Title: Session 1: Java Enterprise

Steve Mirman: Hi, my name's Steve Mirman, and I'm an IBM Application Infrastructure Expert. Today we're going to discuss the standardization of Java 7, JEE 6 and OSGI, as well as go into some of the details of new functionality that's available in the platform.

For the agenda, first we're going to discuss the terminology, then we're going to talk about Java SE version 7, Java EE version 6, and talk a little bit about OSGI.

So first, let's get into Java. Java really refers to two things: It's both a language and a platform. So a language is the development aspect, writing your code in Java, utilizing the APIs. The platform itself is what your application is going to run on and there's really three flavors of that. There's the Java Standard Edition, or JSE, which really contains a good all-around mix of general purpose libraries. So here we're talking JDBC, JNDI, RMI and a few others, really designed for the most common use cases. The second is Java EE which is the Enterprise Edition. This includes everything that's in Java Standard Edition but also has some of the advanced capabilities. So Java Mail, JMS, EJBs. Really, every enterprise scenario is covered in Java EE. Now the third flavor we have is Java ME, which is Java Micro Edition. And this includes the majority of the stuff in Java SE, but also some additional APIs specific for hand-held devices. So this is kind of digging a little bit more into the mobile space.

So let's go ahead and start talking about Java SE, Version 7. So what is Java 7, and also what is the capability and what are the features of it. So this was a major platform release, and it was really designed to affect all the areas of the Java programming languages. There were five main categories, though, that they were focused on. First was the compatibility. So, Java 7 was designed so that code that you developed in a previous JD case should in theory function correctly under Java 7. So your Java 6 application should function properly. Things were deprecated in some of the functionality, and we'll discuss that in a little bit. But the goal was that, whatever you've written under a previous release should in fact compile and run under the new release. Productivity was kind of another key focus for Java 7, and it's really designed around streamlining some of the programming models—make it easier for developers to develop code, whether it's making less code to write, giving annotations to make it easier. Having the compiler and having the Java underlying system make it easier to manage resources and manage them for you. The third was performance. So there's several new APIs that were introduced to affect the overall performance. And they're really designed around, let's take advantage of multi-threading, let's take advantage of multiple core systems. Let's allow synchronization or let's allow sending out async calls so that we can now multi-thread a lot of our applications. It makes it very easy to take an application that was currently single threaded running in somewhat of a slow fashion, multi-thread it, give it a more wide range approach, and get it to perform a lot better. As well as performance improvements in the underlying system which made it so that, not only does the code that you write run better as is, but also give you other APIs to help you improve performance. The fourth was universality. So dynamic languages even since Java 6 have been supported, so we're talking languages like jruby, jython, clojure.

And the goal was that, so we previously supported it, but we want them to execute, perform, run better so that users of those languages have a complete option to say, you know what, I want to mix Java, I want to mix jython, I want to mix jruby, and we get the good performance that makes it so that it's a benefit to them to do so. The final one was integration. So developers have always asked for tighter integrated, or somewhat tighter linking to the operating system. So Java 7 introduces a new file system API that's much improved, and we're going to discuss this a little later. But it gives you ways to kind of integrate further with the operating system so that you can manage some of the file system aspects, the directory lookups, the traversing of directories. Make it so that you can integrate better with the system that you're running on.

So let's jump right in to some of the changes that were in the code, and these are really minor changes, but little quirks that kind of make it easier from a developer perspective. So first is the strings in the switch statement. So, previously developers had to use integers or Boolean values as part of the switch statement. Now you can use the string value itself, and it can be utilized as part of the cast statement. And this is kind of a big thing. So before we have to set a value or set it to a Boolean value or set it to some variable that we're going to use in a switch statement, get the case statements to key on that, and then essentially reassign the variable. Now it's all part of the string and you can just pass the string and move on. The second one was type inference. So we have something called the Diamond Operator in JDK 7, which really reduces your extra typing while initialization. So it not only reduces your code but ensures the type checking. So here you can see we have Map and we have String,My type, foo, and we're setting up a new map, but then again, we're redefining that it's string and the type that it is. The new way is using the Diamond, which is the less than greater than sign together. We just now specify a new map. Again, less coding—it makes it less chance for error.

