



Enterprise Computing with Java

MCA-305 UNIT III

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason

U III. 1



Learning Objectives

- What is EJB, Motivation for EJB.
- Advantages and applications of EJB?
- EJB Echo system basics and components.
- J2EE technologies(different J2EE technologies)
- Enterprise beans types and its uses.
- Distributed objects and middleware.
- Developing EJB components.
- Remote local and home interface.
- Bean class and deployment descriptor.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason

U III. 2




Introduction

- The EJB technology depends on other Java technologies to function properly.
- First, it uses Java Remote Method Invocation (RMI) as the communication protocol between two enterprise beans and between an enterprise bean and its client. RMI is used in Java-distributed computing to invoke remote methods on a remote machine.
- Another technology used in EJB is Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP), where ORB stands for Object Request Broker. RMI-IIOP is a more portable version of RMI that can use the IIOP from the Object Management Group (OMG). RMI-IIOP is especially used in communications between an enterprise bean and a client.
- Lastly, EJB uses Java Naming and Directory Interface (JNDI) as the naming service that binds a name with an enterprise bean.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason


U III. 3



What is an Enterprise Java Bean?

- In a nutshell, an enterprise JavaBean is a Java-based server-side component. Just like any other component, an enterprise bean encapsulates business logic of an application.
- Enterprise beans must conform to the EJB specifications, however, and they are deployed and can run only in an EJB container, identical to the way servlets run inside a servlet container.
- A servlet container provides services for servlets, such as session management and security.
- Likewise, an EJB container provides system-level services for EJB applications. In fact, as you will soon find out, it's the EJB container that makes EJB so great.
- Despite the similarity of names, EJB has little to do with JavaBeans


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. .#



Benefits of EJB

- EJB applications are much more complex and more difficult to build and administer than servlets/JSP applications.
- For someone new to EJB, the learning curve also is steeper. If this is the case, why is EJB so popular and why do so many organizations want to invest in it? The answer is simple—after you know the nuts and bolts of EJB, writing an application is an easy task, and, more importantly, you can enjoy some benefits provided for you by the EJB container.
- The following is the list of some of the benefits of EJB:
 - EJB applications are easy to develop because the application developer can concentrate on the business logic. At the same time, the developer uses the services provided by the EJB container, such as transactions and connection pooling. Again, the hardest part is the learning process.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. .#



EJB Benefits (Contd.)

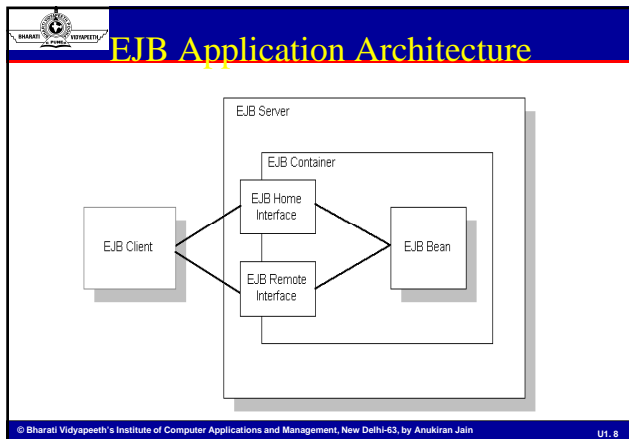
- EJBs are components. Chances are good that there are EJB vendors who sell components that encapsulate the functionality that you need. By purchasing a third-party's EJBs, you can avoid needing to develop your own beans, which means your application development is more rapid. The EJB specification makes sure that beans developed by others can be used in your application.
- There is a clear separation of labor in the development, deployment, and administration of an EJB application. This makes the development and deployment process even faster.
- The EJB container manages transactions, state management details, multithreading, connection pooling, and other low-level APIs without you, the developer, having to understand them.
- The EJB container provides security for the applications.
- The EJB architecture is compatible with other Java APIs.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. .#

When to use EJB?

