Pages 1 - 60

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

BEFORE THE HONORABLE WILLIAM H. ALSUP

ORACLE AMERICA, INC., Plaintiff, vs. GOOGLE INC., Defendant.	NO. C 10-03561 WHA San Francisco, California Wednesday April 6, 2011 1:41 p.m.

TRANSCRIPT OF PROCEEDINGS

APPEARANCES:

For Plaintiff:

Morrison & Foerster
755 Page Mill Road
Palo Alto, CA 94304-1018
(650) 813-5600
(650) 494-0792 (fax)

BY: Michael A. Jacobs
Marc David Peters
Ruchika Agrawal
Roman Swoopes

For Plaintiff:

Oracle
500 Oracle Parkway M/S 50P7
Redwood Shores, CA 94065
(650) 506-9432
(650) 506-7114 (fax)

BY: Andrew C. Temkin

(Appearances continued on next page)

Reported By: Lydia Zinn, CSR #9223, RPR
Official Reporter - U.S. District Court

APPEARANCES (CONT'D)

For Defendant: King & Spalding LLP
1185 Avenue of the Americas
New York, NY 10036-4003
(212) 556-2100
BY: Scott T. Weingaertner
Mark H. Francis

King & Spalding
1180 Peachtree Street, N.E.
Atlanta, GA 30309-3521
(404) 572-4826
(404) 572-5134 (fax)
BY: Bruce W. Baber

King & Spalding
100 N Tryon Street, Suite 2900
Charlotte, NC 28202
(704) 503-2630
704) 503-2622 (fax)
BY: Steven T. Snyder

King & Spalding
101 Second Street, Suite 2300
San Francisco, CA 94105
(415) 318-1220
(415) 318-1300 (fax)
BY: Donald Frederick Zimmer, JR.

For Defendant: Google
1600 Amphitheatre Parkway
Mountain View, CA 94043
(650) 253-0000
(650) 618-1806 (fax)
BY: Renny Hwang

Also Present: **John Mitchell**

THE COURT: Good afternoon, everyone. Welcome back. Please be seated.

Okay. We're here in the case of *Oracle versus Google*. Number 10-3561. Please make your appearances.

MR. JACOBS: Michael Jacobs, from Morrison & Foerster, your Honor, for Oracle.

With me at counsel table is Ruchika Agrawal, who, consistent with your Honor's guidance, will have a speaking role in the tutorial today. And then next to her is Roman Swoopes, who's assisting on the tutorial; Professor John Mitchell, of the Computer Science Department at Stanford University, our retained expert in the case; Andrew Temkin, corporate counsel at Oracle; and Marc Peters, my partner, who'll be handling the discovery issues this morning.

THE COURT: All right. And on the other side.

MR. WEINGAERTNER: Good afternoon, your Honor. Scott Weingaertner, of King & Spalding, for Google.

Here with me at counsel table are Bruce Baber; Renny Hwang, of Google; Steve Snyder, also of King & Spalding; Fritz Zimmer, King & Spalding; and Mark Francis, also of King & Spalding, who will be assisting with the presentation today.

THE COURT: All right. Here's what we will do. I know I noticed the discovery hearing for this. So we'll take that up last, if time permits. If it doesn't permit, then I

may have to refer all discovery in this case to a Magistrate Judge, because I'm in a trial that takes from 5:00 a.m. until 1:00 every day, and I run out of time.

I'm going to do your claim construction.

Yes, sir.

MR. BABER: Your Honor, I was just going to say -- and Mr. Peters can jump in, but we did have a plus-four hours meet-and-confer. We have reached agreement on a number of things. I believe it's going to be a question of reading into the record the agreements we reached.

THE COURT: How long will that take?

MR. BABER: Probably less than five minutes.

THE COURT: All right. Well, then, does that end it, or is there more for me to rule on?

MR. BABER: At this point, your Honor, I don't know if there's any more for you to rule on.

THE COURT: All right. Then let's put it on the record, and then we'll go to your tutorial. So that's good. That's good. Thank you for doing that.

Okay. Mr. Jacobs said he would be arguing the discovery.

Is there more -- is that true? -- to put on the record?

MR. PETERS: It will be me, your Honor. There won't be anything to argue if we just put this on the record.

THE COURT: Great. You two come forward. Here's the way you do it. You state what the agreement is. Pause after each sentence. And then the other side says, "Agreed," or "Not agreed," as the case may be.

Go ahead.

MR. BABER: Okay. Your Honor, we did discuss a lot of issues. We reached a agreement on a number of them. One by one. First are some process steps.

We agreed that if either party has raised an issue by e-mail or by letter to the other side, and hasn't heard back in a reasonable period of time for a response, that party will follow up, and simply remind the other party that the request is outstanding. In response to the follow-up, the other side will, by the close of next business day, respond with either some communication saying, "We're not going to do what you asked us to," or "We are looking into it, and we expect to provide a substantive response by a date certain." Just a process. Sort of make sure that communications keep flowing appropriately.

MR. PETERS: We agree to that procedure, your Honor.

THE COURT: And before you go any further, can I make a comment?

MR. BABER: Yes, sir.

THE COURT: I have a pretty strong rule -- not rule, but encouragement -- to get young people to argue in court.

On the other hand, I have found that if you delegate down to young people the discovery disputes, which often happens, they don't get resolved, and that they actually get resolved better if you give them the higher up the chain you go, because the lawyers higher up the chain have the competence to give something up. So I encourage you to either have your number-one or number-two person on the case to do that second letter.

In other words, if you're going to write back and say, "We're not going to do anything more on that," or "Here's all we're going to do, and that's the end of it," I encourage it to be number one or number two, and not a lower-down person.

MR. BABER: Your Honor, I think you've read our collective minds.

Second point is that every ten days, from now until the time of trial, we will have a telephone conference in the case, in which the senior-level lawyers on both sides will participate; either Mr. Weingaertner, myself, or Mr. Perry for Google; Mr. Jacobs, Mr. -- Dr. Peters, or Mr. Holtzman for Oracle, depending on the issues. We will have an agenda two days in advance of those calls, so any discovery or other issues that are out there that either side wants to talk about, we will have a set time for the senior folks on the case to talk about them, and try to resolve them.

And we have undertaken to send to Oracle, by the end

of this week, a proposed, specific schedule for the every ten days from here until the time of trial.

MR. PETERS: That's right, your Honor.

THE COURT: Great. Wonderful.

MR. BABER: Okay. Point number three, your Honor, is on supplementation of interrogatory answers, we have agreed that both parties will supplement interrogatory answers by Friday, April 22nd, which is two weeks from this Friday. And then we also have some additional supplementation from Google. I'll go into the details in a second. The broad parameter is April 22nd.

MR. PETERS: That's right.

THE COURT: Stop on that. That's good.

I want just say to you both I won't make a ruling on this now, but if I was in your position, I would have an exceedingly good answer, because I may encourage somebody to bring a motion to knock out a defense or knock out a claim for insufficient basis. So if you want to protect yourself against that, you should err on the side of more disclosure and more answer, and not hide the ball. Okay?

MR. BABER: Yes, your Honor.

THE COURT: That's just a word to the wise.