So, better integral literals. We've always had decimal, hexadecimal and octal literals, but Java 7 introduced the binary literal, so now you can actually have the 0b prefix and specify a binary number. Another thing is the underscores. So there was the introduction of underscore as a digital separator, and you can see in this example, so the first number is 34, 409,000. It's very easy to read because the underscores are separating where numbers would logically be separated. So this improves readability and can really limit some simple mistakes. So every number in that set there, even the final one which is 1,234, are valid numbers, just simply separated by the underscores. Another key thing is the simplified Varargs method implication. So since Java 7, developers have been able to write a method capable of accepting a variable number of parameters and allow the compiler to essentially package that list into an array. So, utilizing @SafeVarargs annotation, it's really a simple way of suppressing all the unavoidable warnings associated with Varargs. So when you set up Varargs, you can—when you do your compilation you can get a list of warnings because you just don't know exactly what's going to come up and you want to make sure that everything is safe and type safe. So by simplifying it and making this annotation @ into the SafeVarargs, it really cleans up the output so a developer can identify and focus on the real problems. They're not focused on a set of minor things that are out there, and they can, from an output perspective, they can easily say, “I’m eliminating these warnings so I can focus on what the real problem is in my application.”

So let’s talk about Automatic Resource Management. One of the leading sources of failures in production applications, honestly, is the mismanagement of resources. So more server restarts occur due to an abundance of either database connections or file descriptors, or anything remaining open that’s utilizing memory and just taking functional space of the application that causes the actual application to crash or run out of memory. So in Java 7, Automatic Resource Management was introduced. This new construct really, it extends the try block to declare resources much like a four loop. So when you look at a four loop you have I equals this, phy is greater than this, do this, and at the end that’s all cleaned up. So now the resource that’s declared within the try block in this opening scenario was automatically closed by the system, so the new construct really shields you from having to pair try blocks with corresponding finally blocks. Typically you would do try, you would open up your file, you’d open up your database connection. You’d have a catch to catch exceptions, you’d have a finally block to then do the final close-out, depending on what happens in your application. Now by actually putting it inside of the try method, you’re letting the server be responsible for closing the connections. It’s a much safer and a much more reliable approach, because there’s so many scenarios where you could forget to close or you could close a connection incorrectly. And even from a troubleshooting perspective, it can be a nightmare when you’re managing these connections yourself.

So the Multi-Catch. This is really somewhat of a self-explanatory enhancement. But in Java 7 you can catch more than one exception in a single catch phrase. So traditionally developers would have several catch clauses to handle individual exceptions only to then call the state method. So, example, I’m catching a null point or exception, I’m catching, you know, any number of exceptions, and let’s say, in any case, I still want to send it to function A or function B—it really doesn’t matter. Now you can actually put them all into one catch phrase to say, “catch this or this or this,” and then call that one function that you were going to call in any case. Again, this is designed to save the amount of code that you write, and just make it easier from a developer perspective.

So, the NIO.2 filesystem API. This really associates the notion of metadata with attributes and provides access to them through the Java and I/O file attributes package. So what it does is it provides methods, essentially. So these are files class supervised methods to allow you to check if a path is readable, if it’s writeable, if it’s executable, whether it’s a hidden path. So these checks enable you to determine what kind of file or directory you’re dealing with before you try to apply any operations such as read or write. So again, this is really tying you more to the underlying system and giving you access to parse the file system, look through the file system, trigger certain events. So another key to it is the file visitor interface, and it really provides support for recursively traversing a file tree. So it allows you to represent key points in the traversal process. You can go through and see what files have been visited. You can go through an entire directory. You can traverse the thing in a recursive manner to see exactly what’s happening. Another key API is the WatchService API. So this was introduced in Java

7, obviously, and it's really a thread safe service that's capable of watching objects for changing events. So it's more of a file system monitor or directory monitor or a file monitor. So, for example, think of something like Notepad. So you can be editing a file in Notepad, but you can also go into the operating system itself and modify that same file. Now, when you go back into Notepad you'll get that notification to pop-up and say, you know what, someone has modified this file outside of Notepad, do you want to refresh it with the updated code. We see this a lot when looking at log data. And again, it's the same idea. It's that we can watch a file, we can take events based on when things occur, and we can handle that as part of the Java application.

