit this report in response to the Opening Expert Report of Dr. Owen Astrachan, dated July 29, 2011 ("Astrachan Report"). My opening copyright report, filed that same day, already addresses many of the issues that have been raised by Dr. Astrachan. Rather than repeat the opinions and information contained in my opening report, I incorporate that report here by reference.

I. THE JAVA API SPECIFICATIONS CONTAIN MANY ORIGINAL, COPYRIGHTABLE ELEMENTS

A. Overview

2. A central theme of Dr. Astrachan's report is that the elements of an API, and the organization of those elements, are dictated by function and do not reflect creativity. (*See, e.g.*, Astrachan Report ¶ 118.) I strongly disagree with this view. As described in detail in my opening report, and additionally in this report below, it is my opinion that multiple elements of the Java API specifications contain copyrightable expression that reflect creativity and are not dictated by function. These include the selection, coordination and arrangement of the Java packages; the class names and definitions, fields, methods and method signatures contained in each package; and the prose text that explains each of these elements.

3. Dr. Astrachan's opinion that the elements of the Java APIs do not reflect creativity is at odds with the views that have been expressed by Google employees who played key roles in the API design for Java and Android, and in industry publications, and the position that Google itself has taken with respect to the protection of the intellectual property rights contained within its own APIs.

4. As discussed in my opening report, Joshua Bloch, a former Sun engineer who now works for Google, and who authored many of the APIs at issue, wrote in 2005 that an API can be "among a company's greatest assets." (OAGOOGLE0100219511 at 512.) Bloch expressed his view in notes from an invited talk at a major software conference that "API design is an art, not a science." (http://www.infoq.com/articles/API-Design-Joshua-Bloch.)

5. After my opening report was filed, Oracle took the deposition of Robert ("Bob") Lee, a former Google engineer who was the "core library lead for Android." (*See* 8/3/2011 Lee Dep. 5:23-6:18.) Mr. Lee was responsible for, as he described it, "re-implementing" the Java APIs for Android, as well as designing some of the additional Android APIs and APIs for other Google applications. Mr. Lee testified that designing APIs is "absolutely" a creative activity, and wrote in a self-evaluation at the time that he would "much rather be designing APIs than re-implementing them." (*See* 8/3/2011 Lee Dep. 13:9-14:11; GOOGLE-40-00034698.) What Mr. Lee refers to as the much less interesting work of "re-implementing" APIs is writing source code to implement the core Java libraries for Android based on Oracle's pre-existing API specifications. Yet re-implementation is the portion of the API development process to which Dr. Astrachan's report assigns all of the creative credit. (*See, e.g.*, Astrachan Report ¶¶ 53-54.)

6. There are many creative decisions that go into crafting a good API, and APIs involve significant design choices. For this reason, there are many articles, and even entire books, dedicated to the techniques and considerations involved in writing APIs. These publications echo the sentiments expressed by Mr. Bloch and Mr. Lee that APIs are important and that writing APIs involves a great deal of creativity and skill. For example, in a 2009 article that appeared in "Communications of the ACM," the most prominent journal of the Association for Computing Machinery, the primary professional

organization in the field of computing worldwide, Michi Henning writes that, "for every way to design an API correctly, there are usually dozens of ways to design it incorrectly. Simply put, it is very easy to create a bad API and rather difficult to create a good one." (M. Henning, "API Design Matters," Communications of the ACM, Vol 52 No. 5 at 46-56, *available at* http://cacm.acm.org/magazines/2009/5/24646-api-design-matters/fulltext.) The article notes that, "There seems to be something elusive about API design that, despite years of progress, we have yet to master," and continues:

> *Good APIs are hard*. We all recognize a good API when we get to use one. Good APIs are a joy to use. They work without friction and almost disappear from sight: the right call for a particular job is available at just the right time, can be found and memorized easily, is well documented, has an interface that is intuitive to use, and deals correctly with boundary conditions.

(*Id.* (emphasis in original).)

7. The book *API Design for C++* is, as the name implies, a book focused on describing techniques for designing APIs for the programming language C++. The author describes the importance of interface design: "A well-designed API can be your organization's biggest asset. Conversely, a poor API can create a support nightmare and even turn your users toward your competitor . . ." (M. Reddy, *API Design for C++* 4 (2011).)

8. As the author of another book on API design explains:

> It's hard to write good APIs that can be consumed by a wide audience of users, especially an international one. Everyone has their own style of understanding and their own way of viewing problems. Satisfying all of them is hard. Satisfying all of them at once is usually impossible. Moreover, if your API is targeted to an international audience, it needs to deal with various cultural differences.

> That's why writing good, widely approachable APIs is hard.

(J. Tulach, *Practical API Design: Confessions of a Java Framework Architect* 3 (2008).)

9. I refer to the publications above merely for the purpose of showing that there is a great deal of energy and thinking that goes into designing APIs for a complex, modern software development platform like Java, and writing them properly and elegantly. In my opinion it is simply incorrect to suggest that writing these types of APIs is any less creative or more constrained by functionality than writing other parts of software.

10. Dr. Astrachan's position that APIs are not copyrightable because they do not involve creativity and are constrained by function is also contradicted by Google's own attempts to protect the copyrighted expression in its own API specifications. Google licenses its APIs to developers and end-users, as Oracle does, but makes clear to them that all rights in the APIs are reserved to Google. For example, it is apparently Google's position that it owns the copyright to its AdSense APIs. The AdSense API Terms and Conditions (http://code.google.com/apis/adsense/terms.html) state:

> "AdSense API Specifications" means all information and documentation Google provides specifying or concerning AdSense API specifications and protocols and any Google-supplied implementations or methods of use of AdSense API. . . .
>
> As between You and Google, Google and its applicable licensors retain all intellectual property rights (including all patent, trademark, copyright, and other proprietary rights) in and to AdSense API Specifications, all Google websites and all Google services and any derivative works in connection therewith. All license rights granted herein are not sublicenseable, transferable or assignable unless otherwise stated herein.

11. It is apparently Google's position that it owns the copyright to the YouTube APIs. The YouTube API Terms of Service (http://code.google.com/apis/youtube/terms.html) read:

> As between You and YouTube, YouTube, its corporate affiliates, and its applicable licensors retain all intellectual property rights (including all patent, trademark, copyright, trade secret, and other proprietary rights) in and to the YouTube API and its documentation and specifications, all YouTube websites and all YouTube services and any derivative works in connection therewith.

12. Google also appears to take the position that it owns all right, title and interest in and to the Google SOAP Search API, including copyright rights. The "Terms and Conditions for Google SOAP Search API Service" (http://code.google.com/apis/soapsearch/api_terms.html) state:

> You agree not to remove, obscure, or alter Google's copyright notice, trademarks, or other proprietary rights notices affixed to or contained within Google SOAP Search API. You also acknowledge that Google owns all right, title and interest in and to Google SOAP Search API, including without limitation all intellectual property rights (the "Google Rights"). The Google Rights include rights to the following: (1) the APIs developed and provided by Google, (2) all software associated with the Google SOAP Search API server, and (3) the search results and spell checking you obtain when you use Google SOAP Search API. The Google Rights do not include the following: (1) third-party components used as part of Google SOAP Search API; or (2) software developed by you in conjunction with using Google SOAP Search API.

13. I note that, according to the terms and conditions, licensed users of the Google SOAP Search API are not permitted to remove, obscure, or alter Google's copyright notices on its Google SOAP Search API. The API specification

(http://code.google.com/apis/soapsearch/reference.html) bears the legend "©2006 Google."

14. I have not analyzed these Google APIs in any detail, but that is not necessary to see that Google states that it has proprietary rights in these APIs, including copyrights.

15. I now turn to discussing some of the specific elements of APIs that are (or are not) addressed by Dr. Astrachan in his opening report.

B. The Selection, Coordination and Arrangement of Packages and Classes in the Java APIs Require Creativity And Contain Original Expression.

16. A great deal of creativity was involved in selecting what to include in the Java APIs, and establishing the arrangement, hierarchy and interdependencies of the packages and classes.

17. To better understand this, it would be useful to start with a discussion of the definition of the term "interface." The word interface is used to describe many things in the computer world. At its simplest, the term interface is used to describe the parts of a component that are visible to other components, and it applies in the context of both software and hardware. It can also apply to the point of interaction between a human and a computer. As may be apparent from this definition, there are a huge variety of types of interfaces, of varying degrees of complexity. It follows that different types of interfaces may contain vastly different degrees of original expression and creativity.

18. The Java APIs are an example of a very complex form of interface. Dr. Astrachan states that APIs "are similar to the interfaces that a computer user uses to operate software, like a keyboard command or button." (Astrachan Report ¶ 26.) This is a vast oversimplification and is not a meaningful comparison. Nor is the comparison to the interface of a car which, at least at the very high level of the components described by

Dr. Astrachan, is at this point in the history of the automobile relatively well defined. Neither analogy captures the depth or complexity of API design or the interdependencies among the thousands of elements involved. These APIs are designed for programmers and are designed to provide and describe a very rich development environment. As such, this type of interface is much more detailed and complex than a keyboard command. It is also much more detailed and complex than a pull down menu that organizes commands for a user. Some of today's software systems are among the most complex artifacts ever created by man, and the use of APIs is the core structuring concept that software designers use to manage their complexity.

19. As described in my opening report, the Java API specifications relate to a set of extensive Java Class Libraries. The API specifications describe each class and its specified behaviors. As described further below, the Java programming language also has its own internal construct called Interface, which is essentially a form of class that relies on other classes to implement its objects. (In hopes of avoiding confusion, I will use the capitalized term Interface to refer to the Java language construct with this name.) The APIs include the name of each class and Interface in the library, and define the relationships between classes, Interfaces, and packages of classes. One important relationship in Java is the subtype relation: if one class or Interface is a subtype of another, then objects of the first type are treated as a special case of the second type. The subtype hierarchy therefore is a form of taxonomic hierarchy that organizes a complex set of possible variations into simplifying categories that articulate common features across the library. For every class or Interface in the library, the API also describes the fields and methods that may be available internally as well as those that are exposed to other packages or classes, according to several visibility choices provided by the Java

language. There are literally thousands of fields, methods, classes and packages that make up the APIs.

20. Dr. Astrachan's report devotes relatively little time to discussing the selection of packages and classes, along with their arrangement and hierarchy, largely viewing an API from the perspective of an application programmer using a completed API rather than considering the art of creating new APIs. The decision regarding the classes and packages to include is not trivial, however, and is not one that is simply dictated by function. It is important to realize that, while the robust and elegant API specifications and class libraries that implement them have been important to Java's success, it was not necessary to include any particular class or package (beyond perhaps a very few classes like Object and Class that are tied closely to the Java language) for the Java language to function. The architects of the Java APIs chose to include specific packages and classes as part of the Java platform to provide developers with tools that allow them to program more quickly and efficiently by calling upon certain sections of code that are already written, rather than having to write them from scratch. But this was not a functional requirement.