Okay. What's your next --

MR. BABER: Next point, your Honor, is a little more specific. On the supplementation of interrogatory answers,

Oracle has agreed that it will supplement its responses to Google's Interrogatories 1 through 10 by April the 22nd, and will include in those supplemental responses any additional information that supports its contentions or positions, and/or to take out any items that it no longer relies on to the extent its contentions have changed, and things that may have been identified as potentially infringing early on now may drop out of the equation, but that will be Oracle's supplementation.

Google's supplementation by April 22nd is that we will supplement our responses to Interrogatories 4. We will supplement our response to Number 5, if response to Oracle's recently supplemented infringement contentions. We will also supplement Number 6, 7, 8, 10 through 16 in the first instance.

And then on April 29th, after we've received their supplements, we will supplement again as to our answers to Number 9 and 10, which are copyright-related ones that deal with specific files. So we need to see what they come back at us with before we can respond.

MR. PETERS: That's our agreement, your Honor.

THE COURT: Wonderful. Next.

MR. BABER: Your Honor, we also discussed Google's response to Oracle's Interrogatory Number 3, which relates to noninfringement. We have agreed that by next Monday, we will send to Oracle's counsel a letter summarizing what we think we heard from them in the meet-and-confer, and clarify what more

it is they want us or would ask us to put in that interrogatory answer. They will respond to that, and confirm that either that is or is not the case. And we will then see what's on the table, and we will supplement it to the extent we believe it needs supplementation. And we'll do it within 14 days after that.

Once we have supplemented again on noninfringement, Oracle will then come back to us and tell us where they disagree with any new information we've shared with them. In other words, if we say, "It doesn't -- it's not covered by a claim for this reason," and then they'll come back and, if we ask them to, they'll tell us, "Well, we say it is, and here's why." So again, back and forth on a 14-day framework.

MR. PETERS: That's right, your Honor.

THE COURT: Okay.

MR. BABER: With respect to supplemental interrogatory answers generally, the parties have agreed in concept aspirationally to the concept of parity. That is, both parties will make their best efforts in good faith to provide a comparable level of particularity, granularity, detail -- whatever you want to call it -- that is being provided by the other side.

THE COURT: Isn't that what you agreed to last time?

MR. BABER: I beg your pardon, your Honor?

THE COURT: That's what both sides agreed to last

time.

MR. BABER: Well, we thought so as well.

THE COURT: It fell apart.

MR. BABER: I'm not sure it fell apart. We may be moving to a different level of granularity at this point. And so we will try to be sure we are each providing apples and apples. And that's true whether it's a question of fact, a contention, or, where appropriate, the application of fact to law.

MR. PETERS: Consistent with your Honor's earlier indication about getting as much information as possible through interrogatory responses, that is our agreement.

THE COURT: Great.

MR. BABER: Next issue, your Honor, has to do, instead of interrogatories, with the plaintiff's infringement contentions under the patent local rules.

Oracle has declined Google's offer to allow Oracle to supplement its infringement contentions again. Oracle has chosen to rely on its infringement contentions as currently framed.

Google's position that they are inadequate, in particular on the issue of proof of direct infringement; but the parties have agreed to disagree.

Google has withdrawn its offer to consent to an amendment of the infringement contentions.

And both parties have agreed to simply reserve their rights for later; to tee it up, if necessary, with your Honor.

MR. PETERS: That's right, your Honor. We have agreed to disagree.

THE COURT: Well, all right, but you remember what I said, you know. Shifting sands, and all that of that.

MR. PETERS: I know, your Honor.

THE COURT: If you want to stand on your shifting sands, and they get shifted out from under you, then you just lose. End of story. So I'm not saying you will, but we've got those disclosure rules for a reason. And if your -- if your disclosures aren't good enough, there won't be a second chance. You just lose. All right.

MR. BABER: And I believe the last one --

THE COURT: What's next?

MR. BABER: -- your Honor, was actually not an agreement, but simply to report to the Court we discussed a lot of other issues in the conference: Document productions, processes, electronic discovery. We didn't have a chance to address them all, or even resolve many of them, but we think, given the processes we've now put in place for conferences and communications, we're hopeful we will be able to address those.

We have some disconnects in terms of your Honor's order on 30(b)(6) topics, for example; but we think that with a little further work, we may be able to get those resolved so

that we can keep continuing forward. We've started taking depositions and --

THE COURT: Discovery cutoff is in July, so you'd better get cracking.

MR. BABER: We're moving, your Honor.

THE COURT: Well, moving is not -- I mean, okay. You've got to get moving, but there's a deadline.

MR. BABER: We understand.

THE COURT: All right. Okay. Is that it?

MR. PETERS: Yes, your Honor.

THE COURT: You all should be working for the State Department, you did such a good job. You found a way to just find ways to agree; not really an agreement; but you're finding ways to agree. So I need you over, you know, in the Mideast, I think, to help solve some problems. Okay. Thank you.

MR. PETERS: Thank you, your Honor.

THE COURT: Now we're going to go to -- well, the main event, which is -- and I do need that -- I've got to be chairing a meeting at 3:00 p.m., so I only have 55 minutes. And I'm sorry for that, but each side gets 30 minutes. So you use the 30 minutes as you wish.

Now, I don't want -- if you want to go over into arguments about the claim construction and all -- a little bit of that's okay, but right now I'm basically interested in

learning your take on the underlying technology.

It does help me in these tutorials to know roughly where the old technology started or stopped, and new tech in the invention began, because sometimes I go away from these tutorials thinking, "My God. That's great," and then I learn that all of it was prior art.

So it helps to know where the prior art stops and the new invention -- but you know, overarching thing is I want to learn the basic technology.

So, Mr. Jacobs, you go right ahead.

MR. JACOBS: That's our goal, your Honor. And we have for you the materials that we had prepared. I will confess that they are the materials from the longer version of the tutorial --

(Whereupon a document was tendered to the Court)

MR. JACOBS: -- from the longer version of tutorial that we had prepared. So there's more content that we've handed you than we're going to present today.

Our agenda is to cover the basic problem that Java solves: Application development and distribution for heterogeneous or diverse architectures.

The Java solution to that -- what's referred to as "Write once; run anywhere."

I want to explain a little bit about the Java programming environment that's background for the patents we're

talking about.

I want to spend a minute on Java licensing models.

We'll talk about and illustrate the patents and the embodiments in the specification for three patents. And I'll explain how we'll do that.

And then briefly I want to draw the Court's attention to the basic architecture of Android, as exemplified by Android -- the Android website.

So the problem is that computers are different. They present different functionalities. They have different instruction sets. The term of art is "architecture." They present different architectures. And they especially present different architectures to application programmers.

And that means that the programmer, when he -- when the programmer's considering, "Am I writing this application for a PC? Am I writing it for a Mac," he might write it differently. He might compile it into object code differently. It's all going to depend -- in the prototypical case, it's going to depend on the target's architecture; the target environment.

And so we've illustrated that here with an old Mac, an old PC, because in the '90s this was a very substantial issue, and an issue that the Java developers were aiming to solve.

The solution is -- this architecture is illustrated

on this Slide 3, in which the programmer writes an application program in, typically, Java programming language. That is compiled by a compiler at the development site into something called "bytecode," which is an intermediate form of code. It's architecture neutral. It is not targeted at a specific hardware platform or operating system. It is aimed at, instead, some code that is residing on the computer called "the Java Virtual Machine."