So Java.util.concurrent updates. Multicore processors and multi-threaded applications have really become increasingly prevalent in the Java community. I mean, why not? Hadoop using that simple divide and conquer approach has been very successful, and in Java 7 they're really using a more fort join framework capable of better managing multiple thread applications. And the goal, again, is to take advantage of the underlying system—take advantage of these multicore processors. You have a lot more opportunities to utilize more threads, make your application run better and bring it back together at the end so that it performs quicker. So two of the more popular enhancements are the TransferQueue and the Phaser. So the TransferQueue, it's similar to a traditional queue approach with really one caveat. So, instead of dropping a thread on a queue and just letting it wait there and assuming at some point another thread's going to come and pick it up, TransferQueue essentially performs a real handoff. So it puts it on the queue but it really waits until some other thread comes along to take it and move forward. So now you can trigger an event based on, okay, the handoff has occurred, I know it's been passed out to another thread and I can move forward. The Phaser enhancement, it really provides a very flexible, easy to use approach for synchronizing multiple processing threads before moving on to the next task. So when you're looking at a (new?) type approach or a divide and conquer type approach, you have a bunch of threads that have now been sent out to do their individual tasks, but at the end you have to have a simply way of bringing them all together, guaranteeing that all the responses are back, synchronizing the response and then moving forward. And that's really where the Phaser approach comes in.

So, invokedynamic. Jython, jrudy and other dynamic type languages are still very, very popular in the development community. So Java 7 embraces the dynamic type languages, and really this JSR 292 allows an easier mix and match approach for those dynamic and static languages in the JVM. So what it does it, now we've made it so that the dynamic type languages will function and function very efficiently inside of Java 7. So it makes it easy for developers to say, you know what, maybe I can do this better in jython or maybe I can do this better in jruby. I still want to utilize my Java application but I want to be able to integrate between the two. So it makes it very easy for a developer to say, you know what, I'm going to pick the right language for the job and I'm going to utilize that. I'm going to use Java for this portion because it's most efficient, I'm going to use jython here. Whatever you can use and whatever makes it more efficient from a developer perspective is completely acceptable now because enhancements have been made so that they perform much, much better.

All right, so let’s get into Java EE6, and now we’re going to get into more of the service side components and some of the new capabilities that are included in the Enterprise Edition. Now, before we jump into that, there’s three quotes that I have here, and I usually take these—in some cases I take them out of the slides, but I think it’s important because this is kind of my developer coding rant. So the first one says, “Measuring programming process by lines of code is like measuring aircraft building process by weight.” And this is actually a pretty key one, and it’s the old thing that hey, I’ve written an application that is a million lines of code—it must be very elaborate and a great application. And that actually goes against what Java 7 and JEE6 are all about. The goal is to streamline the code. Just because you have millions of lines of code doesn’t necessarily mean you have good code. It could be that you’re limited in reuse or that you’re doing things in an inefficient manner. So don’t look at code by how many lines you have or how many files you have or how broken apart you’ve made it, look at it by how efficient you can make it utilizing some of these new APIs. The second one says, “Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.” And this is a big one. As an IBMer, and I do a lot of proof of concepts where I go in and I either write the code or I integrate with code from other companies, it’s very hard to pick up where somebody left off if they didn’t use some standard Java practices, whether it’s commenting out properly, whether it’s using a common set of methods, whether it’s modularizing the code so that it’s easy to use and you’re getting a lot of reuse. Always consider, how can I make it a little more efficient, how can I make it so that it runs a little better, and how can I make it easy to pass onto somebody else, because that’s going to help you out both when you’re maintaining your own code, but also if you have to pass it off onto somebody else. And the last one says, “Good code is its own best documentation. As you’re about to add a comment, ask yourself how can I improve the code so that this comment isn’t needed.” And this is actually a big one, and the reason is, when you initially start to write an application everybody’s gung ho and they put in all the comments they need and they really specify, okay, this is what this function does, this is what this function goes. The problem is, as future enhancements come in and a code changes are necessary, the comments themselves typically don’t get updated. So it really makes more sense to take your application, use variables that make sense. Use method names that really make sense to a developer. So if you’re doing a car lookup, for example, make the function, hey, this is car lookup function. The data you’re returning or if you’re looping through a set of data coming back, instead of it using the standard 4i equals this and looping through and your variable name is “I” use a variable name like car name or car VIN or whatever it is of the value that you’re actually pulling back so that an outside person can go into your code and completely understand just by following the logic exactly what’s happening. So that’s the end of my developer’s rant.

