No. 18-956

IN THE

Supreme Court of the United States

GOOGLE LLC,

Petitioner,

v.

ORACLE AMERICA, INC.,

Respondent.

On Writ of Certiorari
to the United States Court of Appeals
for the Federal Circuit

BRIEF AMICI CURIAE
OF XX COMPUTER SCIENTISTS
IN SUPPORT OF PETITIONER

Phillip R. Malone
Counsel of Record
JUELSGAARD INTELLECTUAL
PROPERTY AND
INNOVATION CLINIC
MILLS LEGAL CLINIC AT
STANFORD LAW SCHOOL
559 Nathan Abbott Way
Stanford, CA 94305
(650) 725-6369
pmalone@stanford.edu

Counsel for Amici Curiae

TABLE OF CONTENTS

TABLE OF AUTHORITIES

INTEREST OF *AMICI CURIAE*

Amici are 78 computer scientists, engineers, and professors who are pioneering and influential figures in the computer industry.[1] Amici include the architects of iconic computers from the mainframe era to the microcomputer era, including the IBM S/360 and the Apple II; languages such as AppleScript, AWK, C, C#, C++, Delphi, Go, Haskell, PL/I, Python, RenderMan, Scala, Scheme, Standard ML, Smalltalk, and TypeScript; and operating systems such as MS-DOS and Unix.[2] Amici are responsible for key advances in the field, including in computer graphics,

---

[1] Petitioner granted blanket consent for the filing of this brief; Respondent declined to consent. No counsel for a party authored this brief in whole or in part, and no party or counsel for a party made a monetary contribution intended to fund its preparation or submission. No person, other than *amici* or their counsel, made a monetary contribution to the preparation or submission of this brief.

[2] Amici's biographies are attached as Appendix A. Amici sign this brief on their own behalf and not on behalf of the companies or organizations with which they are affiliated; those affiliations are for identification purposes only. Amici represent a cross section of the world's most distinguished computer scientists and engineers. As such, the 78 amici include five who are presently Google employees (indicated by * next to their names); two who receive some support from Google (indicated by **); two who testified as unpaid fact witnesses at trial in this case (indicated by †); and one who was retained as an expert by Google but did not testify (indicated by ‡). Each of these amici signs this brief based on their personal experience and beliefs as individual computer scientists whose work in the field long preceded their affiliation with Google or their participation in this case. None sign on behalf of Google or at Google's request.

computer animation, computer system architecture, cloud computing, algorithms, public key cryptography, the theory of computation, object-oriented programming, relational databases, design patterns, virtual reality, the spreadsheet, and the Internet. Amici wrote the standard college textbooks in areas including artificial intelligence, algorithms, computer architecture, computer graphics, computer security, data structures, functional programming, Java programming, operating systems, software engineering, and the theory of programming languages.

Amici have been widely recognized for their achievements. They include at least 12 Association for Computing Machinery (ACM) Turing Award winners (computer science's most prestigious award); 24 ACM Fellows; 11 Institute of Electrical and Electronics Engineers (IEEE) Fellows; 14 American Academy of Arts and Sciences Fellows; 6 National Academy of Sciences Members; 24 National Academy of Engineering Members; 5 National Medal of Technology recipients; and numerous professors at many of the world's leading universities.

As computer scientists, amici have relied on reimplementing interfaces to create fundamental software. They join this brief because they believe, based on their extensive experience with and knowledge of computer software and programming, that the decisions below threaten to upend decades of settled expectations across the computer industry and chill continued innovation in the field.

SUMMARY OF ARGUMENT

The decisions of the Federal Circuit below are wrong and threaten significant disruption if allowed to stand. They undermine a fundamental process—software interface reimplementation—that has spurred historic innovation across the software industry for decades.

Software *interfaces*, including those embodied in the Java Application Programming Interface (API) at issue here, are purely functional systems or methods of operating a computer program or platform. They are not computer programs themselves. Interfaces merely describe what functional tasks a computer program will perform without specifying how it does so. The Java API's functional interfaces, called declarations, are written using the Java programming language, which mandates each declaration's precise form.

In contrast, *implementations* provide the actual step-by-step instructions to perform each task included in an interface. Sun implemented the Java API for desktop computers. Google reimplemented—or wrote its own original implementation of—the Java API when it created the Android platform for smartphones and tablets. Android was highly transformative: It enabled programs written in the Java programming language to successfully run on smartphones and tablets for the first time. Doing so required Google to make significant additions to the Java API to handle mobile-specific features, like touchscreen inputs.