21. In this way, the Java APIs differ fundamentally from, for example, an interface that is required in order for a video game to play on a particular game machine platform. In the latter case, if the proper interface is not in place, the video game will not be able to play on the platform. But a software developer can program in Java without ever using any particular library classes that are implementable in the Java language. There is room for a great deal of creativity in the selection of what to include in the APIs and it is incorrect to say that this design process is constrained by functional requirements or compatibility. The decision of what to include is not a straightforward proposition at all, and sometimes the decision of what not to include can be as important as the decision

of what to include. Include too little and the developers will not have the flexibility and tools they desire. Include too much and the APIs will become overwhelming and difficult to use.

22. As just one example, a situation that an API designer will face repeatedly is how many parameters define for a particular method and the order in which to list them. It might be tempting to provide APIs that offer as many parameter options as possible to provide maximum flexibility. But often the right solution is to limit the options, not just to narrow the range of choices for the sake of comprehensibility, but because doing so will make the system more stable. In Java language terms, it is better to establish a set order among parameters, for example, and always use that order, rather than provide overloaded methods that give the programmer many options. Designs that offer many ways to call a function and designs that offer fewer ways offer the same underlying function to application developers, but one is considered the better and more elegant API design.

23. An important purpose of the Java APIs that Dr. Astrachan's report does not adequately acknowledge is that the APIs are geared towards humans, not just towards the computer. The APIs impose a level of abstraction and structure on top of the underlying software development platform. The computer doesn't fundamentally care how the API is organized at all. A list of names corresponding to different commands could be stored in any random order; the Java compiler will match function calls to function definitions by name, no matter how confusing or complex the organization. But humans require some form of order in order to solve complex programming problems and work efficiently. The expression of the API in a particular hierarchical structure, with rules that are consistent, logical and easy to follow, is an important aspect of the API specification that is not functionally mandated by the computer. I disagree with Dr.

Astrachan's characterization of the APIs as simply "a means by which a person operates something," which leaves out this fundamental purpose of the APIs altogether.

24. One critical design choice is which classes of objects will inherit characteristics from others. As I explained in my previous report, classes of software objects in Java are arranged hierarchically, so that a subclass may inherit characteristics from its parent. The API designer must decide this hierarchy and also the interdependency among the numerous classes ─ in other words, should a class be independent, or should it be used to define other classes, and if so, which ones.

25. These decisions have many important practical implications for programming. To give just one, software today is frequently implemented by large development teams, often spread out in different cities and even in different countries. A programming team, following the API, will know that if the methods in a particular class are independent, the developers can work on that code relatively independently, without having to worry about the potential impact on other classes. If the subclass inherits characteristics from a level higher up, or is exposed to, or called upon by, another class in another package, the team has to make sure to coordinate closely.

26. Another example of the way that classes can interact is through a reference type in Java called an "Interface." This reference type is not to be confused with the more general term "interface" that is used in the phrase Application Program Interface (API) or described at the beginning of this report Again, I will distinguish between them by using a capital "I" for the Java reference type. In Java, the API architect uses the Java "Interface" to group classes that are similar in some respect but need not share implementation. An Interface construct within Java contains only constants and abstract (*i.e.*, declared, but not yet implemented) methods. In other words, a Java Interface does not have enough associated implementation for programmers to create objects with that

Interface directly. Instead, such objects arise from classes that are subtypes of the Interface.

27. While the details may be confusing to non-programmers, an important aspect of Java Interfaces is that a class may be defined as a subtype of more than one Interface. In contrast, for reasons having to do with the way Java is implemented, a class cannot be a subtype of more than one class. Therefore, basic features that are shared across many classes are best defined as Interfaces instead of classes. For example, the Comparable Interface is used to identify classes that have the compareTo operation, which is a clever combination of the less-than, greater-than, and equal-to tests. That the Java API designers chose to use a single compareTo operation (whatever its name) instead of three operations called less-than, greater-than, and equal-to, and create an Interface for identifying all classes that define this form of comparison, is a succinct but illustrative example of the elegance of the Java API specification and the creativity in its arrangement. In Java SE 1.5, I counted approximately 50 classes that are subtypes of the Comparable Interface.

28. This type of complex hierarchy and interdependency among thousands of different elements makes the Java APIs both quantitatively and qualitatively different from the example of a user who pushes a button to affect a command. While Dr. Astrachan notes that a programmer who invokes an API need not review the source code associated with implementing a particular method (Astrachan Report ¶ 27), unlike a user who simply pushes a button, a programmer using an API in designing a complex software program will often need to understand the hierarchical rules and interdependencies between classes in an API package, or between packages, in order to program effectively.

29. The value of elegant API design, and the wide array of choices that are available to API authors, is evident from the effort that developers put into "refactoring" existing APIs. Refactoring[1] is "a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior." (M. Fowler, *Refactoring Home Page*, http://refactoring.com/.) There are commercial tools available that are designed to help a programmer better diagnose structural problems and carry out this refactoring process, and refactoring is a frequent seminar topic at industry conferences and events. Refactoring shows that the structure of the API can be determined and revised independent of the functionality of carrying out a method or command. The primary purpose of refactoring is to try to move to a more efficient structure without changing the functionality to the end user: the functionality to the end user remains the same regardless of where the method is implemented or who maintains it.

30. The authors of the Java API specifications made creative design choices throughout the platform, that were not dictated by function, when they decided whether to include methods in Interfaces or at particular levels in the class hierarchy. As an illustration, the class and Interface hierarchy for java.net is shown below. Dr. Astrachan essentially dismisses the creative effort of the designers of the java.net API specification when he states that, "the java.net package contains functionality relating to networking, and every modern mobile software platform must have networking functionality." (Astrachan Report ¶ 129.) There are, however, many creative options available to an API

[1] For an overview of refactoring, *see* T. Mens & T. Tourwé, A Survey of Software Refactoring, IEEE Transactions On Software Engineering, Vol. 30, No. 2 (Feb. 2004) , *available at* http://www.cc.gatech.edu/classes/AY2008/cs6330_summer/readings/mens.pdf.

designer in describing and bringing about that functionality, as the diagram for the java.net package below shows.

Interface Hierarchy

- java.net.**ContentHandlerFactory**
- java.net.**DatagramSocketImplFactory**
- java.net.**FileNameMap**
- java.net.**SocketImplFactory**
- java.net.**SocketOptions**
- java.net.**URLStreamHandlerFactory**

Class Hierarchy

- java.lang.**Object**
  - java.net.**Authenticator**
  - java.net.**CacheRequest**
  - java.net.**CacheResponse**
    - java.net.**SecureCacheResponse**
  - java.lang.**ClassLoader**
    - java.security.**SecureClassLoader**
      - java.net.**URLClassLoader**
  - java.net.**ContentHandler**
  - java.net.**CookieHandler**
  - java.net.**DatagramPacket**
  - java.net.**DatagramSocket**
    - java.net.**MulticastSocket**
  - java.net.**DatagramSocketImpl** (implements java.net.SocketOptions)
  - java.net.**InetAddress** (implements java.io.Serializable)
    - java.net.**Inet4Address**
    - java.net.**Inet6Address**
  - java.net.**NetworkInterface**
  - java.net.**PasswordAuthentication**
  - java.security.**Permission** (implements java.security.Guard, java.io.Serializable)
    - java.security.**BasicPermission** (implements java.io.Serializable)
      - java.net.**NetPermission**
    - java.net.**SocketPermission** (implements java.io.Serializable)
  - java.net.**Proxy**
  - java.net.**ProxySelector**
  - java.net.**ResponseCache**
  - java.net.**ServerSocket**
  - java.net.**Socket**

- java.net.**SocketAddress** (implements java.io.Serializable)
  - java.net.**InetSocketAddress**
- java.net.**SocketImpl** (implements java.net.SocketOptions)
- java.lang.**Throwable** (implements java.io.Serializable)
  - java.lang.**Exception**
    - java.io.**IOException**
      - java.net.**HttpRetryException**
      - java.io.**InterruptedIOException**
        - java.net.**SocketTimeoutException**
      - java.net.**MalformedURLException**
      - java.net.**ProtocolException**
      - java.net.**SocketException**
        - java.net.**BindException**
        - java.net.**ConnectException**
        - java.net.**NoRouteToHostException**
        - java.net.**PortUnreachableException**
      - java.net.**UnknownHostException**
      - java.net.**UnknownServiceException**
    - java.net.**URISyntaxException**
- java.net.**URI** (implements java.lang.Comparable<T>, java.io.Serializable)
- java.net.**URL** (implements java.io.Serializable)
- java.net.**URLConnection**
  - java.net.**HttpURLConnection**
  - java.net.**JarURLConnection**
- java.net.**URLDecoder**
- java.net.**URLEncoder**
- java.net.**URLStreamHandler**