All right, so let’s get into the Java EE enhancement themes. So, where have we come from and kind of where are we headed? So the Java community, again, it started in Java EE5, or actually, let’s get a little bit before that. So in Java EE, in the previous versions, 2, 3, 4, the idea was, let’s put as much functionality as possible out there. Let’s get everything that developers need, let’s put it in the product and make it so that it’s completely available. Starting in Java EE7 what the goal was is that, okay, we’ve got the functionality—sorry, in Java EE5 and in 6—we’ve got the functionality the developers need. What we want to do is, let’s make it easier to use. We know that they can function, so let’s make it so that they can develop quicker, let’s make it so they can write less code, let’s bring in some things, let’s deprecate some functionality. So you can see, we started to deprecate JAX-RPC, some of the EJ entity beans, JAXR. Now, these are still in the product, but know that they’re kind of slowly on their way out because they’re just not the same use cases that they once were. But again, the real enhancement themes were, I want to make it so that developers can pick up Java quickly, can utilize the advanced functionality with a limited amount of code, and really get real benefits performance-wise.

So let’s start off with annotations. So these annotations have really been further enhanced in JEE6. In the service space developers cannot use the @WebServlet annotation to really dynamically map the service. So when you look at this example here, traditionally, servers were mapped in the web.xml file and were somewhat static in nature. So you can see in this XML snippet, we have the servlet name, we’re giving it the URL pattern. This is all part of the web.xml and it’s really static as part of the application. Now, the web.xml file, it’s still implemented, and in fact would still override any WebServlet annotations. However, from an ease of use perspective, dynamic servlet mapping via annotations is really a huge benefit to the development community. So if you look at the example at the bottom, you can now see we specify the @WebServlet annotation, we give it the name, we give it the URL pattern. And this is all done directly in the code itself. So we no longer have to modify the web.xml file every time we want to add in a servlet. The developer themselves can use an annotation and dynamically add it to their code. They compile it, they build their code, and their servlet is now available. Again, it’s really designed around ease of use. And this isn’t something that’s just for developers—this can certainly be used in production. It’s really just an easier way to map your servlets.

So let’s get into the programmatic configuration. So now, along with dynamic servlet mapping, servlets can also essentially be added dynamically. So in this case the server, it’s not initialized until the status parameter, in this example here—until that status parameter is equal to VIP for a specific call. So one thing to know is that once a servlet is added it’s going to remain active until the JVM is stopped, but it’s a potential resource saver and really an additional security layer. So why have a servlet up and running that in many cases may not be used at all unless a very specific event occurs. So in this case what we’re doing is we’re doing a simple check, and if it turns out that, hey, a VIP comes in, or it really can be any parameter that you specify. When that event occurs then we’re going to add that servlet in. So in this case the VIP servlet. So until then the servlet essentially remains dormant and is inaccessible.

Alright, asynchronous servlets. Now we’re going to start to get into the idea of, okay, how are we going to take advantage of multi-threading? How are we going to make it so that our applications can perform a little bit better, especially for some of these long running calls. So in the Java Server 3 specification there was added support for asynchronous processing. So now what happens is a servlet no longer really has to wait for a resource response. So, imagine a database call. I'm going through a servlet, I'm making a database call. Typically I would wait for that response to come back before I could continue processing. So in that case the thread is essentially blocked until you get the response. So using asynchronous servlets, that thread essentially is no longer blocked. Now, there was support for asynchronous servlets prior to the servlet 3.0 specification, but it was proprietary in nature. So now there's a new standardized approach which is universal in all servlet 3.0 compliant JVMs. And again, it makes it so that I can go out and I can send out a request inside of my servlet, I can have it make a call—I mean, there's a couple of great use cases here whether it's quality of service or a chat instance. So I'm sending out a response and I'm waiting for some other event to occur, waiting for a response to come back. Even accessing REST services. So I can make a call to the REST service, meanwhile process the rest of the servlet and when I get the response back, just refresh the page. It's really designed to take advantage of the improvements in the system and the performance of the underlying system.

So what is REST? So REST is really—you know, what I want to say is, why is it important? REST is one of the most popular data transfer of calls in use today, especially in the mobile space. And I would venture to say that in mobility, 75% of the requests and the responses occur using the REST protocol. REST really relies on three easy things. One is a verb describing what action is going to occur. So here, it's whether to get a post, a put, a delete. A second is the noun. So where am I actually going to be sending the request? So what's the URI? And the third part is an adjective. So what's going to be returned? What's the document that I'm going to get back and how am I going to manage it? So it's a very simple protocol to implement, and although this is a very simplistic example, you can see how a REST call compared to a SOAP call is a much smaller request. So instead of creating the envelope just to submit this one simple user ID, I can make a very quick call with the URL and just pass in the parameter. And using (Jsome?) as the data structure for the response produces not only a smaller response, but it can be very quickly parsed, especially when you're talking about mobile devices. So I'm not here to really advocate one protocol or the other. I know there's plenty of SOAP users or plenty of XML users and there's just as many REST users. I'm just really trying to give a background of REST because the next thing we're going to cover is JAX-RS.