And so the bytecode and the Java Virtual Machine are the key interface element between the code that has been distributed by the application programmer, and the target hardware platform. You can think of the virtual machine as another layer; an insulating layer that insulates the application programmer from the underlying hardware and operating system.

THE COURT: May I ask you a question?

MR. JACOBS: Mm-hm.

THE COURT: If you start down there at the bottom of that slide --

MR. JACOBS: Right.

THE COURT: I think I know the answers to most of these questions, but those three computers -- you're not claiming that is the invention?

MR. JACOBS: That's right.

THE COURT: All right. And then the physical

machine -- you don't claim that as the invention?

MR. JACOBS: That's right.

THE COURT: Nor the operating system?

MR. JACOBS: That's right.

THE COURT: How about Java Virtual Machine?

MR. JACOBS: The specific inventions are a product of this solution. We do not have a patent that we're asserting here on this whole architectural solution, but in order to understand what the patented inventions arise out of, you have to -- one needs to understand that the -- that this is a solution, but it created its own problems. And those problems are resolved by the patented inventions.

THE COURT: All right. One other question. So I understand what you say. So you're saying that those four blocks at the top, counting the arrow -- that, while that is not the invention, the particularized inventions arise out of that: Those four blocks.

MR. JACOBS: Exactly.

THE COURT: All right. Now, was virtual machine in the prior art, or was that something that was invented by this invention?

MR. JACOBS: There were virtual machines in the prior art.

THE COURT: Okay. That helps me a lot.

Okay. Please continue.

MR. JACOBS: Sure. And, just to put this in the everyday experience, from the standpoint of everyday experience, you may have noticed, sitting at a computer one day, that code asked for the latest version of Java on your desktop machine. And that's exemplified here by this little Java coffee cup.

What happens then is your computer goes to a Java-delivering website, and it delivers the latest version of the Java Virtual Machine, or, as it's often referred to, "the Java runtime environment"; the "JRE." So that's how we encounter and deal with Java in our daily life on computers.

So the Java programming environment consists of the Java programming language. And the Java programming language is a programming language. And it's documented in a book called, The Java Language Specification.

And it's an object-oriented language, which means that it is -- that the code is written in packages. Those packages are called "classes." So the class is the key construct for a Java programmer. And the class is a -- can be thought of as a template or a form for instances of the class which will be objects.

So in the case of an e-mail, for example, the class will define the various fields of your e-mail message: Your "To"; your "From"; your "Subject" line. And then it will have methods that will cause the e-mail message to be sent, read,

and received, et cetera.

THE COURT: All right. Let's pause on that.

Coming into today's hearing, I could not understand exactly what was meant by "class" -- that term. So go over that again, and help me understand what -- how "class" is used in this -- in this technology.

MR. JACOBS: So as I understand it, literally, when you are programming in Java, you say to yourself, "I'm going to divide this overall program into a bunch of tasks and definitions, et cetera." And you divide that into what are called "classes."

In the old days, preJava, you programmed in modules.

In Java, you program into a class.

And the basic difference is that a class keeps the data and the executable code -- if you will, the methods -- together, and encapsulates them with clearly defined interfaces between classes. That's the essence of object-oriented programming.

And so the class is -- you literally write "class." And then you write the code and the data definition under it.

And, as you'll see -- I think this next slide will help. What happens, then, in the Java environment is that the programmer has organized his or her source code into classes. The compiler has compiled that into bytecode. And the bytecode itself is a class file or, more precisely, a .class file.

And then it is those .class files that the virtual machine -- the format of which the virtual machine understands, and can execute on the target platform.

Now, the reason I think the meaning of "class" is somewhat confused is that classes are just sort of pervasive in Java. And, in particular, there's something called "the Java class libraries." And those are libraries of classes. So those are libraries of code that have been written and distributed, with the idea that, rather than having an application programmer have to rewrite that code each time, the application programmer can just write an interface to that code, and invoke it at runtime.

So the class is something that's written by the programmer. It's also something that is available in the Java Platform. And there are dozens and dozens of these classes in the Java class library.

THE COURT: This one said "class calendar." What -- what would that be?

MR. JACOBS: Exactly. So that is a calendar function. In the e-mail example, perhaps you want it to get the date, so you -- rather than writing code for a calendar, you invoke the calendar function through the application programming interface of that class. It's a function, but it's been -- it's in the form of a class. And on the runtime in the -- on the computer, the end user's computer, those class

libraries are again available. Just as the bytecode was deposited in .class files, the class libraries are available at runtime or execution time for the application program to invoke and cause to be executed.

And those class libraries have an application programming interface which is documented in a couple of books called The Java Application Programming Interface, and, more recently, on the Web in documentation, so that programmers know how to write the application program interface -- how to use the API when they're writing their code.

THE COURT: What were those books that you just held up?

MR. JACOBS: This is actually from about '96, I think. This is The Java Application Programming Interface, Volume 1 and 2, by Gosling, Yellin, and the Java team.

THE COURT: What's the other book?

MR. JACOBS: Hm? It's Volume 1. Sorry.

THE COURT: Oh. That's Volume 1.

MR. JACOBS: Volume 1 and Volume 2.

And then the other book is The Java Language Specification, by Gosling, Joy, and Steele.

THE COURT: Okay.

MR. JACOBS: So Java licensing models. The -- Java has been in existence now for -- commercially, for around 15 years. And Sun, of course, commercialized Java, and Oracle

inherited that.

It's a quite complicated commercial licensing scheme, but boiled down to it, there are really three ways to get Java from Sun/now Oracle. One is through royalty-bearing commercial licenses. If you want to distribute Java on your system, then you can enter into a commercial license with Oracle. And there are -- these terms are all over the Java industry. The SCSL or the TLDA are well-known acronyms for Java licensing.

An important part of Sun's strategy for Java was to penetrate the academic environment and get students programming in Java. It was enormously successful. Java is the leading programming language in universities today. That was facilitated by a royalty-free academic licensing program.

Then most recently, Java was released by Sun -- what's called "OpenJDK." It was released under the GPL; the new public license. The GPL is a very strong open-source license, in the sense that, if you are a recipient of code under the GPL, you inherit GPL obligations when you modify or extend that code.

So, just as the original source -- in this case, Oracle -- has to make available GPL code as open-source code, the recipient of that code must similarly make his or her code on top of the Oracle code available as open-source code, and all the way down the line. That has the effect, probably, of inhibiting its commercial use.

There are some commercial uses of OpenJDK, but it has not had a substantial impact on the commercial licensing business of Oracle, because of these strong GPL restrictions that are imposed on downstream users.

THE COURT: The word "open" --

MR. JACOBS: Open source.

THE COURT: What does that mean?

MR. JACOBS: Open source. The source code's publicly available.

THE COURT: Without fee?

MR. JACOBS: Without fee; but with this copy left, as it's referred to, obligation to make your own source code publicly available.

So what will happen is, when we talk about Android later in the case, what you'll hear is that Android did not adopt OpenJDK, because the GPL restrictions were viewed as too constraining on handset developers. That's why this is ultimately going to be relevant.

THE COURT: Okay.

MR. JACOBS: So let's talk about the patents-in-suit, and, more particularly, three patents that present claim-construction issues for the Court in a couple of weeks. There are actually six patents that raise issues; but three of those only raise this computer-readable-medium issue that you'll be getting into when you read the briefs. Three of them

raise more technical issues, if you will. And those are the '104, the '702, and the '520.