- An enterprise java bean can be used if an application has any of the following requirement:
 - The application must be scalable. To accommodate a growing number of users, one may need to distribute an application's components across multiple machines, but their location will remain transparent to the clients.
 - Transactions are required to ensure data integrity. Enterprise beans support transactions, the mechanisms that manage the concurrent access of shared objects.
 - The application will have a variety of clients. With just a few lines of code, remote clients can easily locate enterprise beans. These clients can be thin, various and numerous.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 6



EJB Architecture

- The Enterprise JavaBeans (EJB) technology defines a model for the development and deployment of reusable Java server components, called **EJB components**. An **EJB component** is a non-visual server component with methods that typically provide business logic in distributed applications.
- A **remote client**, called an EJB client, can invoke these methods, which typically results in database updates.
- EJB server** The EJB server contains the EJB container, which provides the services required by the EJB component. EAServer is an EJB server.
- EJB client** An EJB client usually provides the user-interface logic on a client machine. The EJB client makes calls to remote EJB components on a server and needs to know how to find the EJB server and how to interact with the EJB components. An EJB component can act as an EJB client by calling methods in another EJB component.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 6



EJB Architecture

- An EJB client does not communicate directly with an EJB component. The container provides proxy objects that implement the components home and remote interfaces. The component's **remote interface** defines the business methods that can be called by the client. The client calls the **home interface** methods to create and destroy proxies for the remote interface.
- **EJB container** The EJB specification defines a container as the environment in which one or more EJB components execute. The container provides the infrastructure required to run distributed components, allowing client and component developers to focus on programming business logic, and not system-level code.
- **EJB component implementation** The Java class that runs in the server implements the bean's business logic. The class must implement the remote interface methods and additional methods for lifecycle management.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 4



Motivation for EJB

- **EJB Advantages**
 1. The underpinning EJB specification
 2. Integration with the J2EE platform
 3. Almost transparent scalability
 4. Free access and usage of complex resources
 5. A strong and vibrant industry and community
- **EJB Disadvantages**
 1. Large, complicated specification
 2. Increased development time
 3. Added complexity compared to straight Java classes
 4. Potential to produce a more complex and costly solution than is necessary
 5. Continual specification revisions


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 4



The EJB EchoSystem

- To have an EJB deployment up and running, one needs more than an application server and components. There are six more parties involved:
 1. The Bean provider: The bean provider supplies the business components to the enterprise applications. These business components are not complete applications but can be combined to form complete enterprise applications. These bean providers could be an external provider selling components or an internal component provider.
 2. The Application Assembler: The application assembler is responsible for integrating the components. This party writes applications to combine components so as to develop the target application that can be deployed under various environments.
 3. The EJB Deployer: After the application developer builds the application, the application must be deployed on the server. This involves configuring the security parameter settings, performance tuning, etc. An application assembler is not familiar with these issues. This is where the EJB deployer comes into play.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 4



The EJB Echo System

- The System Administrator: The system administrator is responsible for the upkeep and monitoring of the deployed system and may make use of monitoring and management tools to closely observe the deployed system.
- The Container and Server providers: The container provider supplies the EJB container (an application server). This is the runtime environment where the beans live. The container supplies the middleware services to the beans and manages them. Some of the various containers are: BEA's WebLogic, iPlanet's iPlanet Application Server, IBM's WebSphere, Oracle's Oracle 9i Application Server and Oracle 10g Application Server, and the JBoss open source Application Server. The server provider is the same as the container provider.
- The Tool Vendors: There are various IDEs available to assist the developer in rapidly building and debugging components, for example Eclipse, NetBeans, and JBuilder. For the modeling of components one can use Rational Rose. There are many other tools, some used for testing (JUnit) and others used for building (Ant, XDoclet).