Android also provided interoperability with Java: Programmers could use their preexisting knowledge to simultaneously write Java programs for both desktops and smartphones. Reimplementing the Java API was the only way to make Android interoperable with Java. Reimplementation requires duplicating an interface's declarations and organizational scheme—its structure, sequence, and organization (SSO). Had Android changed the Java API's declarations or SSO, programmers would have been forced to write different software for desktops and smartphones, eliminating one of Android's most significant benefits.

Google's reimplementation of an existing interface was not unusual. Reimplementing software interfaces is a long-standing, ubiquitous practice that has been essential to realizing fundamental advances in computing. It unleashed the personal computer revolution, created popular operating systems and programming languages, and established the foundation upon which the Internet and cloud computing depend. It continues to increase consumer choice, lower prices, and foster compatibility between programs. Free reimplementation of software interfaces has long been, and remains, essential for innovation and competition in software.

The Court should reverse the decisions below to preserve software interfaces as uncopyrightable and prevent copyright from stifling innovation in software.

ARGUMENT

I. The Decisions Below Reflect the Federal Circuit's Fundamental Misunderstanding of How Interfaces Differ from Programs

The decisions below extend copyright protection to software interfaces—including the Java API—by erroneously equating them with computer programs. Asserting that software interfaces are simply a type of computer program, all of which are "by definition functional," the Federal Circuit misapplied general Ninth Circuit law recognizing computer programs as copyrightable. *See Oracle Am., Inc. v. Google Inc.* (Copyright II), 750 F.3d 1339, 1367 (Fed. Cir. 2014). But software interfaces are not computer programs, and no party argues that "one can copy line-for-line someone else's copyrighted computer program." *Oracle Am., Inc. v. Google Inc.* (Copyright I), 872 F. Supp. 2d 974, 987 (N.D. Cal. 2012).

The Federal Circuit's conclusory review fails to appreciate the district court's reasoned—and correct—recognition of software interfaces as uncopyrightable under 17 U.S.C. § 102(b) and the merger doctrine. *See Copyright I*, 872 F. Supp. 2d at 998-1000. The Federal Circuit compounded its error by overturning a jury finding of fair use and holding that Google's creation of Android was not fair use as a matter of law. *See Oracle Am., Inc. v. Google LLC* (Fair Use II), 886 F.3d 1179, 1185-86 (Fed. Cir. 2018).

Amici join Google's arguments that software interfaces cannot be copyrighted under either § 102(b) or the merger doctrine, and that in any event, the jury could reasonably have found that Google's creation of Android was fair use. Brief for Petitioner at 19, 34, *Google LLC v. Oracle Am., Inc.*, No. 18-956 (Jan. 6, 2020). In support of those arguments, amici emphasize that software interfaces correspond to functional ideas, that Google had to duplicate the Java API's declarations exactly to provide interoperability between Android and Java, and that Android was a transformative achievement that successfully introduced Java to smartphones for the first time.

A. Software Interfaces Specify What a Program Does, Not How It Does So

A software interface specifies the set of commands used to operate a computer program or system. Each command defines one functional task a program must accomplish, such as finding the maximum of two numbers, sorting a list of numbers, or displaying text on the screen.

Each command in an interface includes its name, inputs, and outputs. Together, these comprise the command's "declaration." The declaration for a command to find the maximum of two numbers, for example, would include the name "max," two numbers as inputs, and one number—the maximum—as output. Declarations are purely functional: They specify *what* a computer program or system needs to do without specifying *how* it does so. By themselves, declarations do not instruct a computer to do anything.

In contrast, an interface's *implementation* is the actual "set of statements or instructions to be used directly or indirectly in a computer in order to bring about a certain result," namely, carrying out the tasks specified by its declarations. 17 U.S.C. § 101 (defining "computer program"). The same declaration can be implemented in various ways to accomplish the same task. Some implementations prioritize speed, others memory use. So long as an implementation carries out the specified task, it is valid. While the "specification is the *idea*," the "implementation is the *expression*." *Copyright I*, 872 F. Supp. 2d at 998 (emphasis in original).