So the '104 patent is the earliest patent in the case. It dates back to the very early days of Java development. And it is about resolving data references in generated code.

And, in order to understand this, I need to step back once more into this issue of diverse architectures and the challenge facing application programmers. So an application programmer can program in a source language, and then compile that code into a form that is intended for a specific machine.

If you are writing for a SPARC workstation, then you would take your source code, run it through a Sun compiler that is designed for the SPARC environment; similarly, for an IBM PC with an Intel chip. And then the program would execute that object code tailored for that target environment. Indeed, it was optimized for that target environment. So we're talking about code that should execute, if well written and the hardware can support it, lickety split.

This is the classic way that programs were distributed. If you and I went to Egghead Software 20 years ago to buy software for our personal computers, we looked at the side of it, and it said, "This will run on a Windows machine with a 386 processor," or "This will run on a Mac." That code went through a compiler intended for that

environment.

The other way to execute code is through an interpreter. And in the interpreter case, the source code is actually installed on the end user's machine. And an interpreter interprets each source instruction, one by one, and executes the code.

This interpreter is flexible, in the sense that all you can take is source code. And source code is relatively portable, relatively usable, machine to machine, but it's slow, because you're going through this interpretive step, and this step-by-step execution process.

So the '104 patent explains that what they are about in 1992 is developing a hybrid compiler/interpreter, comprising a compiler for compiling source code, and an interpreter for interpreting the compiled code. That's the backdrop to the invention of this patent.

An example of that is the Java architecture, as illustrated here in the way that we've illustrated earlier.

So the patent sets up the basic problem here, by explaining the following.

In Figure 1A, we're dealing with a compiler model. And we've taken some source-level instruction that required a variable to be captured. And the compiler, when interpreting that source-level instruction, figured out in the compilation model is the way to do that is do a Load 2 instruction, and

find the data value at Slot 2.

So compilation gets all the way down to the data-structure level, and tells the computer, "Go to Slot 2, and find the value we're looking for." It's fast, but it's not very flexible.

The other model is interpretation. And the '104 Patent discusses what happens under interpretation.

THE COURT: We have it there now.

MR. JACOBS: That's in Figure 1B. And in Figure 1B, we've -- the interpreter is receiving the instruction in the form, "Load Y." It's an intermediate form of an instruction that was probably something like, "Add X and Y," or "Draw a point, X and Y." That got compiled to an intermediate instruction of the form "Load Y"; but now the interpreter has to -- each time it encounters this instruction, it has to go figure out where Y is. And that is going to take machine cycles. That is going to be slow.

Now, recall that what we're going to do in Java is we're going to install this virtual machine layer on our end-user computer. And unless we change it, it's going to going through these steps. It is going to receive bytecode. It's going to encounter "Load Y," and it's going to resolve it, time after time. And that is going to be slow and inefficient.

So what the inventors came up with in 1992 was a kind of a hybrid solution for this hybrid architecture. And the

hybrid solution is to resolve this symbolic reference -- the "Load Y" reference -- the first time; but the next time around and subsequent times, rely on that previous resolution to the numeric value.

So we --

THE COURT: Say that again; that last sentence.

MR. JACOBS: It's going to rely on the resolution to that numeric value. So we'll see that the -- in this illustrative example in the patent, in the embodiment disclosed, the first time the interpreter goes through this process, it will find Y. It will find that it is in Slot 2. And then it will, somewhere, whether -- and this is part of the issue between the companies. Whether it's by literally rewriting the old reference, or somewhere else, some other method of relying on the fact that this resolution has occurred, the next time around, it will go straight to Slot 2.

So the symbolic reference has been understood once, and then, in a sense, converted to a numerical reference, speeding up execution. And this was a quite important contribution to the art of this hybrid compiler virtual machine architecture.

And we can see this expressed -- the invention expressed in an illustrative claim: Claim 11. So we have intermediate form object code. That's what the compiler generated in this hybrid architecture. And it has symbolic

references. In the illustration, that's a Y.

And then we are going to determine a numerical reference that corresponds to the symbolic reference. And then we're going to obtain data in accordance with that numerical reference. So I think of this. In my own mind, this is hybridizing the hybrid architecture, because, at the virtual machine, we're doing a kind of a compiler-like thing after we've received the code. We're resolving the variable into a numerical reference, doing it once, and not doing it repetitiously.

So the next patent with some substantive claim-construction issues is the '702 class file redundancy removal. So this is why it's important to understand class files. So the -- because the problem that this patent addresses is that, as we explained, we have the class files written by the application programmer. They retain their "class fileness" after compilation by a Java compiler. They are now .class files.

And then we have the Java class libraries, which are also class files.

And the upshot of all of this is that the -- when -- if you just load all of this up in memory and don't do anything else to it, as the patent explains, there could be hundreds and thousands of class files taking up space in your memory. And so what the '702 patent discloses is a way to solve that

problem. And the way that is going to be occur is by taking out the redundant elements, and putting them somewhere else.

So we have some illustrative class files here. The different colors represent different kinds of different elements.

It could be a lengthy string that would otherwise take up a lot of memory. And what the -- what a preprocessor does is examine the class files; pull out the common elements; put them in a shared location; and create a new format called an "mclass," or modified class files. The original class files have now been compacted by the removal of the shared elements. And illustratively, we're now using far less memory space.

Now, this is a -- an important function, because, as I mentioned, the Java bytecode has all of these class files in it that are independent chunks of code. This results in part of this being an object-oriented programming language.

Now, there's another step that's worth noting, because this is kind of a technical point that goes to the claim construction -- claim-construction issue here. In the original class file, recall the class files are self-contained. So the data and the code that relates to it are all in the same little package. And so the references in the original class file to what will ultimately be defined to be a shared element -- those are internal references.

When the mclass file is created, this new package

that contains the reduced class file and the shared elements that has to be -- somehow the reduced class files need to be adjusted, so that they point to the shared location. They're not merely -- technically, they can't just be a subset of the original reduced class files. They have to be modified or adjusted in order to point now to this shared location.

And, in fact, the specification discusses this, by noting that the -- there's a new constant type. And that new constant type replaces the duplicated element in the embodiment disclosed and discussed here, and replaces it with this pointer; this index into the shared location. So that's the '702.

An illustrative claim. We're going to preprocess the class files. So this is before the file is actually loaded into the virtual machine. We're going to form a shared table. We're going to remove duplicated elements. And then we're going to form this multiclass or mclass file that has a plurality of the reduced class files and the shared table.

The last patent that we'll be discussing is the '520. And Ms. Agrawal will explain that patent.

THE COURT: Thank you, Mr. Jacobs.

MS. AGRAWAL: Good afternoon, your Honor.

THE COURT: Good afternoon.

MS. AGRAWAL: So we talked with you earlier about how Java source code gets compiled into bytecode.

And what the inventors of the '520 patent realized was that sometimes the bytecode -- certain functions expressed in bytecode can be inefficient as is. So what the inventors realized is that you start off with the source code; compile it into bytecode. The inventors realized that if you simulate execution without execution, or simulate --

THE COURT: Say that again. You simulate?

MS. AGRAWAL: You simulate execution before actual execution. You play execute it.

THE COURT: Uh-huh.