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



The EJB Echo System

- Each of these parties is an expert in its own field and is responsible for a key part of a successful EJB deployment. Because each party is a specialist, the total time required to build an enterprise-class deployment is significantly reduced. Together, these players form the *EJB Ecosystem*.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



The Application Assembler

- The application assembler is the overall application architect. This party is responsible for understanding how various components fit together and writing the applications that combine components. An application assembler may even author a few components along the way. His or her job is to build an application from those components that can be deployed in a number of settings.
- The application assembler is the *consumer of the beans supplied by the bean provider*.
- The application assembler could perform any or all of the following tasks:
 - i) From knowledge of the business problem, decide which combination of existing components and new enterprise beans are needed to provide an effective solution; in essence, plan the application assembly.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



The Application Assembler

- ii) Supply a user interface (perhaps Swing, servlet or JSP, application or applet) or a Web Service.
- iii) Write new enterprise beans to solve some problems specific to your business problem.
- iv) Write the code that calls on components supplied by bean providers.
- v) Write integration code that maps data between components supplied by different bean providers. After all, components won't magically work together to solve a business problem, especially if different parties write the components.

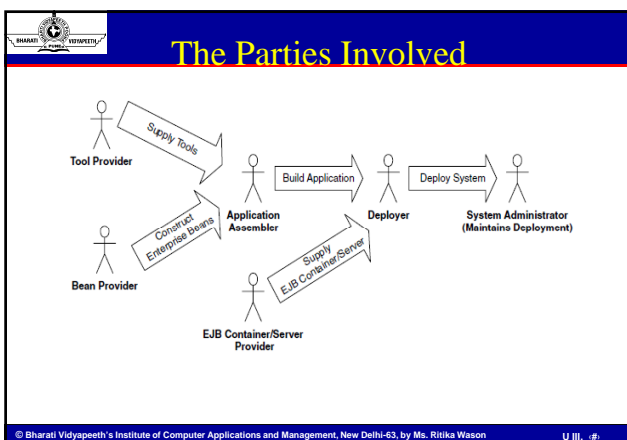
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 6




The EJB Deployer

- After the application assembler builds the application, the application must be *deployed (and go live) in a running operational environment. Some challenges* faced here include the following:
 - i) Securing the deployment with a hardware or software firewall and other protective measures.
 - ii) Integrating with enterprise security and policy repositories, which oftentimes is an LDAP server such as Sun Java System Directory Server (formerly Netscape Directory Server), Novell Directory Server, or Microsoft Active Directory.
 - iii) Choosing hardware that provides the required level of quality of service.
 - iv) Providing redundant hardware and other resources for reliability and fault tolerance.
 - v) Performance-tuning the system.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason
U III. 6






Java Beans versus Enterprise Java Beans

- JavaBeans are completely different from Enterprise JavaBeans.
- In a nutshell, JavaBeans are Java classes that have *get/set methods on them*. They are reusable Java components with properties, events, and methods (similar to Microsoft *ActiveX controls*) that can be easily wired together to create (often visual) Java applications.
- The JavaBeans framework is lightweight compared to Enterprise JavaBeans. You can use JavaBeans to assemble larger components or to build entire applications.
- JavaBeans, however, are development components and are not deployable components. You typically do not deploy a JavaBean; rather, JavaBeans help you construct larger software that is deployable. And because they cannot be deployed, JavaBeans do not need to live in a runtime environment and hence, in a container.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



Java Beans versus EJB

- Since JavaBeans are just Java classes, they do not need an application server to instantiate them, to destroy them, and to provide other services to them. An EJB application can use JavaBeans, especially when marshalling data from EJB layer to another, say to components belonging to a presentation tier or to a non-J2EE application written in Java.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49



J2EE

- EJB is only a portion of a larger offering from the Java Community Process (a.k.a. JCP—a Java industry standards body) called the Java 2 Platform, Enterprise Edition (J2EE).
- The mission of J2EE is to provide a platform-independent, portable, multiuser, secure, and standard enterprise-class platform for server-side deployments written in the Java language.
- J2EE is a specification, not a product. J2EE specifies the rules of engagement that people must agree on when writing enterprise software. Vendors then implement the J2EE specifications with their J2EE-compliant products.
- Because J2EE is a specification (meant to address the needs of many companies), it is inherently not tied to one vendor; it also supports cross-platform development. This encourages vendors to compete, yielding best-of-breed products.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50