So talking about JAX-RS. So a new feature in Java EE6 is a set of APIs capable of creating RESTful services. So now it makes it, inside of your Java code, easy to utilize REST services, REST APIs, and any services that you have out there. And as you've seen in previous new Java features, annotations are very highly utilized in the JAX-RS programming model. The annotations can be used to not only declare resources, but also specify the entire RESTful contract. So, the real focus of JRS 3.11 is to foster the use of REST services in a variety of application scenarios. So while REST is being put out there and being available to other users and being a primary way of communication, Java is taking that to heart and has kind of made it that, you know, we're going to make it easier for you to utilize REST services.

So let's look at a quick example. So this example just further demonstrates what JAX-RS is all about. As you can specify, we're still specifying the three vital parts of the REST service. So we have our nouns, which is our URIs, and they're all specified using the @Path annotation. So in this case we have orders, another one we have an actual variable. Your methods are being specified, and in this case we have the @Get annotation and we have the @Post annotation to tell you which type of method we're going to use, and these are your verbs as part of your REST service. And then the last part is we have the document format, which is your adjectives. So in this case you have what consumes—that Post method, okay, it consumes application XML. And we also have what's produced by the Get method, which is application XML as well. So in the end you have a very simple precise way of utilizing REST services directly from within the Java code itself. And again, what it was is REST, as it became more and more popular, the powers that be at Java realized, you know what, we need to further enhance that and make it so that it's very easy from a Java perspective to code for REST services.

Alright, so let's talk a little bit about context and dependency injection. So dependency injection, it's been around for a while. It was inspired by SEAM, Google Juice, (Spring?)—a lot of others use it and it's really designed to promote loose coupling. So dependency injection in its most basic level, it refers to the process of supplying essentially external dependencies to a software component. So it aids in design by interface by really… Okay, a good example would be, I have JNDI. So in a current JNDI scenario I would do a lookup to pull JNDI data and say, okay, this is the database information I need. So, instead of looking up the database connection with JNDI, with dependency injection you could inject that database information directly into the code itself. So it's more—it's almost the opposite way of handling it. Instead of a, "I'm going to go out and look for it," it's a "We're going to go ahead and give it to you," type approach. So the enhancement, though, with JEE6 is the idea of context and dependency injection. So this is a new Java EE6 specification, which not only defines a powerful type safe dependency injection, but it also really introduces the concept of contextual references. So the CDI means they live in a defined scope which is what makes them different from just the standard dependency injection. They're created and destroyed on demand by the container, and they come with four predefined scopes. So here you can see we have request scope, session scope, application scope and the conversation scope. So it's a way of using dependency injection but also wrapping a context around it. And what makes this important is it assists with the cleanup. So now you don't have to manually get rid of certain things because now they're tied to a very specific context that allows the cleanup and the management of those aspects much easier from a developer perspective.

So defining a CDI Bean. A lot of times you just take a very simple POJO a plain old Java object, and what does it take to make it so that it could be utilized as a CDI Bean? And it's really four very simple things. It's really just opting in by deploying inside of a Bean archive. You just must have a no argument constructor, or you can have a constructor annotated with the inject object, which you can see we have in our code sample here. It has to be a concrete class, unless you're using Decorator, and it just must not be a non-static inner class. So it's a very small set of rules that make it very easy for you to take a simple POJO and make it a CDI Bean.

Okay, so EJBs. EJBs, they've been popular then they kind of went to the background then they became popular again. So right now I'm not sure if they're on the upswing or the downswing. But the EJB 3.x specification was a big enhancement. And the real underlying concept is, it centers on plain old Java objects and that programming model. But using Java annotations to capture the information that deployment descriptors previously maintained—in fact, I mean, in today's world, deployment descriptors are now optional in most cases. So using default values literally also means that we no longer have to set these default values. We don't have to sit there and go through and create things. So you can see on the left we have this public void EJB active, public void EJB passivate, things that now are considered default variables that we no longer have to specify. We only need to make modifications if we decide to go outside of the defaults. And it really speaks, again, to the core values of JEE6. We want to improve the functionality, we want to make it easier for users to use, and we want to take better advantage of annotations so that from a coding perspective we have a lot more checking that's done by the system itself, and it makes it easier for you to use EJBs, use these capabilities without having to program so much. And you can look, just by looking at the code itself, you can see how EJB 2.1 code and 3.x code, you can see the general benefits, the use of annotations, the less amount of coding. You're not having to extend and throw and catch some exceptions. It's just an easier way of doing it and it makes it much easier from a developer perspective to utilize EJBs and get into them a lot quicker.