MS. AGRAWAL: You go line by line, and you simulate the execution of this bytecode. You figure out what its operation is, and you determine a reduced set of instructions. A reduced set of instructions is about -- it's much fewer lines of code than the original bytecode. And the reduced set of instructions get fed into the virtual machine for actual execution.

THE COURT: Is that done every time you run the program, or just -- this is a designer tool?

MS. AGRAWAL: This is designed by Java engineers. The idea is that it's done for certain sorts of static initializations.

The primary example discussed by the '520 patent are static initializations of erase, because they come up often. And the inventors realize that the way that the bytecode

instructions that was designed, there's this -- there's an opportunity to offer this optimization.

THE COURT: Well, let me rephrase my question.

If I were to go out and buy a modern computer with modern software, and -- does it have this play execution featured on it, so that each time I run my software, it figures out a more compact way to economize on steps? And -- or -- you see what I'm saying? Is that the way it works?

MS. AGRAWAL: Yeah. So it's not observable to the end user, in the sense that you don't -- that you don't actually see --

THE COURT: I don't see it, but it's going on inside the computer?

MS. AGRAWAL: But it's going on.

THE COURT: Okay. Good. All right.

MS. AGRAWAL: And here's our illustrative claim of the '520 Patent, which just covers exactly what we've talked about. You receive bytecode, and you play execute the bytecode, without actually running the code, to identify the operation, in order to figure out what it does, in order to create a reduced set of instructions.

THE COURT: Let me just read that.

A method comprising. Receiving the code. Play executing the code without running the code. Among the processing component to identify the operation. If the code

were run by the processing component and creating an instruction for the processing component to perform the operation.

Okay. All right. Go ahead.

THE COURT: Is that it?

MR. JACOBS: So these are the -- where we started with was this point that these inventions arise out of the Java architecture. They solve problems created by this solution to the problem of distributing code for differing computers.

It's -- the Java architecture is worth -- it's worth observing how the Java architecture is exemplified in Android. Just to set up where we're going in this case, this is all material that is available on the Android website.

So if you go to the Android website, you'll see that the application programming interface level, the Android software development kit provides the tools and application programming interfaces necessary to begin developing applications on Android using Java.

So what is the programming language for Android? It's the Java programming language.

And then this virtual machine architecture is -- is -- has its modeling in Android. Every Android application runs on an Android device under a virtual machine. In Android, that virtual machine is called "Dalvik." And it runs classes that have been compiled by a Java language compiler. And then

Android developers added this step of converting the Java bytecode into something called "dex," or ".x," using the dx tool.

And Android includes a set of core libraries that provide most the functionality available in the core libraries of the Java programming language. So we talked about libraries; the class libraries. A subset of class libraries are called "core libraries." And Android provides a set of those.

So one of them is this calendar application programming interface. We saw this earlier in the Java documentation. And this is a set of the Java core libraries at Slide 28. These are Java core libraries provided with the Android distribution.

So if you go look at the Android source code, you encounter this set of java.awt.font, et cetera, and java.beans, and down the line. This is basically a compare of the Java core libraries with the Android core libraries.

And so if you look at the Java architecture and the Android architecture, you see that in both cases, the programmers are programming in Java. They're both -- in both cases, they're presented with class library APIs. Actually, both of them use a Java compiler. In the case of Android, there's this add-on of Java bytecode translated to dex code. They both present this virtual-machine architecture, so that

the bytecode can run on any device that has the VM installed. They both present Java runtime -- Java class libraries at runtime. And then, of course, it can be any operating system and physical machine that will support the virtual machine in each case.

And our basic contention is that, having chosen that architectural similarity, what we'll be proving later in the case is that they didn't have much choice but to implement the inventions embodied in the patent-in-suit.

THE COURT: Now, the -- is there a copyright claim in this case? I thought there was.

MR. JACOBS: There is a copyright claim.

THE COURT: Just on that slide -- I know we're here today on claim construction, but how does the copyright claim figure into that slide?

MR. JACOBS: It figures into the slide here (indicating). It figures into the implementation of Java APIs; documented and copyrighted documentation made available under license that has requirements that Android does not meet. And it figures in the Java runtime libraries which implement that API.

And then there are some elements of actually directly copied code that we have observed in the two sets of the code represented by each implementation.

THE COURT: All right. Okay. Thank you. You've got

about two minutes left for last word at the end, but let's now give equal time to the defendant, Google.

MR. WEINGAERTNER: Good afternoon, your Honor.

THE COURT: Good afternoon.

MR. WEINGAERTNER: We also have some hard copies.

(Whereupon a document was tendered to the Court).

THE COURT: I only need one of these, right?

MR. WEINGAERTNER: Yes, your Honor. I can take those back.

THE COURT: Give the rest to my law clerk.

MR. WEINGAERTNER: In the months that have gone by since filing this lawsuit, Android devices, which are accused, have become the fastest-selling smartphones in the world.

Oracle's point here is that that is somehow due to Java technology.

The Court, in its scheduling order, pointed out that this dispute is a Java-fueled dispute, and that is certainly true. Our view is not that it's a Java-fueled product, and we'll explain that to some extent.

Our to tutorial is, I think, focused on a little bit more technical nuts and bolts. I know we don't have a whole lot of time. I'll try to go through them in a meaningful way. And perhaps if your Honor has specific questions, I'm happy to try to address them to the extent we can. Obviously, the subject matter is very technical. The seven patents-in-suit --

this, I think, transcends claim construction, to try to give the Court more tools to be able to understand all the different balls that are in the air in this case.

And what I'll try to do is tie them up at the end, and perhaps provide a little bit of rebuttal to some of the things that Oracle's counsel pointed out.

Next slide, please.

What I thought might be helpful is first to give just kind of a brief overview to provide some context for what's happening on the accused-product side.

Google and other members of an organization, a nonprofit-type organization called "The Open Handset Alliance," made up of various companies, including handset developers, developed Android as an open-source platform.

Your Honor asked about what that means. In this case, it means truly free. I think there are probably differences of view as to whether it is economically free. In this case, Java is free both monetarily, and also free from basically restrictions on how it's used.

There is a compatibility test that people can do, and then they'll use the Java trademark; but it's very open. And when -- and what I'll do is point up to the screen here.

The Android open-source code is provided on a server. Anybody can access it, but it's typically accessed by folks who can actually put it to good use, such as handset and device

manufacturers. And they create devices like the one sitting right here on my belt. And a lot of folks in the room have them as well. Some of the better-known ones are Motorola, Samsung, HTC.

And that code then gets downloaded onto a handheld device. And the folks that create the code that actually gets on the device can do basically whatever they want with it. And that's actually a bit of an issue in the case. It relates to whether the burden is being met on infringement contentions. So it doesn't really relate so much to -- it relates a little bit to claim construction, because it has to do with how the claim language is parsed and applied throughout what is sort of known affectionately as "the ecosystem."

And so the ecosystem, again, starts with the creation of the open-source code; flows down through the handsets. Handsets are generally sold by the mobile operators, who give a discount because you, you know, sign up for, you know, a contract of some kind to use their services. And then there's developers that create applications.

And the idea behind -- one of the ideas behind open source and behind Android is that, by having this be free, application developers flock to the opportunity to create code. And there are architectural features Android that have made this specifically attractive. They, I don't think, have much to do with this case. I think they're independent.

Android is a big project. I think there are something like 11 million lines of code over many years. It's been, again, developed by a number of different folks; Google being sort of the lead shepherd.