Because real-world software interfaces can include thousands of declarations, programmers group related declarations into their own "folders," just as everyday computer users group related files into folders on their desktop. The courts and parties have referred to this organizational scheme throughout this litigation as the interface's structure, sequence, and organization (SSO).

i. Declarations specify the individual tasks a program must perform

To better understand the relationship between an interface's declarations, implementations, and SSO, consider the `sort` declaration in the Java API.[3] In English, this declaration would read, "Given a list of numbers, sort them in ascending order." To express this functional requirement in terms a computer can understand, a programmer would write the following declaration in the Java language[4]:

```
public static void sort(int[] a)
```

Before explaining each component of this declaration, we emphasize that this line does not instruct the computer to do anything. If a programmer attempted to run this "program," nothing would happen because there are no instructions to run. The

[3] `Courier` font denotes Java keywords and declarations.

[4] The Java language is one part of the Java platform (J2SE), which also includes the API and API implementations (the latter are also called "libraries"). While the boundary between the language and the API is indefinite, the language is generally responsible for defining the syntax and keywords programmers use to write software. Only the API is at issue here. *See Copyright I*, 872 F. Supp. 2d at 978.

line simply indicates that this declaration's implementation will include a command, which Java calls a "method," for sorting numbers. The Java language requires almost every word in this declaration. A programmer *must* type those words exactly as they appear above, including the same capitalization, punctuation, and order. Otherwise, the declaration will cause an error or specify a method with different functionality, like sorting words instead of numbers.

The word `public` is a Java language keyword that enables other programs to use `sort` once it has been implemented (other keywords, like `private`, restrict other programs' access to a method). Similarly, the Java language requires `static` for `sort` to work as expected.[5] The void keyword means that the method does not have any output; rather than output a sorted copy of the list, `sort` simply rearranges the given list of numbers. Finally, the parentheses enclose the method inputs. Here, the only input is the list of integers to be sorted—designated by the Java keyword `int[]`.

In contrast, only two words in the declaration leave the programmer any choice, and both are names. The first is `sort` itself. This word descriptively names the method based on the task its implementation will perform. While it would be possible to use a

[5] The Java language primarily views programs in terms of interactions among "objects" representing the program's data. Related objects are members of the same "class." Adding the `static` keyword to a method declaration allows that method to be called on all objects of a class even if the method could not be added to the class directly, as is the case here.

synonym—perhaps “arrange” or “order”—for the same method, few names are as intuitive as `sort` to describe the task this method’s implementation will perform. Particularly short and intuitive names for common operations like `sort` become customary terms of art used across interfaces.[6] Deliberate naming enhances an interface’s readability and minimizes errors, especially when, as is typically the case, that interface is designed and used by different programmers.

Similarly, `a` names the input “array,” or list, of numbers to be sorted. Just as with `sort`, the programmer designing the interface chooses the input’s name. Other options could be “array,” “numbers,” or “list.” But just as with `sort`, the universe of reasonable names is small and further restricted by linguistic convention. While software interface designers have some choice for naming methods and inputs, the method’s function, name length, and clarity constrain their choice. Particularly for programming language interfaces, which define the most basic commands used across programs, there are few practical options for naming declarations that satisfy these constraints.

[6] As of January 2020, eight of the top ten most used programming languages (Java, Python, C++, C#, Visual Basic .NET, JavaScript, PHP, and Swift) include a command called `sort` to arrange a list in ascending order. *See* TIOBE Index for January 2020, TIOBE (last visited Jan. 5, 2020), https://www.tiobe.com/tiobe-index.

ii. Implementations provide the step-by-step instructions to perform the tasks declarations specify

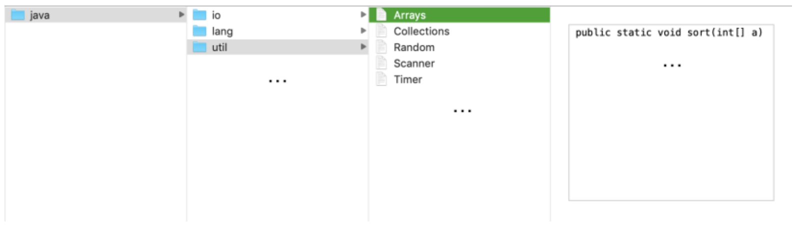
Once a software interface has been designed, programmers can supply implementations to carry out the tasks specified by its declarations. Google, for example, wrote its own implementations for the Java API's declarations. Implementations take the inputs listed in declarations and manipulate them to produce the correct output. While the syntax of the programming language dictates the form of each declaration, implementations are open-ended and can be thousands of lines long. Naïve implementations can be prohibitively slow or use excessive amounts of memory. In contrast, clever implementations can run quickly enough to make formerly unfeasible operations practical or conserve enough memory to allow programs to run on entirely new hardware—such as phones, tablets, televisions, or even home thermostats—that have far less memory available than desktop computers.