So Singleton session beans. So, a new feature in EJB 3.1 was the idea of Singleton session beans, and this is really a new type of session bean in addition to stateless and stateful. So one of the big things about Singleton session beans is it kind of gives you a way from within the JVM to cache data, because what you're doing is you have one single instance, it's per JVM, it's started up with the JVM is started up, it stops when the JVM is stopped, and it gives you a bean that can be accessible by all the other applications running the JVM. So think of a scenario for a Singleton session bean where maybe initially you load pricing data or you load some set of data that all the other threads may need at some point, but you only want one copy—essentially a static copy that's going to be used by all the other environments or all the other threads as they come in. So there's a couple of things that are key to it. Number one is guarantee that there's only going to be one single instance of this being per JVM. The initialization is going to occur during the startup and it's going to be destroyed when the application is closed. And since there's only a single instance in each JVM it's important that the implementation handles the concurrent method calls, because again, lots of different methods are going to be called into this because it's a single instance containing a single set of data essentially giving you an in-memory cache that's JVM specific. So it's a very powerful tool and it's a very useful tool, you just need to maintain that when you make the calls to it that you don't look at it as a single point of contention.

So asynchronous bean invocations. So the EJB specification now has a mechanism to declare that a business method should run asynchronously to the client, and it doesn't matter if it's local or remote. So the purpose again of this functionality is to improve performance by pushing requests out of multiple threads. We don't need to have everything running on the same thread, we don't need to run everything single threaded. And there's really two ways of doing this. First is the fire and forget. So with fire and forget, the asynchronous method is declared and what you get back is a void return type. So the application client really has no way of knowing when or even if that method completed. So this would be useful for scenarios where there's some really nice to have function but it's not really important if it's completes. So using an asynchronous request in this scenario, we're going to send it off, we're going to kick it out, we're going to say, okay, you know what, run this request. I don't really care if you come back, I just need to make sure that I sent the request off. The second is really the fire and return results, and in most cases this is the most popular one. And the reason is, is because this asynchronous method is declared to have a return type. So when you click it off you're going to immediately get the return type prior to even it kicking off of the asynchronous method. So then it allows you, using that return, using that method, to say, okay, I need to check the status of this request, and eventually I want to obtain the result. So kind of an easy way of explaining this is, think of a scenario where I'm telling my kid to go clean his room. If I use that fire and forget method, I don't really care if he cleans his room or not. You know, I've sent off the request, I've told him to do it, I'm watching football now, he's off doing something. I really have no way of finding out did he really do it or not. All I know is that he's gone and he went and did something. If I use the fire and return results method, I can check his progress while he's going through it, and I can find out when he's done. I can be notified, okay, he's finished. He's actually doing what I told him to do and he did the work. So it's really two different methods and two different ways of kind of handling asynchronous beans.

So the embeddable EJB container. This is a big thing for developers, and the reason is, typically with EJBs it's a pain to test because from a local machine I need to mock up a bunch of services. If I want to run it on my laptop, I need to mock up services, I need to make the calls, and I need to add in a whole bunch of functionality just to test these simple EJBs. So one of the big ease of use enhancements was to introduce the embeddable EJB container. So this is really, again, targeted to developers. It's not designed for a production level EJB container, but it makes it very easy to, if I'm within your code, actually have the services available so that you can run, test, validate your EJBs without having to hold back in-service set up and configured. So looking at a quick code sample, you can see that what I'm doing here is I have an EJB container, I'm creating it, I'm giving it the provider, I'm giving it my app name, I'm giving it my modules. I'm actually creating the container based on those properties, and then I can do a lookup for that container. And again, this is all built within the JVM and in the application itself so that a developer running on his own machine can utilize this and get the full capabilities. It makes it very easy to test, and from a developer productivity perspective, this is a huge benefit.