J2EE

- It also has its downside, which is that incompatibilities between vendor products will arise—some problems due to ambiguities with specifications, other problems due to the human nature of competition.
- J2EE is one of *three different Java platforms*. Each platform is a conceptual superset of the next smaller platform.
- **The Java 2 Platform, Micro Edition (J2ME) is a development platform** for applications running on mobile Java-enabled devices, such as Phones, Palm Pilots, Pagers, set-top TV boxes, and so on. This is a restricted form of the Java language due to the inherent performance and capacity limitations of small-form-factor wireless devices.
- **The Java 2 Platform, Standard Edition (J2SE) defines a standard for** core libraries that can be used by applets, applications, J2EE applications, mobile applications, and such. These core libraries span a much wider spectrum including input/output, graphical user interface facilities, networking, and so on. This platform contains what most people use in standard Java programming.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



J2EE

- **The Java 2 Platform, Enterprise Edition (J2EE) is an umbrella standard** for Java's enterprise computing facilities. It basically bundles together technologies for a complete enterprise-class server-side development and deployment platform in Java.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49



The J2EE Technologies

- J2EE is a robust suite of middleware services that make life very easy for server-side application developers. J2EE builds on the existing technologies in the J2SE. J2EE includes support for core Java language semantics as well as various libraries (.awt, .net, .io, and so on). Because J2EE builds on J2SE, a
- J2EE-compliant product must not only implement all of J2EE, but must also implement all of J2SE. This means that building a J2EE product is an absolutely *huge undertaking*.
- *The various J2EE technologies are as under:*
- **Enterprise JavaBeans (EJB). EJB defines how server-side components** are written and provides a standard contract between components and the application servers that manage them. EJB is the cornerstone for J2EE and uses several other J2EE technologies.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50



The J2EE Technologies

- **Java API for XML RPC (JAX-RPC).** JAX-RPC is the main technology that provides support for developing Web Services on the J2EE platform. It defines two Web Service endpoint models—one based on servlet technology and another based on EJB. It also specifies a lot of runtime requirements regarding the way Web Services should be supported in a J2EE runtime. Another specification called Web Services for J2EE defines deployment requirements for Web Services and uses the JAX-RPC programming model.
- **Java Remote Method Invocation (RMI) and RMI-IIOP.** RMI is the Java language's native way to communicate between distributed objects, such as two different objects running on different machines. RMI-IIOP is an extension of RMI that can be used for CORBA integration. RMI-IIOP is the official API that we use in J2EE (not RMI).


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



The J2EE Technologies

- **Java Naming and Directory Interface (JNDI).** JNDI is used to access naming and directory systems. You use JNDI from your application code for a variety of purposes, such as connecting to EJB components or other resources across the network, or accessing user data stored in a naming service such as Microsoft Exchange or Lotus Notes.
- **Java Database Connectivity (JDBC).** JDBC is an API for accessing relational databases. The value of JDBC is that you can access any relational database using the same API.
- **Java Transaction API (JTA) and Java Transaction Service (JTS).** The JTA and JTS specifications allow for components to be bolstered with reliable transaction support.
- **Java Messaging Service (JMS).** JMS allows for your J2EE deployment to communicate using messaging. You can use messaging to communicate within your J2EE system as well as outside your J2EE system. For example, you can connect to existing message-oriented middleware (MOM) systems such as IBM MQSeries or Microsoft Message Queue (MSMQ). Messaging is an alternative paradigm to RMI-IIOP, and has its advantages and disadvantages.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49



The J2EE Technologies

- **Java servlets.** Servlets are networked components that you can use to extend the functionality of a Web server. Servlets are request/response oriented in that they take requests from some client host (such as a Web browser) and issue a response back to that host. This makes servlets ideal for performing Web tasks, such as rendering an HTML interface. Servlets differ from EJB components in that the breadth of server-side component features that EJB offers, such as declarative transactions, is not readily available to servlets. Servlets are much better suited to handling simple request/response needs, and they do not require sophisticated management by an application server.
- **Java IDL.** Java IDL is the Sun Microsystems Java-based implementation of CORBA. Java IDL allows for integration with other languages. Java IDL also allows for distributed objects to leverage the full range of CORBA services. J2EE is thus fully compatible with CORBA, completing the Java 2 Platform, Enterprise Edition.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50