Computer scientists have evaluated dozens of implementations for `sort`. One of the simplest implementations is "selection sort." Given a list of numbers, a selection sort implementation starts at the beginning of the list and walks through number by number, keeping a running tally of the smallest number it has found. Once it reaches the end of the list, it swaps the smallest number with the number at the beginning of the list. Then, the program searches through the remainder of the list a second time, this time looking for the second smallest number to swap into the second position. This process repeats until the program has swapped every number into its correct position. Unfortunately, this implementation is prohibitively slow for large lists of numbers.

More sophisticated implementations for `sort`, like "quicksort" or "mergesort," can sort even large lists efficiently. With modern data sets comprising hundreds of millions or even billions of numbers, names, or images, inefficient sorting implementations like selection sort make entire categories of programs impossible to use. Because different devices have different constraints, software engineers devote considerable effort to choosing the best implementation to meet their specific needs. Their choice could mean the difference between the success of two competing pieces of software.

iii. SSOs establish how software interfaces group related declarations

Because interfaces can include tens of thousands of declarations, their designers organize related declarations in the same way users organize related files into folders on their desktop. In fact, Java's designers organized the Java API's files in exactly this way. *See* Figure 1.



```
public static void sort(int[] a)
...
```

Figure 1

Java's API is organized in three tiers: packages, classes, and methods. Packages correspond to folders, classes to files, and declarations to individual lines in a file. The full file path for `sort`, for example, is `java.util.Arrays.sort`. The overall folder for the interface is named `java`, while `util`, short for utility, is the name of the package, or subfolder, containing the API's various general-purpose classes. One such class, `Arrays`, is a file that contains methods for manipulating lists of objects, like numbers. One of the lines in `Arrays` is the declaration given above for `sort`.

Programmers who reimplement, i.e., provide their own implementation for, an interface must maintain its SSO. Failure to do so will necessarily result in incompatibility. Just as users must know how to navigate to their saved documents, programmers using a software interface must specify the path for each declaration they use, like `sort`, so that the computer knows where to find the corresponding implementation. Telling a person to click on "My Documents," then on a folder called "Receipts," and finally on a file called "Sofa" to find how much their sofa cost is just like a program navigating through the Java API to a package called `util` and opening a class called `Arrays` to find the implementation for the `sort` method.

Changing this standard organizational scheme would prevent a person or a program from locating the file or implementation they need, rendering the interface specification incompatible. Thus, while interface designers have some choice in naming their method declarations and inputs, programmers who are reimplementing an existing interface, like Google did with the Java API, *must* use the same standard names and structure to achieve interoperability.

B. Google Wrote Its Own Implementation of the Java API to Promote Interoperability and Transform Java to Run on Smartphones

Google created the Android platform to promote interoperability and enable Java to run on an entirely new class of devices: smartphones. This required Google to reimplement the Java API: It duplicated the Java API's declarations and SSO but wrote its own implementations. *See Copyright I*, 872 F. Supp. 2d at 978. It would have been impossible for Google to make Android interoperable, or compatible, with Java without reimplementing the Java API.[7] In this context, making software interoperable *means* reimplementing a software interface.

In both of its opinions, the Federal Circuit questioned Google's claim that Android reimplemented the Java API to promote interoperability with Java because programs written for Android are not fully compatible with Java. *Fair Use II*, 886 F.3d at 1206 n.11 (finding evidence "unrebutted" that "Google designed Android to be incompatible with the Java platform"); *see also Copyright II*, 750 F.3d at 1371 (finding "Google's interoperability argument confusing"). But complete

---

[7] We follow convention in using the terms "interoperability" and "compatibility" interchangeably. Oracle's requirement that companies obtain a Java Compatibility Kit (JCK) license to demonstrate "compatibility" is merely a licensing scheme, not a technical necessity.

compatibility is not necessary, or even desirable, to promote interoperability in software development.

Because of its longevity, Java, and almost every other computer system, must remain backwards-compatible. Any program written in earlier versions of Java must also run on later versions, or programmers would be unable to make cumulative improvements and the software ecosystem would break down. However, this also means that inefficient or outdated software survives several generations of software development solely to maintain compatibility.