And the developers have access to something called the "Android SDK." It's the development kit. And they create applications. And anybody can do this. Big companies do it. Individuals do it. I can go onto a store and download an app. I can write my own. I can reach in and modify code. One of the folks in our legal team is the type of person who does that. I haven't done it myself yet.

Next slide, please.

This just gives kind of an overview. I know that some of the typeface is small, your Honor, so there's no way to really fit it in otherwise. And this is kind of a map of the Android architecture. Other products, including Java, you know, have illustrations that kind of show this sort of "stack," as it's called.

And the different layers are sometimes known in computer science -- known as "layers of abstraction," in some sense.

At the top are applications that we're all familiar with: Keeping our contacts, you know; an actual phone functionality browser.

There's the application framework. That is a little

bit more difficult to parse.

"Window Manager" is sort of ascertainable as we stand here in the court.

And there are libraries, which are the next level of kind of complexity in a way that are harder to apprehend as we sit here today.

And on the very lowest level is something called "the Linux kernel." Linux is an operating system developed -- I think it's some sort of derivative of UNIX in some way, that was developed as open source probably 20 years ago or so, or maybe a little bit less, that provides a lot of underlying tools that people commonly rely on.

Now, where the action is in this case is the Android runtime environment. And within that -- and I apologize, your Honor, that the typeface is small here, but it's the Dalvik virtual machine, which is named for a town in Iceland where one of the engineers' family used to live, for whatever that's worth.

And there are core libraries that are specific; that are very tightly wound up with how the Dalvik virtual machine works, that are kind of shown here.

And the rest of this refers to and calls functionality; run -- running applications that people have downloaded.

Next slide, please.

We thought it would be helpful to give just a very kind of high-level overview of just what the structure is of a computer as you sit there and look at it; not --

THE COURT: I understand that much. Skip that one.

MR. WEINGAERTNER: Okay. Thank you, your Honor. Appreciate the feedback.

Just a brief word on APIs. This was something that your Honor asked about. One of the things -- and I'll just go very briefly on this. It's basically a mechanism to allow different applications, different programs, to talk to one another. It makes -- it makes the market function more smoothly, and allows programs to reach down into things like libraries, which we referred to earlier -- or Mr. Jacobs referred to earlier -- without everybody having to -- it's sort of a Tower of Babel solution, in a way, your Honor.

And just one quick note is that APIs permit an application to reach down into the operating-system layer or into other programs.

I thought it might make sense to be speak very briefly about just what programming languages are, and how they differ.

There's very high-level languages, which are sort of human-readable. Folks who write code look at them as being very human readable; but machines can't read them. So you have to kind of move from the human write-able and readable side,

down to a form that ultimately, as your Honor, I'm sure, knows, ones and zeroes that represent high and low voltages that computers use to actually carry out operations.

The translation from one level to the other that needs to be done is done by compilers that take the kind of human-readable code, and takes it down to a level that's general not human readable, although some of our engineers say that they actually can read some of this very low-level code.

THE COURT: Literally the ones and zeroes on the right side there?

MR. WEINGAERTNER: Yes, your Honor.

And, in fact, I'm glad your Honor raised that question, because there's an issue in this case of what it means to be in an intermediate representation.

This case involves something called "bytecode," which, is in a way, in between high-level languages. It's a little bit easier to see what's going on. And bytecode is important for how virtual machines operate. And I have a few slides on that.

So a virtual machine can be thought of, in a way, as an adapter that allows -- the adapter is basically written for each piece of hardware operating system. And once it's in there, you can write code that runs on any of them. And Mr. Jacobs referred to that. It's an old concept that goes back at least as far as the '60s. And there are reasons why it

became popular later, but it's -- the idea here is that somebody does all the work for you. Somebody writes the virtual machine --

THE COURT: That right-hand column, you're saying, goes back to the 1960s?

MR. WEINGAERTNER: Yes, your Honor. And we've got a time line.

THE COURT: We'll come to that.

MR. WEINGAERTNER: Okay. I --

THE COURT: Just walk me through particularly that second and third step there on the right-hand column.

MR. WEINGAERTNER: Yes, your Honor.

And so -- and again, we've got user here, but in a way, you can also imagine a developer being there who writes the code. The code is written in a way that -- using words like "If," Then," Fetch," Execute," perhaps, or other -- or "Do this for this period of time until this condition is met."

And that kind of -- if you reduce it down to the ones and zeroes level, it has to be very specifically done for this operating system. So the virtual machine says, "Hey. You know what? Just compile it down a little bit; not fully down to -- but somewhere in between."

And that virtual machine will then take that next step from that language down to a specific operating system. So again, the folks that write the virtual machines who do the

hard work of building the bridge here -- that's very specific to the operating system which is, in turn, built to run on a particular hardware.

THE COURT: Yeah, but over on the left-hand side, you've still got it running with ones and zeroes at the bottom with the orange box. So what's the difference between that left column and the right column?

MR. WEINGAERTNER: That's a great question, your Honor. The difference is that this native application right in here, which we don't show, is a compiler. Compiles down specifically for this machine.

And so what I have to do is actually write this application for this operating system. And so if I'm an application developer, it's kind of a huge burden for me to say, "Well, I'm going to write one to run on Mac; and write another one to run on Windows machine; and write another one to run on Android," or whatever it might be.

So the reason that this wasn't always done, because it seems sort of -- well, why would we always do this? -- is because it involves more processing time. And that was an issue. Your Honor may recall there was a time when computers ran very slowly, and they didn't have much memory. And so people would use these things, but it was a real burden.

And in the '90s, as you know, chips became a lot faster. And we all can see the effects of that around us.

Virtual machines became more practicable.

And, in fact, the advent of the Internet made them more practical, in a way, as well, because you'd have code that would be coming down over the 'net to any device. And so -- and so Java, in a way, kind of rode the wave. And, in fact, we have a quote later in the presentation that your Honor can read if we don't get to it, that kind of credits, in a way, the rise of the Internet with Java. And Java kind of caught that wave, in a way, and rode it; but the concept's predated by more than one decade, for sure.

THE COURT: And the phrase "virtual machine" was back in the '60s, too?

MR. WEINGAERTNER: Yes, your Honor.

Next slide.

Just an example of some source code; nothing all that interesting, but I guess the thing, I suppose, is it's slightly more human readable.

Just flip back for a moment.

There's actual English typed words in there. This is the most simple of all programs. It basically says, "Print to the screen. Hello world."

It's used a lot in textbooks to show this basic structure. And this is called, and it doesn't return a value. It just says, "Print to the screen," but that gets represented at the machine level in this language that only computers can

read, where -- sort of that's designed for computer reading. That basically allows the microprocessor that -- most of which had been developed down the road here -- can operate on. So the compiler basically translates from one to the another.

Actually, probably skip this one (indicating).

Again, the point here is that you can't run on two different systems. That was -- hearkens back to the point your Honor raised earlier: What was the difference?

Well, here, there is no compatibility. We all remember the days when you agonized over which computer to buy, because you knew that you might love a Macintosh or you might love a Windows machine, but some of them, you couldn't run certain programs and certain games on them, for example.

Next, please.

So again, just to cover virtual machine -- and we have other definitions -- you know, it basically simulates a physical machine; simulates an operating system, so the programs don't have to worry about where they're running. They think they're running on any machine.