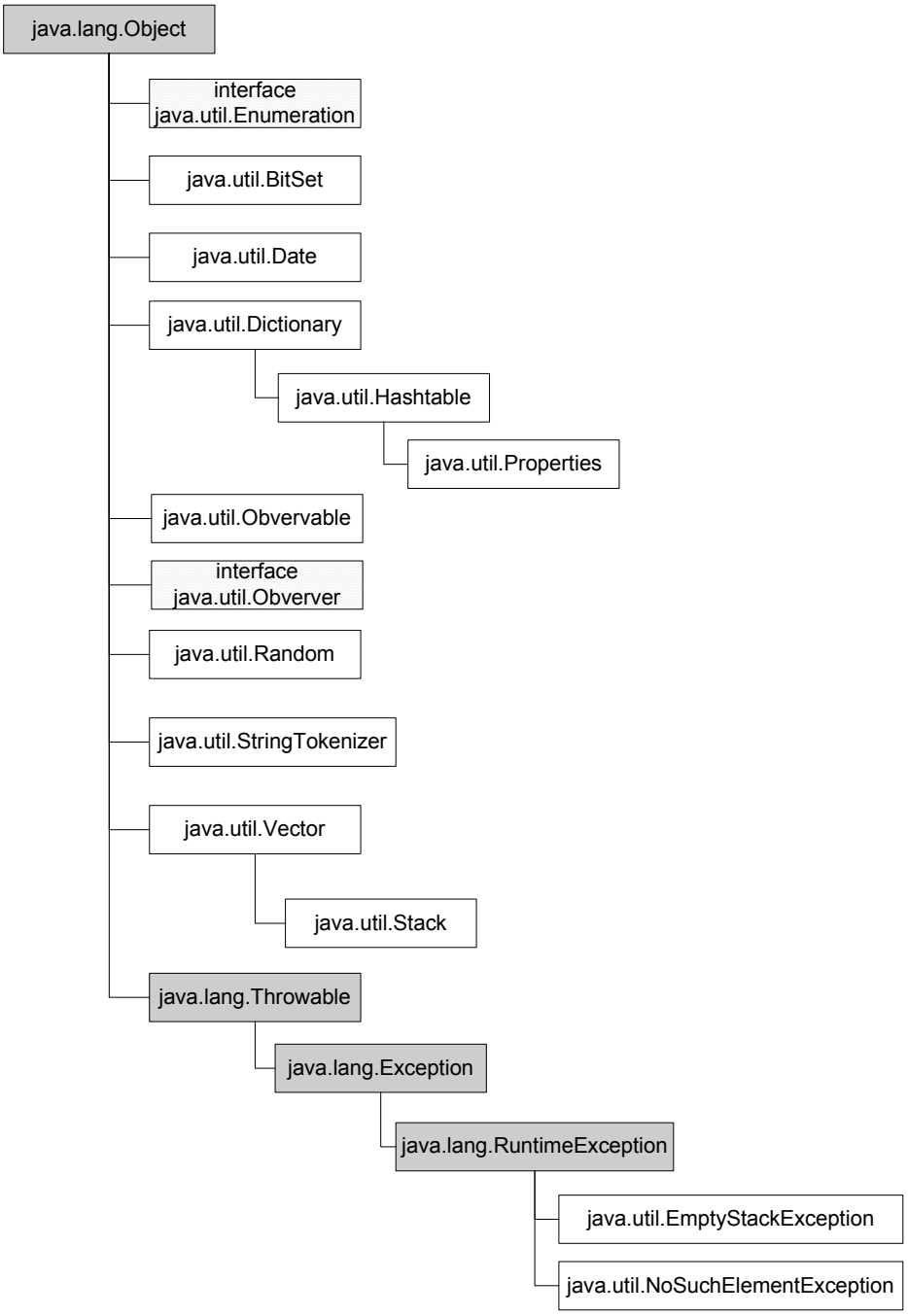
(*See* http://download.oracle.com/javase/1.5.0/docs/api/java/net/package-summary.html).

31. As an initial observation, the java.net API specification includes Interfaces associated with a programming technique called "design patterns." The Interfaces that include the word "Factory" in the name, such as ContentHandlerFactory, SocketImplFactory, and URLStreamHandlerFactory, refer to a design pattern called the factory pattern. It was not an obvious choice to include these classes within the network

package. Some programmers believe strongly in using such design patterns, while others may be skeptical about their merits. In addition, to the extent the programming technique associated with the Factory design pattern, which uses private constructors and public methods to build objects was considered relevant, this pattern did not have to be expressed using Interfaces. Java programs can use the Factory pattern using classes alone. In fact, many Java packages contain no Interfaces at all.

32. The fact that some of the classes within java.net inherit from classes in *other* packages further shows that designing a class hierarchy is a nontrivial task. For example, java.net.URLClassLoader is a subclass of java.security.SecureClassLoader, which is a subclass of java.lang.ClassLoader. Had the designers been following some rule that classes can only inherit from Object and classes within the same package, they would not have been able to take advantage of the common features of these three classes.

33. The design of the java.util package also reflects multiple layers of creative expression. The hierarchy for an early version of the java.util package, adapted from the first edition of James Gosling et al., *The Java™ Language Specification* 616 (1996), is represented below.


java.lang.Object
interface java.util.Enumeration
java.util.BitSet
java.util.Date
java.util.Dictionary
java.util.Hashtable
java.util.Properties
java.util.Obvervable
interface java.util.Obverver
java.util.Random
java.util.StringTokenizer
java.util.Vector
java.util.Stack
java.lang.Throwable
java.lang.Exception
java.lang.RuntimeException
java.util.EmptyStackException
java.util.NoSuchElementException

34. Over time, many additional classes and interfaces were added to subsequent versions of java.util. For example, JDK 1.2 added significant content to the java.util package to completely revamp the way Java handles collections. A listing of the revised interface and class hierarchy for java.util as of JDK 5 is included as Exhibit Copyright-Opposition-A. The fact that the hierarchy and the number of classes and interfaces in java.util could change over time so significantly shows that there was never only one way that the java.util API specification could be structured.

35. Moreover, as the diagram above shows, even the early version of the java.util. API specification was structurally complex. The java.util package included the classes EmptyStackException and java.util.NoSuchElementException, which inherit from java.lang.RuntimeException, which is not part of java.util. The designers had a choice of where to package these exception classes, and they chose java.util instead of java.lang. The package also included an Interface called "Enumeration," which is implemented by the class "StringTokenizer." The chart does not depict a hierarchical relationship between these two elements because the designers chose to define Enumeration as an Interface, not as parent class. It is also worth noting that the Java Standard Edition APIs do not even include a class that implements the Interface "Observer." That task is left to the developer, which again is a conscious design choice.

36. My previous report also showed how different programming languages implement packages in different structural ways, and include different elements in them, further demonstrating that there simply is no one right or functional way to design a package or a class for performing a particular function or set of functions. (*See* Opening Report ¶¶ 179-182, 191-199.)

37. Even the class java.lang.Math, which Dr. Astrachan cites as an example of unoriginality, embodies expressive choices. While one might expect to see methods such

as "cos" and "log" in a math library, the presence of two methods called "ulp," which returns the "unit of least precision" of a floating point number, is hardly standard. The description of the "ulp" methods from the JDK 5 documentation[2]—which Dr. Astrachan omits from the table at paragraph 39 of his report—is printed below. Other platforms, such as C++, express the same idea differently. For example, the Boost C++ Library's "float_distance" takes two parameters—not one, as with ulp—and returns the distance between them. (*See* http://www.boost.org/doc/libs/1_47_0/libs/math/doc/sf_and_dist/html/math_toolkit/utils/next_float/float_distance.html.)

`public static double` **`ulp`**`(double d)`	Returns the size of an ulp of the argument. An ulp of a `double` value is the positive distance between this floating-point value and the `double` value next larger in magnitude. Note that for non-NaN *x*, `ulp(-`*x*`) == ulp(`*x*`)`.
`public static float` **`ulp`**`(float f)`	Returns the size of an ulp of the argument. An ulp of a `float` value is the positive distance between this floating-point value and the `float` value next larger in magnitude. Note that for non-NaN *x*, `ulp(-`*x*`) == ulp(`*x*`)`.

38. The decision of which methods to include within java.lang.Math was also an expressive choice. The class java.lang.Math is not the only "math" library within Java. Instead, Java Standard Edition contains a separate package called java.math that "[p]rovides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal)." (http://download.oracle.com/javase/1.5.0/docs/api/java/math/package-summary.html.) While I understand that the java.math package is not being asserted in this infringement

[2] (http://download.oracle.com/javase/1.5.0/docs/api/java/lang/Math.html.)

case, the decision to include some familiar mathematical methods in java.lang.Math and different yet still familiar methods in java.math represents a creative choice by the Java API architects. The java.math package was not even part of the original Java Development Kit, which further shows that its inclusion is not compelled for any technical or functional reason. (*See id.* (noting that java.math has only existed since JDK version 1.1).) The decision where to include particular classes in the Java API specifications is nontrivial due to the complex interdependencies between elements.

D. The Names in the Java Platform Reflect Original Expression

39. I disagree with Dr. Astrachan's assertion that the names of the classes and methods are all dictated by function. I have already discussed in my opening report how I believe the names reflect creative expression. This is another instance where Dr. Astrachan s overlooks the creative process of designing an API. Instead of asking what possible names could be used, he seems to use the fact that many Java API names "make sense" to suggest that that they are effectively the only way to name a particular method or class. For example, Dr. Astrachan provides a chart, at pages 60-62 of his report, of class names, with a description of their "functionality" and package location. (Astrachan Report ¶ 107). The particular names chosen by Dr. Astrachan do give the programmer an indication of what their likely function will be. But that does not mean that these were the only names that could have been chosen, or that the choice of names was constrained by functionality.

40. To try to support his point, Dr. Astrachan has selectively chosen names, like "sqrt" that tie closely to their associated function. But there are many examples of names that do not. In the java.net package, for example, you will find the name Proxy, which apparently refers to settings for a proxy, rather than a network proxy itself. In the java.beans package, you find an Interface named Customizer and a class named

Introspector. While these names are chosen with some attention to the English meaning of these words, it is hard to see that the names are in any sense uniquely determined by the functions associated with them. In the Java collection hierarchy, the Iterator Interface uses methods named "next" and "hasNext," while the Beans naming convention associated with java.beans would suggest "hasNextElement" and "getNextElement" instead. In commenting on their choice of names in a set of answers to Frequently Asked Questions (FAQ), the Java Collections API authors explain that, "We suspect that the collections APIs will be used quite pervasively, often with multiple method calls on a single line of code, so it is important that the names be short." They further justify their decision by giving the following code sample

```
for (Iterator i = c.iterator(); i.hasNext(); )
       System.out.println(i.next());
```

and stating, "Everything fits neatly on one line, even if the Collection name is a long expression." (See Java Collections API Design FAQ, *available at* http://download.oracle.com/javase/1.4.2/docs/guide/collections/designfaq.html.) In other words, these designers recognize that they departed from a systematic naming convention that many programmers might expect, but they consider their choice more aesthetically pleasing.

41. Similarly, while it is certainly true that there are some names, or parts of names, like "char" or "printf" that are found in other languages, choosing to use such names was in itself a choice that did not have to be made, and there are scores of examples of names, like the ones cited above, that are not familiar or established.

42. I find Dr. Astrachan's statistical analysis at page 65 of his report to be revealing. According to Dr. Astrachan there are *7,796* methods in Oracle's

implementation of Java. (Astrachan Report ¶ 114.) (I have not checked myself to verify the accuracy of this number.) When there are this many names, it does not make sense to parse them into average word length and claim they are not original. Regardless of assigning names to methods associated with functions, or familiar names drawn from other sources, no two programmers could come close to arriving at the exact same naming conventions and names across the board for this many methods. I see this with my students all the time: even when I give relatively simple assignments requiring relatively few names to be generated, there is still a lot of variation. I find it absurd to suggest that there is no originality in the naming of nearly 8,000 methods, plus the associated classes and packages in which they are found. And Dr. Astrachan does not even address Google's copying of all the field names, types, and modifiers, which I discuss in more detail below.