To avoid this problem, Google selectively reimplemented portions of the Java API for Android to eliminate functionality that was obsolete or inappropriate for smartphones, like using a mouse. *See Copyright I*, 872 F. Supp. 2d at 978. Rather than copy Sun's implementations, Google was careful to write its own implementations to carry out the tasks the Java API's declarations specify. Google's decision empowered software developers to write Java programs that run equally well on both desktops and smartphones. *See Oracle Am., Inc. v. Google Inc.* (Fair Use I), 2016 WL 3181206, at *10 (N.D. Cal. 2016).

Android was highly transformative. Creating Android required Google to significantly expand Java's API in novel ways to account for external features and constraints unique to the smartphone context: built-in GPS tracking, limited battery life and memory, fluctuating network connections, and an entirely new user interface based on touchscreen gestures. *See Fair Use I*, 2016 WL 3181206, at *9. In contrast, "Sun and Oracle never successfully developed its own smartphone platform using Java technology." *Copyright I*, 872 F. Supp. 2d at 978. While

Sun did release Java ME to run Java on feature phones, these devices are far less sophisticated than modern smartphones. Moreover, Java ME did not support the entire Java language, omitting basic features like numbers with decimal points. Nor did Java ME support key Java API features like the Java Collections Framework, which is part of `java.util`, a package necessary "to make any worthwhile use of the [Java] language." *Copyright II*, 750 F.3d at 1349. Thus, Java ME was far *less* compatible with standard Java than Android, and Java ME's failure to include such core functionality only underscores how transformative Android was.

Google's significant augmentations to Java's API introduced Java to an entirely new Android platform that, with 2.5 billion active devices, is "by far" the most-used operating system in the world. Liam Tung, *Bigger than Windows, Bigger than iOS: Google Now Has 2.5 Billion Active Android Devices*, ZD Net (May 8, 2019), https://www.zdnet.com/article/bigger-than-windows-bigger-than-ios-google-now-has-2-5-billion-active-android-devices-after-10-years. Programmers using only the reimplemented packages can write programs for desktops and smartphones using the same familiar instructions. Additionally, because Java and Android are both open source (meaning anyone can read and contribute to their implementations), Google's focus on interoperability has enabled outside programmers, including many amici, to contribute improvements to both platforms simultaneously.

Contrary to the Federal Circuit's assertion that there was no evidence of programs that rely only on Google's reimplemented packages, or that "[no] such program would be useful," *Copyright II*, 750 F.3d at

1371 n.15, Java and Android form parts of a broad and largely compatible ecosystem that drastically simplifies writing software for desktops and smartphones. Many important programs, including Guava (which provides efficient implementations of numerous core functions), Gradle and Maven (which serve as project management tools), and JUnit (which helps test the output of a program's subcomponents), are routinely used with programs developed using Java and Android.

Android revitalized this ecosystem, inspiring renewed innovation and collaboration among programmers. Sun's CEO publicly congratulated Google upon Android's release on his official company blog and expressed support for Android. *See* Brief of Defendant-Appellee/Cross-Appellant Google Inc. at 17-18, *Fair Use II*, 886 F.3d 1179 (Docket No. 17-1118), 2017 WL 2305681. Sun's CEO also emailed Google's CEO directly to offer his congratulations on Android's success and to suggest further improvements. *See id.* at 18-19. After acquiring Sun, even Oracle initially praised Google for expanding Java to new devices. *See id.* at 19.

Sun had always promoted the Java API, along with the Java language, as free and open for all to use. *See id.* at 9-10. Many amici, along with instructors at high schools and colleges across the country, decided to teach Java in introductory programming courses precisely because of its free availability. Assertions that the Java API might be copyrightable only emerged after Oracle acquired Sun in 2010. While Oracle does not dispute that the Java language is free and open for all to use, it asserts a copyright interest in the Java API. *Copyright I*, 872 F. Supp. 2d at 978.

Even then, Oracle concedes that at least sixty-two classes, spread across three Java API packages, are necessary for the Java language to work. *Fair Use I*, 2016 WL 3181206, at *5.

As professors, textbook authors, and industry leaders, amici have broad experience with both teaching and using the Java language and do not consider it to be fully separable from the Java API. In fact, for *any* programming language, the core API is integral to the language. Thus, amici agree with the district court that "there is no bright line" between the Java language and API. *Copyright I*, 872 F. Supp. 2d at 982. Introductory Java textbooks typically introduce the Java API at the outset, and amici know of *no* Java textbook that teaches the language without covering the API. A Java program which failed to use the Java API would hardly be recognizable: The API is part of what makes the Java language, Java. Indeed, Oracle's own online tutorials consider portions of the Java API—including packages like `java.util.regex` that it accuses Google of infringing—"essential to most programmers" for programming in Java. *Trail: Essential Classes (The Java^{TM} Tutorials)*, Oracle (last visited Jan. 5, 2020), https://docs.oracle.com/javase/tutorial/essential/index.html.