And we can skip this.

Just for your Honor's reference, there's a few different definitions out there. They're not too controversial.

THE COURT: Still, it must take more time, I would think.

MR. WEINGAERTNER: Certainly it takes more time to actually run, because you have a whole additional level of processing.

And counsel for Oracle pointed out that their -- that their improvements are directed to, the way I see it, efficiency and improvements. They try to make these things run a bit faster. Real estate is at a premium on a cell phone. Power is at a premium. You know, every little bit helps, so to speak; but you know, in our view, it's a very incremental invention. And it's not the whole virtual machine was invented, by any stretch.

Okay. Next slide, please.

Again, the term that comes up is this "Write once; run anywhere" concept. Again, that sort of goes hand in glove with the concept of a virtual machine, and disconnected from any hardware.

Next, please.

Just a quick note. There are fundamentally different ways to realize a virtual machine. And this is a little bit detailed, because it comes from stack-based architecture, where the instructions are going to push down, kind of like cafeteria trays. It's called "Last in, first out." You push your instructions down, and then you kind of pop them back off. It's a way of keeping track of where you are, where you actually run each step.

Next, please.

There's also a completely different approach, which is register based, which you can think of as, rather than pushing down onto a cafeteria tray, you reach individual cubbyholes that have specific names on them. And there's -- it goes down to the fundamental --

And that's actually an important point here. Java -- the Java Virtual Machine is a stack-based architecture. This isn't directly, you know, relevant to any particular claims in the case, but it's the kind of important, we think, in terms of telling the story, that -- that Android is register based. And in some ways, that actually permits/facilitates multitasking, which is something that Android is regarded as doing very well. It makes the handheld devices more useable. People have multiple applications running at the same time. And a register-based approach facilitates that to some extent.

Next slide.

So this just compares. Our point is that they're different.

Next, please.

So, your Honor, this is a time line that shows the various virtual machines over the course of time. And these are all, you know, searchable on Wikipedia and other places. Just to kind of get a sense of where we are, Java Virtual Machine, you know, falls here; a couple, three decades after

some of the earliest ones. Dalvik is out here. And Flash is one you're probably familiar with, your Honor, that permits running of videos, and so forth, on a machine. Microsoft has its version.

So, just to go over pros and cons, programmers can create a single program that will work equally well. It's great for the Internet, because you don't have to are worry if it gets downloaded to whatever machine happens to be running the code. And, of course, folks who have code are continually making it available to be downloaded to create this ecosystem that we referred to.

The problem is that it can be slower, because it's an extra layer of interpretation. It would be like walking around with an interpreter by your side who had to interpret every sentence that you say.

Another potential issue is that, because you're kind of reducing to a particular common denominator, you have to take into account that different machines might have different features, and you have to speak kind of commonly to the ones that overlap with each other.

Okay. This is something that your Honor raised earlier. What is this bytecode?

Bytecode, again, is something that's a notch above the actual underlying machine code, in a way. It's an example of the intermediate form that code can be in. And that's one

of the terms that's being construed in this case.

We're not going to go into details on that. We just wanted to make sure it was clear kind what was really at issue here.

And again, here -- here's an example of what bytecode can look like. It's a little bit more abstract. It can actually be implemented in ones and zeroes.

And what's important, I guess, to know as well is that the virtual machine doesn't compile the code down. It actually interprets the code. So it goes instruction by instruction, and then reduces it down to the ones and zeroes to actually carry out the instruction. And that can be slower as well than compiled code, that's already down at that level.

THE COURT: Can you go back to that last slide right there? Right there.

MR. WEINGAERTNER: And then, just to skip ahead one -- so the point here is that you can actually run Windows on Apple. And the folks that actually write the virtual machine did the underlying footwork to create that sort of adapter that sits on each of these machines.

And this is --

In fact, just put all three of them up.

This just gives an example of, side by side, what the differences are between source code, bytecode.

THE COURT: Let's pause on that.

MR. WEINGAERTNER: Yes, your Honor.

THE COURT: What is that language that's on the bottom left?

MR. WEINGAERTNER: Okay. This one -- I think I'll have to ask Mr. Francis. I think this is in Java. I'm not exactly sure. Java, in a lot of ways, is actually relatively similar, and actually derives in some ways from C, which is an older programming language.

THE COURT: Well, okay; but I'm curious. Go ahead and ask Mr. Francis.

MR. FRANCIS: Your Honor, it's simply an example of what the code at the source-code level would look like. This particular example in this instance, it's creating a button on the screen that --

THE COURT: I understand. What's the name of the language?

MR. FRANCIS: I believe it is Java, your Honor.

THE COURT: Okay. All right. So that one -- so then, to get down to the bytecode level, those -- why is it in different colors? I don't think that one that's in orange and purple and green.

MR. WEINGAERTNER: Probably bad design choice, your Honor, I have to candidly admit.

THE COURT: So the reason I ask you --

MR. WEINGAERTNER: This could easily be in black.

THE COURT: So the -- the middle step -- what does that correspond to in the left-hand step, where it says "iconst_0"?

MR. WEINGAERTNER: Again, this is, I think, an example, your Honor. It would be very difficult, because each one of these instructions will require particular operations, like store a particular value. So it's actually not really even possible. If you could imagine, there isn't a one-to-one correspondence. You'd have to basically take one of these instructions, and kind of expand that.

THE COURT: Is that a real thing, or did somebody just pull out things that look nifty, and stick them up there but they didn't really work?

MR. WEINGAERTNER: It's real. In terms of the format, what we didn't do was take a particular instruction and de-compose that. I'm not sure. I guess it's an example.

THE COURT: It's just illustrative. It may not be real. All right.

And then the bytecode gets reduced down to zeroes and ones.

MR. WEINGAERTNER: Under this -- I'm sorry, your Honor.

THE COURT: Okay.

MR. WEINGAERTNER: That is correct. And also under the same illustrative condition, not specifically taking --

THE COURT: Who invented the word "bytecode"? Where did that come from? Is that in the patent?

MR. WEINGAERTNER: It comes up in the patent. It's certainly -- my understanding is that it actually comes out of the Java -- I'm sorry -- out of the virtual-machine concept that predates Java. I think it goes back before that. I'm not sure of the exact -- etymologically the first time it occurred in the language, but my understanding of it is that the notion of it, at least, goes back, you know, to the early days of Java -- to the early days of virtual machines prior to Java.

THE COURT: Now, if you had -- if you were trying to solve the problem of an application -- a single application being written once, but run on anything, and you had a Mac and a PC over there, so the intermediate bytecode would be different; would wind up being different, I assume -- tell me if I'm right -- so that it would trigger the right machine code for the two respective --

In other words, the bytecode would be different for the --

MR. WEINGAERTNER: No, your Honor.

THE COURT: For the PC, it would be different than the one from the Apple?

No? Not right? Okay.

MR. WEINGAERTNER: It's a great question.

You could almost imagine the virtual machine running

in between here, taking the bytecode, and converting it to the machine code, because that's where that actual layer and additional level of processing and time that's required resides: In this layer.

THE COURT: Something is omitted there. So there's another --

MR. WEINGAERTNER: Sorry, your Honor?

THE COURT: There's another step there that's omitted?

MR. WEINGAERTNER: No, your Honor. This just shows the different kind of languages.

In the earlier ones, we just said, "How do you get from one to the another?"