43. My conclusion is not changed by Dr. Astrachan's claim that "nearly one-third" of these names are "determined" by style guidelines set forth in the Java Language Specification. That still leaves more than 5,500 names that are not. And for the ones that are, the guidelines only specify the basic format of the name, such as "descriptive nouns or noun phrases" for classes or "verbs or verb phrases, in mixed case" for methods. (Astrachan Report ¶ 113.) The rules do not specify the substance of what the name must contain, and still leave ample room for originality, particularly across more than 2,000 names. In addition, as I pointed out above, experienced API designers can and do make considered naming choices that deviate from "standard" naming conventions. I also find it ironic that Dr. Astrachan contends that there is no originality to the names because some of them are set according to style rules that Sun itself created. Google, of course, could have come up with its own style rules, which would have produced names in a

completely different format. It chose instead to copy Sun's class, method, and field names essentially word-for-word.

44. I also note that Dr. Astrachan's report does not address the selection, naming, or arrangement of the input parameters for a method. As I explained in my opening report, in object oriented code, like Java, the method signature includes the name of the method as well as its associated parameters. These parameters themselves have names, and the choice of which parameters to include with that method in the API, and the order in which the parameters should be arranged, give additional opportunities for creativity. As also noted in my opening report, the names of the parameters can be chosen according to the taste and aims of the API designer, because developers do not have to make use of the parameter names to invoke the API. When these parameters and their names are combined with the thousands of method names, the possible choices for names become even more diverse.

C. Dr. Astrachan's Report Does Not Address Google's Copying of Data Field Names and Organization

45. Dr. Astrachan's report also does not address the presence of fields, which form an important feature of the architecture of the Java API specification. Java is an object-oriented programming language, and in Java, classes are a way to encapsulate both data stored in fields and functionality, defined in methods.

46. One important decision in designing a class is whether to make its data fields public—that is, accessible to methods of other classes—or private. One common design technique is to replace a public field with a private field, and allow public access via "get" and "set" methods. This allows the class designer to change the underlying data structure of the field in the future without changing its interactions with other classes or

application code. Notably, fields, unlike methods, do not execute procedures. They merely store data.

47. In this regard, field names are analogous to data labels in database design. They are typically chosen to express or describe the content of the data elements. Like method names, they could be anything (like "Steve," as Dr. Astrachan suggests). Any name would still work fine from the computer's perspective, but confusing or idiosyncratic names would be a poor choice from the human developer's perspective. Developers need to absorb and internalize the spirit and organizing principles of an API specification so they can use the problem-solving concepts it presents properly. In contrast to methods, there is no code written to perform a function when a field name is used.

48. The fields defined in Sun Microsystems's (now Oracle's) Java library API specifications and the fields defined in Android's API specifications are virtually identical. According to analysis conducted by Alan Purdy, among the 458 classes that appear in both API specifications, there are 893 fields defined in the Android API specifications that have the same modifiers, types, and names as the fields defined in the Oracle Java API specifications. There are only 34 fields in the Android API specifications that do not appear in the corresponding Oracle Java API specification with exactly the same modifiers, types, and names. There are only 167 fields in the Oracle Java API specifications that do not appear in the corresponding Android API specification with exactly the same modifiers, types, and names. (Opening Expert Report of Alan Purdy Regarding Copyright ¶ 17.)

49. The selection and arrangement of about 1,000 fields, as well as the modifiers, types, and names, among 458 common classes is such that the combination of these elements is original and creative. No two programmers, tasked with implementing

the same functionality, are likely to ever come close to arriving at the exact same fields in the API specifications for this many fields.

50. The extent of similarity is even greater when one considers the private fields that are not part of the Oracle Java API specifications but were adopted by Google anyway. As I mentioned, it is often good programming practice to "encapsulate" class fields by declaring a field to be private, not public, and allow public access via "get" and "set" methods. The "roughly 2,000 [methods] that begin with either 'get' or 'set'" according to Dr. Astrachan (Astrachan Report ¶ 114) are part of this encapsulation. This means that when a class has a method named "getXXX," the class will typically also have a private field named "XXX." This can be seen in the class ProtectionDomain.java, the Android version of which is shown in Exhibit Copyright-G of my opening report. The class ProtectionDomain is defined to suggest public methods named "getCodeSource," "getClassLoader," "getPermissions," and "getPrincipals." The class ProtectionDomain is defined to suggest fields named "codeSource," "classLoader," "permissions," and "principals," all declared private and so not visible as part of the API specification. Yet all of these Android fields, their modifiers, types, and names, are the same as those found in the Oracle Java API implementation of the ProtectionDomain class. So Google's copying extends beyond what is expressed in the API specifications to the fields of its source code implementation of the API as well.

51. For similar reasons as I have discussed above, Google's replication of Oracle's selection, arrangement, and organization of about 1,000 fields, including their modifiers, types, and name, was not done for functional considerations, and particularly not for the fields declared private, which are not publicly exposed and so are not needed at all for any compatibility reasons.

D. Android's API Documentation Is Substantially Similar To Oracle's API Documentation.

52. I disagree with Dr. Astrachan's opinion that Android's documentation is not substantially similar to Oracle's API documentation. I have included a side by side comparison of some of the specifications for the Java and Android APIs in exhibits E and F of my opening report. As the comparison shows, the names running down the left hand column of the API specifications at issue are virtually identical. When one considers the hundreds of pages of specifications that make up the APIs, this similarity is more than just striking. It is impossible to explain by anything other than copying. In fact, Robert Lee, the lead core library developer for Android, admitted at his deposition that he "consulted Sun's websites for the API specifications when doing the work for Google." (8/3/2011 Lee Dep. 65:8-66:16.)

53. Dr. Astrachan appears to assume, for the purposes of his comparison of the documentation, that all of the names of the API components have already been chosen. (*See* Astrachan Report ¶ 146.) But, as described above, there is considerable creative expression in the choice of the names in the Java APIs, and certainly in the compilation of the thousands of method, package and class names reflected in the documentation that is common to both parties.

54. I also do not agree with Dr. Astrachan that "Two different authors documenting the same API components would necessarily write very similar documentation because they are describing the same functionality." (*Id.*) While this might hold true for the description of a limited number of components, one would not expect it to hold true across thousands of components; rather, one would expect to see significant variations in the choices used to describe them. In this case, a very high percentage of the English language text descriptions contained in the Android

specifications appear to have been written to closely track the text descriptions in the Java APIs.

55. Dr. Astrachan attempts to minimize the significance of the documentation, stating that "just as one does not say that the definition of a lion in a dictionary is, in fact, a lion, so the API's specification is not the API, but rather the description of the API." (Astrachan Report ¶ 55.) In my opinion, in the common parlance used by software developers, the API specification is considered to be part of the API. Moreover, from the standpoint of copyrightable expression, the specification is a written expression of the API. The source code is also a written expression of the API, one that is derived from and implements the API specification. At the end of the day, Dr. Astrachan's distinction is one without a difference. Google copied and derived from both.

E. It Was Possible to Create a Platform for Java-Language Programs Without Copying the Java APIs

56. I disagree with Dr. Astrachan's conclusion that it was necessary for Google to copy the Java API specifications to achieve "basic functionality and interoperability." (Astrachan Report at 73.) I agree that it was extremely valuable and desirable for Google to copy the Java API specifications and implement them in Android. As explained in detail in my opening report, among other advantages, including the Java APIs saved Google critical time to market, and provided developers with a familiar programming environment that encouraged them to develop applications for Android. This paved the way for an almost immediate explosion in the development of applications for the platform, which have been key to Google's strategy and its remarkable success. (*See, e.g.*, Opening Report ¶ 97.) But desirability and value should not be confused with functionality.

57. As discussed above, it is not functionally necessary to include the Java APIs or class libraries (except perhaps for a very few classes like Object and Class that are tied closely to the Java language) in order to make use of the Java language. In general, the libraries contain prepackaged code, written in the Java language, which allows the developer to avoid having to rewrite the code herself. But the developer still can write the equivalent code herself if she wishes. More to the point, Google could have independently created its own API specifications for Android class libraries. It did not have to copy Oracle's.

58. Many organizations have developed their own libraries for other programming platforms. As mentioned in my opening report, C++ is the one of the most popular programming languages. For C++ alone, there are several organizations that have developed versions of the C++ Standard Library.[3] Boost has developed several C++ libraries.[4] Silicon Graphics developed its own "Standard Template Library."[5] Writing one's own libraries for a successful platform is hardly unheard of.

59. Over the years there have been unofficial APIs for Java as well. For example, before Sun added the Collections Framework in JDK 1.2, developers relied on the third party Generic Collection Library, which provided similar functionality to the Collections Framework but had a different structure. (*See* http://web.archive.org/web/19980614025724/http://www.objectspace.com/jgl/.) Bob Lee, who was the core library lead for Android prior to 2010, also contributed to a different set of Collections implemented in Java that is now called Guava. (8/3/2011 Lee Dep. 13:20-22.)

---

[3] (*See, e.g.*, http://www.roguewave.com/products/sourcepro.aspx.)

[4] (*See* http://www.boost.org/.)

[5] (*See* http://www.sgi.com/tech/stl/table_of_contents.html.)

60. I noted in my previous report that Objective-C, which had a much smaller developer base than Java before 2006, formed the basis for the iPhone, which is one of the most successful devices on the market today. In this case, not only did developers have to learn new APIs but a whole new language. Apple's success shows that it should have been possible for Google to eventually get developers to adopt new APIs for a language they already knew, if it had a strong enough product. Apple had the significant advantage of being the first to market, and was the company that essentially created the market for a certain class of devices. Because of Apple's head start, it would have been much more difficult for Google to compete with Apple if it went to market with an unfamiliar application development environment, and so it is likely that had Google not copied from Oracle, Android would not have enjoyed the phenomenal growth rate in applications that it has had. But from a functional, compatibility and industry standpoint, Google could have developed its own core libraries and APIs that differed substantially from the Java APIs at issue.

61. Additional proof that implementing the Java APIs was not mandated for functionality and interoperability is the fact that Google chose not to implement some of the Java core library APIs and instead replaced them with its own API specifications. Many of the APIs that relate to Android's user interface, for example, were designed by Google. Although Google's user interface APIs express many of the same ideas as the Java user interface APIs, they are very different.

62. Google's deviation from the Java APIs has two implications. First, Android application developers are required to learn new APIs in order to make their programs work with the Android user interface. Second, many existing and future Java programs cannot run on Android because they require Java APIs that Google chose not to

implement in Android. For these reasons, the argument that Google's copying of the Java API specifications was for "compatibility" rings hollow.

63. I will explain in more detail. The Java user interface APIs and the Android user interface APIs express the same idea: there should be a uniform yet flexible user interface to permit an end user to interact with an application, and the interface should include components that are familiar to users, such as buttons, check boxes, labels, tabbed panels, and scroll panes. In the Java APIs, this functionality is provided by the APIs for AWT (java.awt and java.awt.*) and Swing (javax.swing and javax.swing.*). Google, however, wanted its own user interface, and chose to develop its own user interface APIs for Android application developers to use. With the exception of the java.awt.font API, which Google copied, Android does not include the AWT and Swing APIs. As a result, the very large number of Java programs that were written to employ these APIs cannot run on Android—so Android is not in fact compatible with Java SE.