So JPA 2.0 highlights. And I'm not going to spend a lot of time on this. There's just a couple of quick things that I wanted to cover. And it really is around standardization. And you can see the first four bullet points: Standard mapping enhancements, standard query hints, standardized pessimistic locking. The idea was take JPA and in JPA 2 we want to standardize everything in JPA and make it so that regardless of the JVM you're running in, there's a standard approach that we're taking.

Java EE Bean validation. Bean validation is a very simple way of validating your code using annotations. So, a validation constraint is really a decorative condition that the Java Bean you've defined must satisfy to consider to be valid. So in this sample here you'll see we have @NotNull which is a built in constraint to validate that the variable state, it can't be null. So this is going to fail if that value is null. And all we're doing is we're putting an annotation in front of it to say, you know what, I want to make sure that these certain conditions are met as part of my bean, otherwise it's an invalid bean and I don't want to proceed. So annotations, again, come into play here and make it very easy from a developer perspective to kind of—what you could say is, pre-check your code as part of your application development by putting in simple annotations that will allow you to validate events before they occur.

So let's talk about OSGI. So what is OSGI? First of all, OSGI is not a new technology. It's actually a mature technology and used by virtually every Java product. So IBM WebSphere, Oracle, WebLogic, JBoss, Eclipse—they all use OSGI because it's such a benefit, because it allows you to tie specific classes—it allows you to modularize your application. It allows you to kind of hardwire, or dynamically hardwire, what an application needs and what it utilizes to get it to function properly. So you're really allowed to explicitly declare what should and will be exposed from a Java file and explicitly what version, but also declare what's required by a Java file. So this eliminates a lot of class loading issues because now we're actually taking and saying, you know what, instead of throwing a bunch of JAR files inside of the library and just saying that when I call Class Steve, I'm going to pull whichever one happens to pop up first. I'm now specifically saying that I need this version of this JAR or this package so that I can really hardwire and say this is exactly what I need for this application to run properly. So how does it actually help? So there's a couple of things. Number one, it puts essentially a brick wall around the JAR files and you create a bundle that says this is what this particular package, this bundle's going to export, and this is what it requires to run. So now from a class loading perspective, I now have the OSGI container managing what I need. I don't need to worry as much about, is it parent last, is it parent first, am I using a shared library, because I'm specifically telling what I give out to other applications and what I need for my application to run. It enables modular deployment, again, because now each bundle is required to use a specific package. So I can deploy any number of versions of a specific package and not worry about an older application accidentally using a newer version of code just because I've released a newer package of that. And to that end it really enables coexistence. So let's say I have a class or a package called Steve. I could have Steve version 1, Steve version 1.1, 1.2, 1.3, and then through OSGI I can specify and say, you know what, this particular application must use Steve version 1.1 or less, so 1.1 or 1.0. Or I could say, this application requires explicitly version 1.3. So it makes it so that from an OSGI perspective it'll say, okay, I understand exactly what you need; I'm going to find what you need and I'm going to map that dynamically so that when you're running, when you need that part of it, I'm going to find it, I'm going to give it to you. I don't have to worry about class loading from JVM startup, I don't have to worry about where the JAR file is, whether I have included the library inside of the application. It's a very simple way and a precise way of mapping applications to the exact needs.

So let's talk a little bit about the bundles and the class loading itself. So, OSGI, it really is a module system for Java, but the key notion of it is the bundle. And the bundle is, I'm going to specify certain things. So I'm going to specify the bundle version, and there can be any number of versions. I'm going to explicitly tell you, okay, here's my import package. So what packages from other bundles does this bundle depend on? What do I need? So in this case, in this simple example here, I have com dot something dot I dot need version 1.1 and .2. And then I'm also going to explicitly tell you, what am I going to make visible to everybody else? So when you think about it from a Java perspective, there's private classes and there's public classes, so what am I going to make public so that other resources and other bundles out there can utilize? So in this case I'm using com.myservice.api version 1.0.0. Now this, again, makes it very easy from a class loading perspective because each bundle has its own loader. There's not just one standard class path that everyone's using. And the framework really decides and manages the dependencies for me. So I don't need to do as much management as far as, how do I deploy the app, what shared library do I want to use. How do I want to combine it so that I guarantee I'm using the exact version that I want.