Here, in order to get from bytecode, you actually have to bridge this gap, and bridge this gap here; but this right here is actually running on the actual machine, so the bytecode itself is portable.

THE COURT: So the bytecode would look the same for the Apple as the would for a PC?

MR. WEINGAERTNER: Yes, your Honor.

THE COURT: I'm taking up too much of your time. You've got about four or five more minutes, and then --

MR. WEINGAERTNER: We're happy to, whenever we can --

THE COURT: Okay.

MR. WEINGAERTNER: Maybe I can find an example to

skip to. Let's skip to Slide 39. I should have worn my glasses.

Okay. This is the quote that I mentioned earlier.

And, by the way, your Honor, I just wanted to mention one point. There was a discussion about class files earlier, and I think there may have been a misapprehension. The concept of a class, although present in Java, actually grew out of object-oriented programming, which precedes Java. The C++ program language, your Honor, is one of those. And so Java takes advantage of this notion of what a class is.

And so earlier we mentioned that there is -- there are some interesting quotes from the folks who actually developed Java; James Gosling being the person who is known as the progenitor, in some ways, of Java, pointing out that networking and Java go hand in hand, because the network allows you to move things around to different machines really easily.

And so Java, again, in a way, kind of rode that existing wave.

Next slide, please.

We wanted to mention something that had to do with something that Mr. Jacobs pointed out, which is that there's a notion that's at issue here involving what it means to be computer readable, computer-readable medium. I want to not really say much about the claim-construction issue, but he thought there were some technical issues we should present that

would inform the discussion two weeks from now.

And the point is that -- I think if we turn back one slide --

As Mr. Gosling said, getting Java out there is all about making code available over network. It could be one's handheld device. It could be a computer, like the one sitting right in front of us right here. And the code is received over a network onto a machine.

It's important, because the way some of the claims are drafted specifically set out to kind of capture this notion that the code is being transmitted.

Next slide. Thanks a lot.

And it could be transmitted from a local storage device over a wire. It could be transmitted from a server over the Internet. There could be any number of media that get used, known primarily to big companies that have made huge investments in telecommunications transmission equipment. And it could be over a local area network; but in all instances, it's being transmitted.

Next slide.

And these are just some examples of how data and code ends up on our computers. Coaxial cable is well known. You know, cable TV comes over coaxial cable. Copper wires, we all know. Fiber is reasonably common. Infrared. People sometimes will send handheld devices using infrared transmission.

Sometimes it's acoustic waves. Fax machines, for example, use that. Light, which is, of course, related to fiber optics. The point is there's a what's called a "carrier wave." This is one representation that has a particular frequency. And signal that you want to carry in it. This wave a modulated to sort of impose this signal on this carrier. This is just a very generic example, but that's how data actually winds up on machines over the cables and over the Internet.

THE COURT: Is this an issue in the case?

MR. WEINGAERTNER: It is, your Honor. The definition of what it means to be computer readable is an issue. So we wanted to provide a little bit of a background for your Honor to refer to at your Honor's wish; but that's really all we have to say about that.

THE COURT: All right. What would you -- use the rest of your time to take just one major issue, and lay out what your view is and the opposing view is, so I can kind of see where the two of you part company; but just on one major issue that divides you here.

MR. WEINGAERTNER: Okay. One major issue. I guess we could use this '104 Patent. It's the earliest-filed patent of the group. Mr. Gosling's an inventor on this one. It's drafted in some ways very broadly.

Interestingly, the patent was reissued, I think, more than once, years after it originally filed. And the claims got

broader and broader. So we're dealing with the ones that tend to be quite broad, and try to go down at the root level, to what it means to resolve a data reference. We have a difference in view as to what that means. It's one a few examples. They're not -- they're sort of different issues from one another, but I can certainly focus on this one.

And what I want to do is, if we could skip ahead a little bit to Slide -- let's see here -- oh, jeez. Slide 45.

And Mr. Jacobs had very nice slides that showed, with animation, how things worked. The idea here is that you're resolving symbols down to numeric references, which actually means taking something that is unspecified as to where it will sit on a computer to a specific location. It's tied to a specific place that you can go and find data.

And our understanding of the prior art, which is actually referred to in the patent -- and we have lots of others that go back also decades -- that this idea of having to -- you know, to have the flexibility of a virtual machine, but the speed of compilation, if you have to go and basically resolve these each time you refer to them, it's time consuming. And so the idea is to do it once, and not to do it again.

And the way that we believe the patent describes it is by rewriting it, so that you needn't do it that over and over again.

And the two parties have a difference in view as to

whether or not resolving actually requires rewriting, or something less than that. So that's kind of a simply put issue for one of the patents in this case, your Honor.

THE COURT: All right. Let's leave it there.

Mr. Jacobs, I'll give you two minutes if you feel like there's some rebuttal you want to make, but I apologize for not having more time. I need to bring this to a close.

MR. JACOBS: Your Honor, I think in many ways the tutorials complemented each other; but there is one place in Google's presentation there's a complete disconnect between what we see, and what Google sees. And that's on this question of the freedom of device manufacturers to change the code at will.

Let me step back by saying that, in the Java world, there is a whole regime set up to ensure that the interface to the Java Virtual Machine, whoever implements it -- if it's going to be called Java, it's going to meet it interface definition in order to protect "Write once; run anywhere." If the JVM across different machines is different, then the bytecode won't execute properly.

Mr. Weingaertner pointed out that Java bytecode will not execute on Android machines. From the Oracle side of the world, that represents a fork in Java, because instead of protecting "Write once; run anywhere," they've diverged from it, and created a different interface. It has to be dex code

if you're in the Android world; not bytecode.

But within the Android world, they, too, want to protect "Write once; run anywhere" as between diverse Android platforms. So Google has told the users of Android -- the device manufacturers -- to, wherever possible, as illustrated in this slide, use the code available via the Android open-source project -- excuse me -- rather than reimplement significant parts of the system. Develop device implementers are strongly encouraged to based their implementations, to the greatest extent possible, on the upstream source code.

And, more recently, in the news, there have been reports that Google is imposing very tight restrictions on its licensees' ability to adjust the code residing on their devices.

So, from a technical standpoint, we see both companies, within their own worlds, aiming at very similar things.

We aim, in the Java world, at Java compatibility between diverse devices.

In the Android world, they aim at Android compatibility at the Dalvik layer.

And so they have implemented a compatibility test suite that has an analogy to the Java compatibility test suite. And they have told their device implementers, "Use the Android code that we make available."

THE COURT: We have to leave it there.

MR. JACOBS: Thank you very much, your Honor.

THE COURT: You all did a great job. And I look forward to seeing you in two weeks. All right. Thank you.

(At 2:57 p.m. the proceedings were adjourned)

CERTIFICATE OF REPORTER

I, LYDIA ZINN, Official Reporter for the United States Court, Northern District of California, hereby certify that the foregoing proceedings in C. 10-3561 WHA, Oracle America, Inc., v. Google, Inc., were reported by me, a certified shorthand reporter, and were thereafter transcribed under my direction into typewriting; that the foregoing is a full, complete and true record of said proceedings as bound by me at the time of filing.

The validity of the reporter's certification of said transcript may be void upon disassembly and/or removal from the court file.

_____

/s/ Lydia Zinn, CSR 9223, RPR

Friday, April 8, 2011