64. Instead, Google wrote its own user interface APIs, including such APIs as android.app and android.view. These APIs are incompatible with the Java user interface APIs, although from the user's perspective they perform the same functions (opening and closing dialog boxes, selecting checkboxes, picking an item from a list).

65. As an illustration of the differences between the Java and Android user interface APIs, consider how a developer would use them to cause an application to open a dialog box for a user that displays a message and includes a button marked "OK," which will cause the dialog box to disappear when clicked. Using the Java AWT and Swing APIs, a developer can open a dialog box by using a variety of classes, such as JOptionPane, ProgressMonitor, JColorChooser, and JFileChooser, which are part of the javax.swing API. (*See* http://download.oracle.com/javase/tutorial/uiswing/components/dialog.html.) Developers

writing applications for Android do not use these classes, because Google did not provide an implementation of the javax.swing API in Android and manufacturers are forbidden from adding one through Google's "compatibility" requirements. Instead, a developer can cause an Android application to open a dialog box for a user using classes such as AlertDialog and AlertDialog.Builder, which are part of the android.app API. (*See* http://developer.android.com/guide/topics/ui/dialogs.html#AlertDialog and http://developer.android.com/reference/android/app/AlertDialog.html.)

66. Here is an example of Java code that employs the javax.swing API to open a dialog box that will show a simple message and a button marked "OK":

```
JOptionPane.showMessageDialog(frame, "Do you
like green eggs and ham?", "Message");
```

67. Here is an example of Android code that employs the android.app API to open a dialog box that will show the same message. First the box needs to be defined:

```
AlertDialog.Builder builder = new
AlertDialog.Builder(this);
builder.setMessage("Do you like green eggs and
ham?")
       .setNegativeButton("OK", new
DialogInterface.OnClickListener() {
           public void onClick(DialogInterface
dialog, int id) {
                dialog.cancel();
           }
       });
AlertDialog alert = builder.create();
```

68. And then to show the dialog box, the Android developer would use the code:

```
showDialog(ID);
```

where "ID" is the ID of the dialog that was created using the code above, and showDialog is a method of the Activity class of the android.app API.

69. This comparison shows that Android and Java can have very different APIs for performing the same functions. The Android and Java authors found different ways of expressing the same idea (such as "display a dialog box for the user").

70. The Android designers could have done this with all of the other Java APIs at issue (except perhaps for a very few classes like Object and Class that are tied closely to the Java language). As I have discussed above, Google could have independently created its own API specifications for Android class libraries (just as it did for the user interface API specifications) but chose not to.

F. Dr. Astrachan's Assertion That Oracle and Sun Implemented Pre-existing APIs Does Not Appear to Be Relevant to the Copyright Claims At Issue

71. Dr. Astrachan's discussion of the alleged implementation of APIs by Sun and Oracle does not appear to have any bearing on the issue of Google's copying of the Java APIs. Moreover, Dr. Astrachan does not provide sufficient information or context in his report for me to be able to evaluate his claims.

72. The report includes a list of 324 function names purportedly implemented by Microsoft Excel as well as StarOffice but which appear to have originated in VisiCalc in 1979. That program has of course long since been discontinued. It is not clear from the information that Dr. Astrachan provided whether these names are today in the public domain, licensed, or considered proprietary.[6] In any event, from the little information that is provided in his report, Dr. Astrachan's examples do not appear to be analogous to

[6] I understand that there are numerous agreements between Microsoft on the one hand and Sun and Oracle on the other that may undermine Dr. Astrachan's implicit assumptions. Because Dr. Astrachan has not made the case that there is anything proprietary with respect to the Microsoft Excel function names, there is no need to investigate further.

the API specifications at issue in this case. As I have explained above, an API specification is much more than a list of names. In the case of the Java platform, it includes the selection, coordination and arrangement of the Java packages; the class names and definitions, fields, methods and method signatures contained in each package; and the prose text that explains each of these elements. By providing only a list of function names from spreadsheet programs for comparison, Dr. Astrachan's report does not account for the many expressive dimensions that are present in the Java platform.

73. Dr. Astrachan's report also includes a reference to SQL, again dating back over thirty years ago to 1979. SQL is a relatively simple language, and Dr. Astrachan's report makes reference to only a handful of names and functions. It is not clear whether Dr. Astrachan is contending that these were included as part of the SQL APIs or simply as part of the SQL language. It is not clear from the information that Dr. Astrachan has provided whether these names are today in the public domain, licensed, or considered proprietary.[7] From the little information he provided in his report, these names and functions do not appear to be comparable to the Java APIs. And of course, the VisiCalc and SQL APIs are completely irrelevant to the issues of whether the Java APIs are copyrightable and whether Google copied them.

74. I do not understand the significance of Dr. Astrachan's discussion of the Solaris operating system's supposed use of APIs from Linux. Dr. Astrachan's report makes no attempt to compare the Linux APIs he discusses to the Java API specifications at issue in this case. In any event, like the discussion of VisiCalc and SQL APIs, this

[7] I understand that there are also many agreements between IBM on the one hand and Sun and Oracle on the other that may undermine Dr. Astrachan's implicit assumptions. Again, however, because Dr. Astrachan has not made the case that there is anything proprietary with respect to the SQL names, there is no need to investigate further.

issue is not relevant to the question of whether the Java APIs are copyrightable and whether Google copied them.

II. THE ANDROID SOURCE CODE THAT IMPLEMENTS THE ASSERTED APIS EMBODIES THEIR OVERALL DESIGN AND STRUCTURE, NOT SIMPLY ISOLATED NAMES OR LINES

75. The Android source code that implements the API specifications contains the same copyrighted API elements that are discussed above. (*See, e.g.*, ¶¶ 19-24, 26, 29). Comments in the Android source code are, in many cases, not identical to the corresponding prose descriptions in the Java API specifications, but they are generally quite similar.

76. Dr. Astrachan states in his report that the only element from the Java API specifications that is copied into the Android source code is the method name and declaration and some "required organizational lines." (Astrachan Report ¶¶ 57-60.) This is not correct. The entire extensive API structure contained in the 37 packages at issue, with all of the hierarchical relationships and interdependencies, has been implemented in the Android source code. For example, if a class is a subtype of an Interface, or a subclass of another class, this is stated explicitly in the source code using the Java language construct for expressing this relationship. Further, documentation associated with elements of the API is included in the source code, in a form intended for display in a browser.

77. Google deliberately chose to copy, not just the names, but the API structure and organization associated with those names. In my previous report, I included examples of how the Android source code copies lines from the Java API specifications verbatim, or nearly verbatim, and then includes code that implements the precise behavior that is called for by that line of the Java API specification. (*See* Opening Report ¶¶ 209-19.) Google repeated this process literally thousands of times, until it had

replicated all, or nearly all, of the methods contained in the Java API specifications, in accordance with the same architectural structure.

78. The testimony of Bob Lee, the lead core library developer for Android, confirms this. A document written by Mr. Lee in 2008 states that he was tasked with "re-implementing" the Java APIs. (GOOGLE-40-00034698.) Mr. Lee testified that this re-implementation work referred to the work that he did for Google, with the assistance of the Noser software development group retained by Google, to "implement core libraries according to the Java APIs." (8/3/2011 Lee Dep. 14:7-11.) Mr. Lee testified that his development team would read the API specification, which describes the "span of the implementations." They would make sure that the code they had written properly implemented the described specification by debugging it based on errors found when Google wrote and ran test code for the specification or ran others' test code, or when others ran previously existing Java code on Android and reported bugs. (*Id.* at 64:8-20.)

79. Further, Mr. Lee admitted in his deposition that he "consulted Sun's website for the API specifications when doing the work for Google" and that he "assume[d]" the Noser development team did as well. Mr. Lee testified that the reason for consulting these specifications was to ensure that the code was behaving as it was supposed to, since the API specifications contained the description of the required behavior associated with each method. (*Id.* at 65:8-66:16.) Mr. Lee testified that he did see that there were copyright notices on the specifications, but proceeded anyway, without consulting an attorney. (*Id.* at 66:17-67:5.)

80. The copying described by Mr. Lee, and confirmed by the contents of the Android source code itself, is not simply the copying of names. It is the wholesale implementation of Oracle's APIs, including their organization and structure, into the code. The API specifications are the blueprint to the Java Class Libraries. As I have

described above and in my previous report, this blueprint reflects many creative design choices that were made along the way. It saved Google an enormous amount of time to start from and copy the blueprint. Instead of figuring out for itself what to include in these APIs, and their interdependencies, Google simply used the blueprint and wrote segments of code to carry out each of the required functions. To say that this is not copying is like saying somebody who copied the architectural structure of a building and its rooms, hallways and pipes, gets to get away with it because they filled the rooms with their own furniture, or built an addition on the back.

81. Google's copying of Oracle's Java blueprint was deliberate. As Dr. Astrachan acknowledges, and as is discussed in more detail below, there was tremendous value to Google in offering APIs that were already familiar to Java developers in order to attract them to the Android platform.

82. In my opening report, I expressed the opinion that Google literally copied source code, object code and comments from Oracle. Dr. Astrachan's report does not dispute this conclusion, and does not refer to any analysis of the code at issue that was conducted by him or by anybody acting at his direction. Instead, Dr. Astrachan tries to minimize the significance of this copying.

III. DR. ASTRACHAN DOES NOT DISPUTE IN HIS REPORT THAT GOOGLE LITERALLY COPIED SOURCE CODE, OBJECT CODE AND COMMENTS FROM ORACLE.

83. In my opinion, this copying is not reasonably subject to dispute. The "rangeCheck" method from Oracle's Arrays.java, for example, is identical in Android's TimSort and ComparableTimSort, right down to the unusual spacing. Dr. Astrachan states in his report that this code was "necessary for API compatibility with other sort implementations" and so "using code extremely similar to this was necessary for TimSort to be completely compatible with other sort implementations." (Astrachan Report ¶ 157.)

This is not correct, however, because, as indicated in line one of the copied code, this method is designated "private" and so it is not exposed to other sort implementations and does not need to be compatible with them. (*See id.* ¶ 152.) I note also that the comments preceding the Oracle Java and Android versions of the rangeCheck method are virtually identical, something that Dr. Astrachan does not address.