Well, that's it from a presentation perspective. I think we're going to open up the line for questions now, but I did want to let you know that this is the first part of a three-part series. This part covered the Java Standards, Java EE6, Java 7, OSGI. The second section is going to cover WebSphere version 8.5.5, the capabilities of it and the Liberty profile. And our third session is going to cover Worklight mobility programming—how you can take common applications and get them to run in a mobile environment. So stay tuned for those. We're going to send out more information. But let's open it up for questions. Thank you.

Michele Choate: Alright. Thanks very much Steve for that overview and an update on the main industry standards that a Java developer needs to know about. I think you guys probably see down in the bottom left-hand corner there is a green button—a green Q&A button that you can press if you have a question that you want to ask of Steve. So if you would do that now, that would be great. And then right next to that, to the right of that, is a Content button, and you can click on that and get a copy of the presentation that Steve went through. You can download that. And there's a couple of other things out there. The latest WebSphere Application Server announcement letter. We had an announcement at our Impact event in April for WAS, WebSphere Application Server 8.5.5, which is what we'll cover in the next broadcast. But you can get to the announcement letter and a presentation about WebSphere Application Server 8.5.5 from that content button and download either one of those. And while we wait, we have a couple polling questions that we'd like to ask. The first one is—if we can go ahead and roll those out. The first one then is, what version of the Java Specification are you running in production now. And you can click Multiple Versions if you want. So, back to J2EE 1.4, JEE5 or 6 or maybe you don't know or maybe you aren't using Java yet. So if you want to answer that and we'll put up the results of the attendees that are on the line. Go ahead and click on that now. While we're waiting for the results of that, I just wanted to also remind you that this has been recorded. This session, we've been recording it, and so it will be available on demand. So if you want to listen to it again or if you want to let your colleagues know about it, it will be available for 12 months for replay. So you can use the same link that you used today to dial in or you can use the link that you used to register to get to that. Alright, so that's where everyone is. That's good. Great to see. Alright, and as we wait for the next question, why don't we put out the next polling question which has to do with OSGI and just whether or not you're using OSGI or not. We use, in case you don't know, we use OSGI internally inside WebSphere Application Server, and Solomon, who will be our presenter for next week, will be talking about how we take advantage of OSGI in our Liberty profile to make it a very compact and efficient download. Well, not download, but Application Server. And so it only loads the pieces of the Application Server that you need, so it makes it fast to restart and simple to use. So let's see what our poll is now on OSGI and who's using it. You're all using it, actually, if you're using any of the later versions of WebSphere Application Server, but we're really asking whether or not you're using it in your applications. So that's great, 23% are using it, so that's good. Steve, I think we have one question that's come in, and you guys, just a reminder, if you want to click on the green button that says Q&A, we'll get to your questions as they come in. One question that has come in is, what is the level of effort to migrate to JRE7 or JEE and JEE6. So Steve, could you give us a feel for that?

Steve Mirman: Yeah, and that question, I mean, it really depends on how the application's written. But JRE7 and JEE6 were designed to be as backward compatible as possible. So typically, even if you had something as old as 1.4 or JEE5, there's a real good chance that that application will still run. It may not be as efficient as if you had written it with some of the newer technologies, but it should run and still be effective on a JRE7 or JEE6.

Michele Choate: Alright, good. And then the other thing that I just want to mention is that the WebSphere Application Server migration toolkit is available for you to use to migrate from versions to versions that kind of helps you do that migration. And then there's—actually, we have a lot of information out on the web if you want to Google WebSphere Application Server Migration, you'll see a landing page that has some great information for WebSphere. And I know that maybe not all are using WebSphere Application Server, but that's some information we have out there. Alright, I think that is the last of our questions, and we're coming up in the top of the hour. I just want to also remind you that you will get a follow-up email after this call, after this webcast is over, with the link to the live Developer Day events. We've been doing these Developer Day events across North America, and the next live ones will be—there's five of them that we're working on. San Francisco, July 30th, Chicago, August 20th—that s tentative—

Cleveland, September 12th, Philadelphia, September 17th—that's tentative—and Nashville, September 24th. So, please, if you're in any of those cities or have colleagues in any of those cities, let them know. And we'll, in our follow-up email, we'll give you a link to register or you can go to Be Watching to register for those as they firm up. Okay, well, Steve and I just want to thank you very much for joining our webcast today. I hope you found Steve's presentation of value and I hope you join us again next week, next Wednesday at this same time for an overview of WebSphere Application Server 8.5.5 and how it implements the standards that we just talked about. Thanks again.

[END PRESENTATION]