II. The Decisions Below Upend Decades of Settled Expectations and Threaten Future Innovation in Software

Software interfaces are essential to innovation. For decades, programmers have relied upon reimplementing interfaces to create fundamentally transformative technologies. Reimplementing software interfaces also promotes innovation by countering network effects and lock-in effects that otherwise inhibit competition. This Court should reverse the decisions below to preserve software interface reimplementation and the vitality of the software industry.

A. The Computer Industry Has Long Relied on Freely Reimplementing Software Interfaces to Foster Innovation and Competition

Oracle's attempt to assert copyright in the Java API is historically anomalous and jeopardizes the unparalleled innovation and competition that continue to flourish across the computer industry. The first practical description of an API appeared in 1951, *see generally* Maurice V. Wilkes, David J. Wheeler & Stanley Gill, *The Preparation of Programs for an Electronic Digital Computer* (1951), and the specific phrase "application programming interface" dates to at least 1968, *see* Ira W. Cotton & Frank S. Greatorex, Jr., Data Structures and Techniques for Remote Computer Graphics, Am. Fed'n Info. Processing Soc'ys Fall Joint Computer Conf. 533, 534-35 (1968). Programmers have freely reimplemented software interfaces throughout the ensuing decades. By creating standard specifications for computer programs to communicate with each other, uncopyrightable software interfaces have promoted competition in personal computing and led to the rise of popular operating systems, programming languages, the Internet, and cloud computing. Google's reimplementation of the Java API fits squarely within this tradition of innovation and competition.

i. Interface reimplementation unleashed the personal computer revolution

Reimplementing software interfaces made personal computing commonplace. IBM released its first home computer in 1981. Software companies developed an ecosystem of products to run on IBM's machine, including the popular spreadsheet program Lotus 1-2-3 co-created by amicus Mitchell Kapor. To run these programs, however, users had to purchase IBM's PC because the programs required full compatibility with IBM's basic input-output system (BIOS) responsible for starting the operating system and initializing the computer's hardware when turned on. To compete with IBM, programmers like amicus Tom Jennings at software company Phoenix, along with those at computer manufacturers, like Compaq, reimplemented the BIOS API, including its SSO, to enable users to run their favorite IBM-compatible software on competing machines.

Thus, reimplementing the BIOS API resulted in the manufacture and sale of faster, cheaper, and compatible alternatives to IBM's PC that could run important programs like DOS, the operating system responsible for Microsoft's early success. If copyright had prevented competitors from reimplementing IBM's BIOS API and making IBM-compatible PCs, companies like Microsoft would never have been able to revolutionize personal computing.

ii. Interface reimplementation created the world's most ubiquitous operating systems

Operating systems, the fundamental programs responsible for managing all of a computer's hardware and software resources, depend on software interface reimplementation. The first modern operating system, Unix, was developed by amici Ken Thompson and Brian Kernighan and others at AT&T Bell Labs and released in 1969. AT&T licensed Unix's source code to academic institutions for a nominal fee, leading to widespread adoption. Because commercial licenses from AT&T were costly and restrictive, and because hardware evolutions outpaced AT&T's Unix API, programmers reimplemented and extended the API themselves.

Today, nearly 70% of websites run on Unix-based operating systems, including the popular open source operating system Linux. *See Usage Statistics of Unix for Websites*, W3Techs (Jan. 6, 2020), https://w3techs.com/technologies/details/os-unix. Linux alone runs nearly 35% of Internet servers and the 500 fastest supercomputers in the world. *See id*; Steven J. Vaughan-Nichols, *Linux Totally Dominates Supercomputers*, ZDNet (Nov. 14, 2017, 12:04 PM PST), http://www.zdnet.com/article/linux-totally-dominates-supercomputers. Android's operating system, the most popular in the world, *see* Tung, is itself built atop Linux. And Apple, co-founded by amicus Steve Wozniak, also reimplemented the Unix API for its desktop OS X and mobile iOS operating systems. Programmers' ability to reimplement the Unix API established a standardized design for the fundamental program running on any computer: its operating system.

iii. Interface reimplementation fueled widespread adoption of popular programming languages

One of the most influential programming languages, C, became widespread due to the relative ease of reimplementing its API to enable C programs to run on different hardware. Open source enthusiasts reimplemented a version of C compatible with Linux, and industry leaders like Microsoft and Google reimplemented C for their own products. Other popular programming languages like C++, created by amicus Bjarne Stroustrup, also proliferated due in part to reimplementations of their APIs.