84. With respect to the copied rangeCheck method, Dr. Astrachan focuses on the wrong chronology. He states that the code was written by Google employee Joshua Bloch (formerly of Sun), that "the files were first included in Android" and that it "appears that Google offered the files to Oracle." (*See id.* ¶ 159). The facts do not support this conclusion. Dr. Astrachan overlooks the deposition testimony of Mr. Bloch himself, who acknowledged that he wrote the rangeCheck code while he was at Sun before leaving for Google, and that he knew Sun owned the intellectual property rights to that code. (Bloch Tr. at 175:21-176:7.) It also overlooks Mr. Bloch's testimony that a comparison of the code give a "a strong indication" that he accessed Sun's source code while he was working on the TimSort code for Android. (*See id.* at 181:9-14).

85. Mr. Bloch's admission that he likely had access to, and was reviewing, Sun's source code while working on developing Android is highly significant. While the file itself is not large, consisting of 9 lines of code, Mr. Bloch copied the rangeCheck method in its entirety from proprietary Sun source code. The Sun arrays.java source code file calls the rangeCheck method nine times in the arrays.java file, making it qualitatively significant to the organization and process flow of arrays.java.

86. While rangeCheck is a short piece of code, there may be more to it than meets the eye. The code tests some relationships between three values, arrayLen, fromIndex, toIndex. If any of three conditions turn out to be true, an error condition is reported. There are three possible errors, but only one error will be reported, even if more

than one occurs. For example, if fromIndex > toIndex and toIndex > arrayLen, which is certainly a possible combination for these numbers, then there are two error conditions. The author of the code arranged the tests in order so that in this situation, the first error condition is reported and the second one is not. While I have not studied the use of this code in any detail, it is certainly possible that some amount of trial and error went into figuring out how to arrange the tests in this code so that the most informative error condition is reported.

87. In addition, as I noted in my opening report, there are eight Android program files that have significant textual similarity to the corresponding decompiled Oracle class files. Six of the corresponding Oracle Java source code files (AclEntryImpl.java, AclImpl.java, GroupImpl.java, OwnerImpl.java, PermissionImpl.java, PrincipalImpl.java) appear in the src/share/classes/sun/security/acl directory, one (PolicyNodeImpl.java) is in src/share/classes/sun/security/provider/certpath, and one class (AclEnumerator.java) is actually defined in the same file as another (AclImpl.java). I examined the eight Android program files shown in Exhibits Copyright-J through Copyright-Q of my opening report and Oracle's Java source code corresponding to these classes.

88. In my opening copyright report, I described the comparison conducted by Alan Purdy which suggests that the Google files are the result of decompiling class files. In addition, it would be extraordinarily unlikely to implement source code programs this long with this degree of similarity to another program without copying. Dr. Astrachan does not dispute that the Google files are the result of copying from compiled Oracle Java files.

89. In looking at these files further, I have found additional evidence that they are indeed the result of decompilation. For example, the Oracle Java

PolicyNodeImpl.java source code contains the following lines (numbered here for reference):

```
172   public String toString() {
173   StringBuffer buffer = new StringBuffer(this.asString());
174
175   Iterator<PolicyNodeImpl> it = getChildren();
176       while (it.hasNext()) {
177           buffer.append(it.next());
178       }
179       return buffer.toString();
180   }
```

90. This code contains the instantiation of the generic Iterator<E> interface from java.util. (*See* http://download.oracle.com/javase/1.5.0/docs/api/java/util/Iterator.html.) The way Java implements the instantiation of a generic class, as in Iterator<PolicyNodeImpl>, involves something called "type erasure" that results in a Java operation called a runtime type cast. (*See, e.g.*, http://download.oracle.com/javase/1,5.0/docs/guide/language/generics.html.) The compiled and then decompiled version of the code above has a type cast (PolicyNodeImpl)iterator that is indicative of the compilation of Java generics. The same code form appears in the Google Android code. However, as pointed out in the Java documentation for generics, this usage of casts is undesirable and not recommended. Therefore, it is highly likely that the Google code is the result of automatic decompilation, and unlikely that this code would have been written by any programmer familiar with the constructs used in modern Java.

91. Dr. Astrachan asserts that "These files are also in large part "dummy" files — instead of having complex logic, they return certain, fixed values, which is a common practice in test files." (Astrachan Report ¶ 163). He supports his conclusion with the example code for the isNegative() method in AclEntryImpl.java and states that "In a real (not test) file, the 'isNegative' method would do some complex logic to

understand whether the quality was negative. Here, because this is a 'dummy' file used for test purposes, no logic or work is done— instead, it simply immediately returns 'negative.'" (*Id.* at ¶ 164). I disagree with this analysis and his conclusion.

92. In the actual Java source code from which the Oracle compiled and decompiled code was produced, there are comments explaining the purpose of this class and the isNegative() method. The AclEntryImpl class "is a class that describes one entry that associates users or groups with permissions in the ACL." Here "ACL" is an acronym for "access control list," a list indicating permissions. The code comments further specify that an entry may be used to grant or deny permissions, and entries that deny permissions are considered negative. Further, it is clear from the source code that "negative" itself is a private field of this class, storing information about the ACL entry. Thus a method that returns the value "negative" in fact returns the informative value stored in a private field called "negative." This is a real file with meaningful function—it is a class that implements the AclEntry Interface. Dr. Astrachan is incorrect when he says that "In a real (not test) file, the 'isNegative' method would do some complex logic to understand whether the quality was negative."

93. There are also other methods in this and other apparently decompiled files that are more extensive than the isNegative() method that Dr. Astrachan chose as an example. The toString() method of AclEntryImpl.java is reproduced below as an illustration.

```
  /**
   * Return a string representation of the contents of the ACL
entry.
   */
        public String toString() {
          StringBuffer s = new StringBuffer();
          if (negative)
             s.append("-");
```

```
        else
           s.append("+");
        if (user instanceof Group)
           s.append("Group.");
        else
           s.append("User.");
        s.append(user + "=");
        Enumeration e = permissions();
        while(e.hasMoreElements()) {
           Permission p = (Permission) e.nextElement();
           s.append(p);
           if (e.hasMoreElements())
              s.append(",");
        }
        return new String(s);
```

94. The seven source code files associated with the eight apparently decompiled files all have the word fragment Impl as part of their filename. Comments in these files, such as "a class that describes one entry that associates users or groups with permissions in the ACL" in AclEntryImpl.java, "Maintain four tables. one each for positive and negative ACLs" in AclImpl.java, and "This class provides an implementation of the <code>PolicyNode</code> interface" in PolicyNodeImpl.java, contradict Dr. Astrachan's characterization of them as "'dummy' files." These files contain code that implements the Interfaces designated in the code and performs functions named and described in the source code.

95. Dr. Astrachan also characterizes these apparently decompiled code files as "test" files. He provides scant evidence for this, other than a Google file directory path that contains the word "test." In Oracle Java, the files are not stored in "test" directories; they are part of the sun.security.acl.* package. I note that the files are not part of the java.security package, which means that Google did not copy them for compatibility, but for convenience. Even if Dr. Astrachan were correct and Google used these files only for testing, software testing is a significant part of any software development effort. It is well

known, for example, that testing software often requires more time and effort in software companies than writing the code initially. It would be foolish and unusual for any company to knowingly release code to market unless it were tested in some way. Further, with regard to these files in particular, clearly someone or some group of engineers found it useful to produce them. Therefore, I believe it is accurate to conclude that these files were important in some way to Google's Android development process.

96. Finally, Dr. Astrachan attempts to minimize the significance of this copying by comparing the number of lines of code that were copied to the number of lines in the entire Android or Java development platforms as a whole. I do not believe that this is the right measure to use. Under that standard, an individual would be permitted to steal files at will, as long as the program from which the file was stolen was sufficiently large, or the program to which the stolen file was added was sufficiently large. In my opinion, that would be contrary to what most software programmers view as acceptable ethical conduct, and it would create the wrong incentives. A more informative way of looking at Google's actions is to see that Google copied the entirety of eight original works (the class files) by decompiling them from an Oracle binary and distributed the entire copied works in the form of Android source and re-compiled binaries. Further, if one were to make a broader, quantitative and qualitative comparison of the sort suggested by Dr. Astrachan, the API specifications that were copied and implemented throughout the Android platform should be included as well, not merely the literally copied code. As I have emphasized in my opening report and throughout this report as well, the APIs are an extremely important part of the Java platform, and are key to its organizational structure.

97. These 37 APIs in particular appear to comprise most of the core APIs that Google believes to be necessary for application developers. Google's Android developer

website states: "Android includes a set of core libraries that provides most of the functionality available in the core libraries of the Java programming language." (Android Developer website, *available at* http://developer.android.com/guide/basics/what-is-android.html.) Android's core libraries include the 37 packages at issue in this case. Even Dr. Astrachan recognizes the significance of the APIs in his report, contending that once Google made the decision to give developers the ability to write applications in Java, Google was "essentially required [ ] to include the APIs at issue." (Astrachan Report ¶ 130.) He also claims that "Industry and developer practice would tend to make it very difficult for Google to choose a different API, or modify an existing Java API, when the Java language is used and supported by Google." (*Id.* ¶ 135.)

98. Dr. Astrachan does not dispute that there are also two Android files that contain comments that were copied from Oracle's implementation. Dr. Astrachan found there were 8 comments in Android's CodeSourceTest.java that "appear to be the same" as comments in Oracle's CodeSource class, and that there were 12 comments in Android's CollectionCertStoreParametersTest.java that "appear to be the same" as comments in Oracle's implementation of the CollectionCertStoreParameters class. (Astrachan Rerport at ¶ 170).

99. Dr. Astrachan states correctly that the comments are not required for the functionality of the product, which is the test he frequently uses elsewhere in his report to conclude that the copied software is not copyrightable. However, rather than acknowledging that this lack of functionality tends to make comments expressive, Dr. Astrachan opines that it means they did not add material value because they "would not have been distributed as part of any Android-based products." (Id. at 171). Comments generally add value, however, in that they make source code more easily useable by the developer community, in this case the community that Google was trying to get to build

applications for Android. Dr. Astrachan acknowledges that these comments "would have been available to any programmers who downloaded the source code from the Android website." (Id.) In other words, the comments were made available to the people who could derive value from them.

100. Dr. Astrachan expresses the opinion that the version of the comments that Google incorporated into Android were written by an Intel engineer as part of the Apache Harmony project, and suggests that this is the source from which Google obtained them. (Id. ¶ 174). Whether or not this is true, my understanding is that the source from which Google obtained the infringing comments is irrelevant.

IV. GOOGLE'S ANDROID PLATFORM DOES UNDERMINE JAVA COMPATIBILITY AND INTEROPERABILITY

101. In my opening report, I discussed one of Java's key innovations – "**write once, run anywhere**" – that resulted from a combination of interrelated design decisions.

102. The success of Java's "write once, run anywhere" promise is demonstrated by the unprecedented ecosystem built around Java, Java's widespread acceptance among platform vendors, and existence of millions of trained and capable application developers. (*See, e.g.*, GOOGLE-12-00003871 at 873 ("Carriers require Java in their terminal."); GOOGLE-01-00025376 at 419 ("Strategy: Leverage Java for its existing base of developers."); GOOGLE-02-00111218 at 218 ("Java has very little fragmentation, and it's adoptable. If we play our cards right, we can also leverage not only existing developers, but applications as well."); GOOGLE-01-00019511 at 512 ("Java is more accessable [sic] than C++ . . . . There is more standardization in tools and libraries."); GOOGLE-01-00019527 at 527 (Android is building a Java OS. We are making Java central to our solution because a) Java, as a programming language, has some advantages because it's the #1 choice for mobile development b) There exists

documentation and tools c) carriers require managed code d) Java has a suitable security framework.").)

103. The ability to run Java on many different hardware platforms was one of the key things that led Google to choose Java for the Android platform. Dr. Astrachan attempts to justify Google's copying of the 37 APIs at issue by claiming that it was necessary for "compatibility and interoperability." (Astrachan Report ¶ 130.) Google is not in a position to make this claim, however, because, although Google chose to make the Java APIs central to Android, it also deliberately chose to fragment them.

104. Google's Android developer website states: "Android includes a set of core libraries that provides most of the functionality available in the core libraries of the Java programming language." (Android Developer website, *available at* http://developer.android.com/guide/basics/what-is-android.html.) And Google apparently was aware of the resulting benefits. (*See, e.g.*, GOOGLE-04-00042610 at 611 ("Java solves a lot of the portability issues C++ has."); 7/7/2011 Swetland Dep. 33:19-20 ("I think the VM as a platform feature in Android was most important from a portability standpoint . . . ."); 4/5/2011 Rubin Dep. 91:19-23 ("There is no purpose of building an open platform other than to attract third-party developers to it. So anything that we would do to jeopardize the support of third-party developers would be bad for the success of the platform."); 4/5/2011 Rubin Dep. 24:21-25:2 ("Third-party developers contribute to the success of a platform by having their companies invest in the platform by basing their businesses on the platform. It was my intention to create an independent third-party developer ecosystem….").)

105. However, Google fragmented the Java Class Library APIs by adopting a subset and a superset of the Java Class Library APIs, each of which is discussed below. (The examples discussed below are taken from Java SE 5.0. I am aware that Oracle has

not asserted copyrights in relation to each package I discuss below; nevertheless, fragmentation is an independent and germane point.)

java.applet	java.awt.image.renderable	java.math	java.rmi.server	java.util.concurrent.locks	javax.imageio	javax.management.openmbean	javax.net.ssl	Javax.security.auth.login	javax.sql.rowset.serial	javax.swing.plaf.synth	javax.xml.datatype	org.omg.CORBA_2_3	org.omg.DynamicAny.DynAnyFactoryPackage	org.omg.PortableServer.POAPackage
java.awt	java.awt.print	java.net	java.security	java.util.jar	javax.imageio.event	javax.management.relation	javax.print	Javax.security.auth.spi	javax.sql.rowset.spi	javax.swing.table	javax.xml.namespace	org.omg.CORBA_2_3.portable	org.omg.DynamicAny.DynAnyPackage	org.omg.PortableServer.portable
java.awt.color	java.beans	java.nio	java.security.acl	java.util.logging	javax.imageio.metadata	javax.management.remote	javax.print.attribute	Javax.security.auth.x500	javax.swing	javax.swing.text	javax.xml.parsers	org.omg.CORBA.DynAnyPackage	org.omg.IOP	org.omg.PortableServer.ServantLocatorPackage
java.awt.datatransfer	java.beans.beancontext	java.nio.channels	java.security.cert	java.util.prefs	javax.imageio.plugins.bmp	javax.management.remote.rmi	javax.print.attribute.standard	Javax.security.cert	javax.swing.border	javax.swing.text.html	javax.xml.transform	org.omg.CORBA.ORBPackage	org.omg.IOP.CodecFactoryPackage	org.omg.SendingContext
java.awt.dnd	java.io	java.nio.channels.spi	java.security.interfaces	java.util.regex	javax.imageio.plugins.jpeg	javax.management.timer	javax.print.event	Javax.security.sasl	javax.swing.colorchooser	javax.swing.text.html.parser	javax.xml.transform.dom	org.omg.CORBA.portable	org.omg.IOP.CodecPackage	org.omg.stub.java.rmi
java.awt.event	java.lang	java.nio.charset	java.security.spec	java.util.zip	javax.imageio.spi	javax.naming	javax.rmi	Javax.sound.midi	javax.swing.event	javax.swing.text.rtf	javax.xml.transform.sax	org.omg.CORBA.TypeCodePackage	org.omg.Messaging	org.w3c.dom
java.awt.font	java.lang.annotation	java.nio.charset.spi	java.sql	javax.accessibility	javax.imageio.stream	javax.naming.directory	javax.rmi.CORBA	Javax.sound.midi.spi	javax.swing.filechooser	javax.swing.tree	javax.xml.transform.stream	org.omg.CosNaming	org.omg.PortableInterceptor	org.w3c.dom.bootstrap

java.awt.geom	java.lang.instrument	java.rmi	java.text	javax.activity	javax.management	javax.naming.event	javax.rmi.ssl	Javax.sound.sampled	javax.swing.plaf	javax.swing.undo	javax.xml.validation	org.omg.CosNaming.NamingContextExtPackage	org.omg.PortableInterceptor.ORBInitInfoPackage	org.w3c.dom.events
java.awt.im	java.lang.management	java.rmi.activation	java.util	javax.crypto	javax.management.loading	javax.naming.ldap	javax.security.auth	Javax.sound.sampled.spi	javax.swing.plaf.basic	javax.transaction	javax.xml.xpath	org.omg.CosNaming.NamingContextPackage	org.omg.PortableServer	org.w3c.dom.ls
java.awt.im.spi	java.lang.ref	java.rmi.dgc	java.util.concurrent	javax.crypto.interfaces	javax.management.modelmbean	javax.naming.spi	javax.security.auth.callback	Javax.sql	javax.swing.plaf.metal	javax.transaction.xa	org.ietf.jgss	org.omg.Dynamic	org.omg.PortableServer.CurrentPackage	org.xml.sax
java.awt.image	java.lang.reflect	java.rmi.registry	java.util.concurrent.atomic	javax.crypto.spec	javax.management.monitor	javax.net	javax.security.auth.kerberos	Javax.sql.rowset	javax.swing.plaf.multi	javax.xml	org.omg.CORBA	org.omg.DynamicAny	org.omg.PortableServer.POAManagerPackage	org.xml.sax.ext
														org.xml.sax.helpers

A. Google Fragmented the Class Library APIs by Adopting a Subset

106. The checkered diagram above illustrates Google's fragmentation of the Java platform by adopting selected portions of high-level packages encompassing numerous Java Class Library APIs in Android. In other words, Google adopted 51 of 130 high-level packages encompassing numerous Java Class Library APIs, as indicated by the highlighted boxes.

107. The problem for developers is that applications expected to run on a platform that supports the Java Class Library APIs may not run on Android even though Google has promised its Android application developers "a set of core libraries that provides most of the functionality available in the core libraries of the Java programming language." (Android Developer website, *available at* http://developer.android.com/guide/basics/what-is-android.html.) For example, Java application developers may want and expect to use java.awt.color, java.lang.management, java.rmi, and other examples but Google chose to not support these in Android. In fact, all of java.awt, except java.awt.font, and all of javax.swing, are missing, thereby eliminating the ability of application developers to build user interfaces using these Java Class Library APIs.

108. Stated differently, platform fragmentation – such as Android's fragmentation of the Java platform – breaks application interoperability. Applications built for the Java platform may not run on Android; applications built for Android may not run on devices built to support Java. The consequences are summarized as follows:

- Java application developers have to worry about writing and testing code for the variety of platforms for which they hope to supply applications. This means application developers also have to learn about the accompanying variety of execution environments and support tools, making the application development process more cumbersome.
- Device manufacturers have to worry about supporting the variant platforms.
- Consumers have to worry about whether they can download specific applications for specific devices.

Google's fragmentation of the Java platform significantly undermines Java's "write once, run anywhere" promise.

109. Google's fragmentation of the Java platform goes deeper than the top-level missing Java packages in Android. For example, even though Google makes java.awt.font available to Android application developers, Google does not support the java.awt.font package completely. In particular, Google fails to support the following classes within java.awt.font: FontRenderContext, GlyphJustificationInfo, GlyphMetrics, GlyphVector, GraphicAttribute, ImageGraphicAttribute, LayoutPath, LineBreakMeasurer, LineMetrics, ShapeGraphicAttribute, TextHitInfo, TextLayout, TextLayout.CaretPolicy, TextMeasurer, and TransformAttribute.

110. Google's fragmentation goes deeper still. In particular, Google changed java.lang.System.java in its Gingerbread Android release to turn off java.security.SecurityManager.java. By doing so, Google changed the behavior specified in the java.lang API and promised to application developers.

B. Google Fragmented the Java Class Library APIs By Supersetting

111. Google also chose to superset the Java SE Class Library APIs. For example, Google added:

javax.microedition.khronos.egl
javax.microedition.khronos.opengles

These packages are not part of Oracle's Java SE Class Library APIs.

112. Google has also added numerous packages outside of the Java namespace, such as:

android.accessibilityservice
android.accounts
android.animation
android.app
android.app.admin
android.app.backup
android.appwidget
android.bluetooth
android.content
android.content.pm
android.content.res
android.database
android.database.sqlite
android.drm

android.gesture
android.graphics
android.graphics.drawable
android.graphics.drawable.shapes
android.hardware
android.hardware.usb
android.inputmethodservice
android.location
android.media
android.media.audiofx
android.mtp
android.net
android.net.http
android.net.rtp
android.net.sip
android.net.wifi
android.nfc
android.nfc.tech
android.opengl
android.os
android.os.storage
android.preference
android.provider
android.renderscript
android.sax
android.service.wallpaper
android.speech
android.speech.tts
android.telephony
android.telephony.cdma
android.telephony.gsm
android.test
android.test.mock
android.test.suitebuilder
android.text
android.text.format
android.text.method
android.text.style
android.text.util
android.util
android.view
android.view.accessibility
android.view.animation
android.view.inputmethod
android.webkit
android.widget
dalvik.bytecode
dalvik.system

113. The consequences of supersetting are similarly damaging to "write once, run anywhere." Applications built for Android may not run on the Java platform or on devices built to support Java. Supersetting a platform also results in platform fragmentation – such as Android's fragmentation of the Java platform – and breaks application interoperability in the ways discussed above.