Similarly, Sun reimplemented existing APIs as part of the Java platform. Java reimplemented C's math API, which includes methods for calculating a variety of mathematical functions. While at Sun, amicus Joshua Bloch oversaw Sun's reimplementation of the Perl programming language's regular expression API, which allows sophisticated text searches and alterations. Oracle's attempt to copyright Java's API and hold Google liable for infringement of the resulting `java.util.regex` API ignores Java's own history of API reimplementation.

iv. Interface reimplementation enables computer networks, including the Internet, to function

The Internet relies on programmers' ability to reimplement standardized interfaces to transmit data.

Copyrighting those interfaces would defeat the Internet's goal of creating a global network of interconnected computers. In 1983, the Berkeley Systems Research Group released the Berkeley Systems Distribution (BSD) sockets API. Sockets control the endpoints for any communication over the Internet. Because the BSD sockets API was not copyrighted, it became widely adopted: Every major operating system reimplemented it to enable Internet communication. Thus, programmers can write standardized software compatible across computers to manage Internet connectivity.

v. Interface reimplementation is fundamental to cloud computing

Finally, reimplementing software interfaces has been, and continues to be, fundamental to cloud computing – the XXX. With cloud computing, developers can rent powerful computer hardware to run resource-intensive computations, like machine-learning algorithms, without having to purchase and manage expensive hardware themselves. Amazon's Web Services (AWS) API serves as the de facto industry standard for cloud computing. AWS itself reimplemented IBM's BIOS API, enabling familiar BIOS commands to run on Amazon's servers. AWS therefore allows programmers to write programs as if they were running on a standard PC rather than learn commands unique to Amazon.

Major competitors, including Microsoft, Google, and Oracle, have in turn adopted AWS's API.[8] Rather than compete on the API's design, cloud providers compete on business factors—like price and customer service—and on implementation factors—like latency, downtime, and redundancy. Software interface reimplementation therefore fosters competition in the cloud by allowing customers to transfer their data or programs to competing cloud providers that offer cheaper or better service without having to learn an entirely new interface or rewrite their software to conform to a new specification.

B. Allowing Copyright to Restrict the Reimplementation of Software Interfaces Will Stifle Competition by Increasing Barriers to Entry for Startups and Others

The decisions below jeopardize the market for software. Reimplementing software interfaces enables startups to counter network effects and compete with established players. Network effects arise when a service's value increases along with its number of users. They make users unlikely to switch even to technically "better" competing software services that

---

[8] *See* Rita Zhang, *Access Azure Blob Storage from Your Apps Using S3 Java API*, Microsoft (May 22, 2016), https://www.microsoft.com/developerblog/2016/05/22/access-azure-blob-storage-from-your-apps-using-s3-api; *Cloud Storage Interoperability*, Google Cloud (last updated Oct. 23, 2018), https://cloud.google.com/storage/docs/interoperability; *Amazon S3 Compatibility API*, Oracle Cloud (last visited Jan. 6, 2020), https://docs.cloud.oracle.com/iaas/Content/Object/Tasks/s3compatibleapi.htm.

have not yet established a large userbase because much of a service's value comes from its community of users and its secondary market of compatible services. For example, a developer might choose not to learn a new programming language unless it is used by potential employers, even if that language is more intuitive than others and produces efficient results. On the other hand, an archaic language used by institutional employers is worth learning, regardless of its inefficiencies. Uncopyrightable software interfaces address network effect barriers by enabling startups to plug into existing systems and grow through cumulative improvements.

Just as the first car would look laughable today, the first word processing software would be a laughable replacement for modern applications. Yet a steering wheel, turn signals, and gas and brake pedals have been standard in cars for over a century. If Tesla had to re-invent the standard driving interface to make electric-powered cars, it would face high barriers in attracting new customers. *See* Fred von Lohmann, *The New Wave: Copyright and Software Interfaces in the Wake of* Oracle v. Google, 31 Harv. J.L. & Tech. 517, 517 (2018). In software, treating interfaces as copyrightable would be like requiring car manufacturers to invent a substitute for the steering wheel. Startups would not risk manufacturing such a car, and even if they did, consumers likely would not purchase it.