C. Oracle Licenses *Compatible* Implementations Of Java

114. If Google was interested in furthering interoperability and compatibility, it would have fully implemented Java in Android, not created a fragmented platform. Oracle offers to license developers to make their own independent implementations of the Java API specifications. For example, the earliest Java specifications bore this legend on their copyright pages:

> Sun Microsystems, Inc. (SUN) hereby grants to you a fully-paid, nonexclusive, nontransferable, perpetual, worldwide limited license (without the right to sublicense) under SUN's intellectual property rights that are essential to practice this specification. This license allows and is limited to the creation and distribution of clean room implementations of this specification that (i) include a complete implementation of the current version of this specification without subsetting or supersetting, (ii) implement all the interfaces and functionality of the standard java.* packages as defined by SUN, without subsetting or supersetting, (iii) do not add any additional packages, classes or methods to the java.* packages (iv) pass all test suites relating to the most recent published version of this specification that are available from SUN six (6) months prior to any beta release of the clean room implementation or upgrade thereto, (v) do not derive from SUN source code or binary materials, and (vi) do not include any SUN binary materials without an appropriate and separate license from SUN.

(See, e.g., Copyright Notice, The Java Language Specification (1st ed. 1996), *available at* http://java.sun.com/docs/books/jls/first_edition/html/jcopyright.doc.html.) Developers creating compatible, independent implementations can obtain a license from Oracle on these or similar terms. Google could have entered into a license such as this, but chose not to. Google's Android is not a complete implementation, without subsetting or supersetting. In addition, I understand

that Android has not passed the relevant Java compatibility test suites that would check for and ensure total compatibility and Google has not attempted to complete this process. As Android founder and chief Andy Rubin wrote, "They [Sun] will never certify our VM, or put another way, I will not submit Dalvik for certification." (GOOGLE-01-00026813.) Of course, there is no sense in submitting Android for certification — because it is intentionally incompatible, it would fail.

115. Oracle's licensing policy is based on the idea that fragmentation and forking are serious threats to the fulfillment of Java's write once, run anywhere promise. Google recognizes the importance of standardization and the impact of fragmentation. Even as Google has broken with Java compatibility, it enters into "anti-fragmentation" agreements and mandates API compatibility to avoid fragmentation and forking of its own platform. (*See, e.g.*, Google's Android 2.2 to Compatibility Definition Document, *available at* http://static.googleusercontent.com/external_content/untrusted_dlcp/source.android.com/en/us/compatibility/2.2/android-2.2-cdd.pdf):

"3.1. Managed API Compatibility
The managed (Dalvik-based) execution environment is the primary vehicle for Android applications. The Android application programming interface (API) is the set of Android platform interfaces exposed to applications running in the managed VM environment. Device implementations MUST provide complete implementations, including all documented behaviors, of any documented API exposed by the Android 2.1 SDK [Resources, 4].

Device implementations MUST NOT omit any managed APIs, alter API interfaces or signatures, deviate from the documented behavior, or include no-ops, except where specifically allowed by this Compatibility Definition.

…

3.6 API Namespaces

Android follows the package and class namespace conventions defined by the Java programming language. To ensure compatibility with third-party applications, device implementers MUST NOT make any prohibited modifications (see below) to these package namespaces:

• java.*
• javax.*
• sun.*
• android.*
• com.android.*

Prohibited modifications include:

Device implementations MUST NOT modify the publicly exposed APIs on the Android platform by changing any method or class signatures, or by removing classes or class fields.

Device implementers MAY modify the underlying implementation of the APIs, but such modifications MUST NOT impact the stated behavior and Java-language signature of any publicly exposed APIs.

Device implementers MUST NOT add any publicly exposed elements (such as classes or interfaces, or fields or methods to existing classes or interfaces) to the APIs above."

116. I reserve the right to supplement or amend this report, if additional facts and information that affect my opinions become available. In particular, I understand that fact discovery closes on August 15, 2011. I have been informed that Google has yet to complete production of information that may affect my infringement analysis. Some information about various versions of the accused software, systems, and applications is not publicly available. For example, I have not been able to examine the source code for Honeycomb, and I understand that

Google plans to release future versions of Android software. None of these have been made available for my review. As such, my investigation into the specifics and extent of Google's infringement is ongoing. My report is based on the materials that have been available to me up to the date of this report.

117. In arriving at my opinions provided in this report, I have considered the additional materials referenced in the report, as well as Dr. Astrachan's opening report and attached exhibits, and some of the materials cited therein. I may provide further exhibits to be used as a summary of, or support for, my opinions.

Dated August 12, 2011

John C. Mitchell

Exhibit Copyright-Opposition-A

Interface and Class Hierarchy for java.util package

Class Hierarchy

- java.lang.**Object**
  - java.util.**AbstractCollection**<E> (implements java.util.Collection<E>)
    - java.util.**AbstractList**<E> (implements java.util.List<E>)
      - java.util.**AbstractSequentialList**<E>
        - java.util.**LinkedList**<E> (implements java.lang.Cloneable, java.util.List<E>, java.util.Queue<E>, java.io.Serializable)
      - java.util.**ArrayList**<E> (implements java.lang.Cloneable, java.util.List<E>, java.util.RandomAccess, java.io.Serializable)
      - java.util.**Vector**<E> (implements java.lang.Cloneable, java.util.List<E>, java.util.RandomAccess, java.io.Serializable)
        - java.util.**Stack**<E>
    - java.util.**AbstractQueue**<E> (implements java.util.Queue<E>)
      - java.util.**PriorityQueue**<E> (implements java.io.Serializable)
    - java.util.**AbstractSet**<E> (implements java.util.Set<E>)
      - java.util.**EnumSet**<E> (implements java.lang.Cloneable, java.io.Serializable)
      - java.util.**HashSet**<E> (implements java.lang.Cloneable, java.io.Serializable, java.util.Set<E>)
        - java.util.**LinkedHashSet**<E> (implements java.lang.Cloneable, java.io.Serializable, java.util.Set<E>)
      - java.util.**TreeSet**<E> (implements java.lang.Cloneable, java.io.Serializable, java.util.SortedSet<E>)
  - java.util.**AbstractMap**<K,V> (implements java.util.Map<K,V>)
    - java.util.**EnumMap**<K,V> (implements java.lang.Cloneable, java.io.Serializable)
    - java.util.**HashMap**<K,V> (implements java.lang.Cloneable, java.util.Map<K,V>, java.io.Serializable)
      - java.util.**LinkedHashMap**<K,V> (implements java.util.Map<K,V>)
    - java.util.**IdentityHashMap**<K,V> (implements java.lang.Cloneable, java.util.Map<K,V>, java.io.Serializable)
    - java.util.**TreeMap**<K,V> (implements java.lang.Cloneable, java.io.Serializable, java.util.SortedMap<K,V>)
    - java.util.**WeakHashMap**<K,V> (implements java.util.Map<K,V>)
  - java.util.**Arrays**
  - java.util.**BitSet** (implements java.lang.Cloneable, java.io.Serializable)
  - java.util.**Calendar** (implements java.lang.Cloneable, java.lang.Comparable<T>, java.io.Serializable)
    - java.util.**GregorianCalendar**
  - java.util.**Collections**
  - java.util.**Currency** (implements java.io.Serializable)
  - java.util.**Date** (implements java.lang.Cloneable, java.lang.Comparable<T>, java.io.Serializable)
  - java.util.**Dictionary**<K,V>
    - java.util.**Hashtable**<K,V> (implements java.lang.Cloneable, java.util.Map<K,V>, java.io.Serializable)
      - java.util.**Properties**
  - java.util.**EventListenerProxy** (implements java.util.EventListener)
  - java.util.**EventObject** (implements java.io.Serializable)
  - java.util.**FormattableFlags**
  - java.util.**Formatter** (implements java.io.Closeable, java.io.Flushable)
  - java.util.**Locale** (implements java.lang.Cloneable, java.io.Serializable)

- java.util.**Observable**
- java.security.**Permission** (implements java.security.Guard, java.io.Serializable)
  - java.security.**BasicPermission** (implements java.io.Serializable)
    - java.util.**PropertyPermission**
- java.util.**Random** (implements java.io.Serializable)
- java.util.**ResourceBundle**
  - java.util.**ListResourceBundle**
  - java.util.**PropertyResourceBundle**
- java.util.**Scanner** (implements java.util.Iterator<E>)
- java.util.**StringTokenizer** (implements java.util.Enumeration<E>)
- java.lang.**Throwable** (implements java.io.Serializable)
  - java.lang.**Exception**
    - java.io.**IOException**
      - java.util.**InvalidPropertiesFormatException**
    - java.lang.**RuntimeException**
      - java.util.**ConcurrentModificationException**
      - java.util.**EmptyStackException**
      - java.lang.**IllegalArgumentException**
        - java.util.**IllegalFormatException**
          - java.util.**DuplicateFormatFlagsException**
          - java.util.**FormatFlagsConversionMismatchException**
          - java.util.**IllegalFormatCodePointException**
          - java.util.**IllegalFormatConversionException**
          - java.util.**IllegalFormatFlagsException**
          - java.util.**IllegalFormatPrecisionException**
          - java.util.**IllegalFormatWidthException**
          - java.util.**MissingFormatArgumentException**
          - java.util.**MissingFormatWidthException**
          - java.util.**UnknownFormatConversionException**
          - java.util.**UnknownFormatFlagsException**
      - java.lang.**IllegalStateException**
        - java.util.**FormatterClosedException**
      - java.util.**MissingResourceException**
      - java.util.**NoSuchElementException**
        - java.util.**InputMismatchException**
    - java.util.**TooManyListenersException**
- java.util.**Timer**
- java.util.**TimerTask** (implements java.lang.Runnable)
- java.util.**TimeZone** (implements java.lang.Cloneable, java.io.Serializable)
  - java.util.**SimpleTimeZone**
- java.util.**UUID** (implements java.lang.Comparable<T>, java.io.Serializable)

Interface Hierarchy

- java.util.**Comparator**<T>
- java.util.**Enumeration**<E>
- java.util.**EventListener**
- java.util.**Formattable**
- java.lang.**Iterable**<T>
  - java.util.**Collection**<E>
    - java.util.**List**<E>
    - java.util.**Queue**<E>
    - java.util.**Set**<E>

- java.util.**SortedSet**<E>
- java.util.**Iterator**<E>
  - java.util.**ListIterator**<E>
- java.util.**Map**<K,V>
  - java.util.**SortedMap**<K,V>
- java.util.**Map.Entry**<K,V>
- java.util.**Observer**
- java.util.**RandomAccess**

Enum Hierarchy

- java.lang.**Object**
  - java.lang.**Enum**<E> (implements java.lang.Comparable<T>, java.io.Serializable)
    - java.util.**Formatter.BigDecimalLayoutForm**