Furthermore, extending copyright to software interfaces would enable companies to monopolize standard interfaces. Companies could initially make their interfaces freely available to lure developers to their platform, and then, after attracting a significant number of developers, demand a licensing fee for further use. These fees would be passed on to consumers, making software more expensive. Copyrightable interfaces could also curtail employee mobility because different employers would use competing proprietary APIs, and employees with expertise in one proprietary API would be less desirable to employers using another. Innovation could stagnate.

Amazon, for example, could follow Oracle's lead and use the decisions below to force every company that has reimplemented its cloud storage APIs to pay a licensing fee, stifling competition in a vibrant market valued at $42 billion in 2017 and projected to reach $72 billion by 2019. *See* Jay Greene & Laura Stevens, *"You're Stupid If You Don't Get Scared": When Amazon Goes from Partner to Rival*, Wall St. J. (June 1, 2018, 5:30 AM ET), https://www.wsj.com/articles/how-amazon-wins-1527845402. Amazon could gain a monopoly over cloud storage until its competitors redesigned their systems from scratch to avoid infringing on Amazon's APIs. The decisions below will transform copyright into a tool for incumbents to wield to improperly stave off competition.

Forcing companies that reimplement APIs to rely on fair use will not meaningfully address these anti-competitive effects. A fair use standard creates uncertainty because it depends on fact-intensive, case-by-case determinations which can result, as demonstrated by this case, in lengthy and prohibitively expensive litigation. Rather than risk crippling lawsuits, startups will choose not to enter markets at all or will undertake inefficient workarounds. Restricting API reimplementation to situations where fair use can be established would impede innovation and competition almost as much as denying reimplementation outright: Users will suffer from fewer product choices, higher prices, and incompatible software.

C. Restricting the Reimplementation of Software Interfaces Will Exacerbate Lock-In Effects and Create an "Orphan Software" Problem

Reimplementing software interfaces protects consumers from lock-in effects by promoting interoperability among operating systems, programs, and Internet browsers. Consumers depend on operating systems that run on their hardware, programs that run across operating systems, and Internet applications that run across browsers. Under the decisions below, software interfaces enabling interoperability might require expensive licenses, and their owners could significantly restrict their use. Consumers will face higher prices and fewer choices. Software will become harder to use because switching to a competing service will require users to learn an unfamiliar interface. Rather than switch to more innovative software, users will remain locked in to outdated systems.

If software interfaces are copyrightable, it will become economically infeasible to continue using orphan software, i.e., software no longer supported or updated by its creator. Previously, when copyrighted software became unsupported, developers could reimplement its interface to allow it to run on new systems. When NASA needed to refurbish old manufacturing robots for a project, for example, it contracted with a company to reimplement the interface necessary for integrating newly manufactured memory chips with the old robot hardware. Had the interface been copyrighted, NASA would have needed to purchase new robots at a significantly higher cost.

Restricting the reimplementation of software interfaces could make generations of software unusable by the people and organizations who paid for them, hindering, rather than promoting, "the Progress of Science and useful Arts." U.S. Const. art. I, § 8, cl. 8. Copyrightable interfaces would particularly harm public, nonprofit, and research-based entities because of their limited resources, undermining crucial services for public health and safety, national defense, and access to justice.

CONCLUSION

The Court should reverse the decisions below and hold that software interfaces are not copyrightable to ensure continued innovation and protect competition in the software industry.

Respectfully submitted,

Phillip R. Malone
Counsel of Record
JUELSGAARD INTELLECTUAL
PROPERTY AND INNOVATION
CLINIC
MILLS LEGAL CLINIC AT
STANFORD LAW SCHOOL
559 Nathan Abbott Way
Stanford, CA 94305
(650) 725-6369
pmalone@law.stanford.edu

January 13, 2019

APPENDIX A — LIST OF AMICI CURIAE

(In alphabetical order)

Amici sign this brief on their own behalf and not on behalf of the companies or organizations with which they are affiliated; those affiliations are for identification purposes only.[*]

1. **Harold Abelson.**** Dr. Harold "Hal" Abelson is a Professor of Electrical Engineering and Computer Science at MIT, and a fellow of the IEEE. . . .

[*]The * indicates five amici who are current Google employees, ** indicates two amici who receive some support from Google, † indicates two amici who testified as unpaid fact witnesses at trial in this case, and ‡ indicates one amicus who was retained as an expert by Google but did not testify at trial. Each of these amici sign this brief based on their personal experience and beliefs as individual computer scientists whose work in the field long preceded their affiliation with Google or their participation in this case.