The J2EE Technologies

- **JavaServer Pages (JSP).** JSP technology is very similar to servlets. Infact, JSP scripts are compiled into servlets. The largest difference between JSP scripts and servlets is that JSP scripts are not pure Java code; they are much more centered on look-and-feel issues. You would use JSP when you want the look and feel of your deployment to be physically separate and easily maintainable from the rest of your deployment. JSP technology is perfect for this, and it can be easily written and maintained by non-Java-savvy staff members (JSP technology does not require a Java compiler).
- **JavaMail.** The JavaMail service enables you to send e-mail messages in a platform-independent, protocol-independent manner from your Java programs. For example, in a server-side J2EE deployment, you can use JavaMail to confirm a purchase made on your Internet e-commerce site by sending an e-mail to the customer. Note that JavaMail depends on the *JavaBeans Activation Framework (JAF)*, which makes JAF part of J2EE as well.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



The J2EE Technologies

- **The Java API for XML Parsing (JAXP).** There are many applications of XML in a J2EE deployment. For example, you might need to parse XML if you are performing B2B interactions (such as through Web Services), if you are accessing legacy systems and mapping data to and from XML, or if you are persisting XML documents to a database. JAXP is the de facto API for parsing XML documents in a J2EE application and is an implementation-neutral interface to XML parsing technologies such as DOM and SAX. You typically use the JAXP API from within servlets, JSP, or EJB components.
- **The Java Authentication and Authorization Service (JAAS).** JAAS is a standard API for performing security-related operations in J2EE. Conceptually, JAAS also enables you to plug in an authentication mechanism into a J2EE application server.

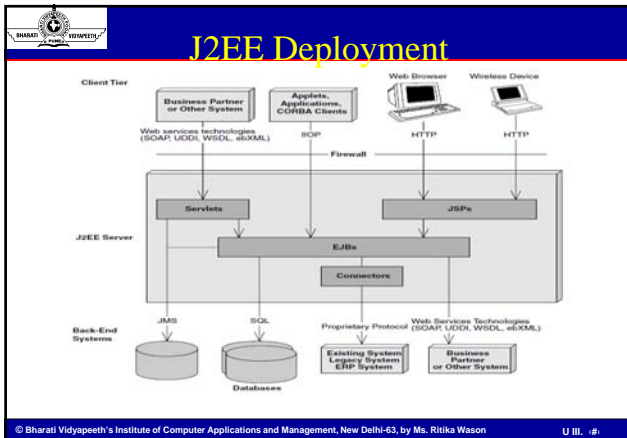
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49



The J2EE Technologies

- **J2EE Connector Architecture (JCA).** Connectors enable you to access existing enterprise information systems from a J2EE application. This could include *any existing system, such as a mainframe system running high-end transactions* (such as those deployed with IBM CICS, or BEA TUXEDO), Enterprise Resource Planning (ERP) systems, or your own proprietary systems. Connectors are useful because they automatically manage the details of middleware integration to existing systems, such as handling transactions and security concerns, life-cycle management, thread management, and so on. Another value of this architecture is that you can write a connector to access an existing system once, and then deploy it into any J2EE-compliant server. This is important because you only need to learn how to access an existing system once. Furthermore, the connector needs to be developed only once and can be reused in any J2EE server. This is extremely useful for independent software vendors (ISVs) such as SAP, Siebel, Peoplesoft and others who want their software to be accessible from within J2EE application servers. Rather than write a custom connector for each application server, these ISVs can write a standard J2EE connector.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50



Enterprise Bean

- An *enterprise bean* is a *server-side software component that can be deployed in a distributed multitier environment*. An enterprise bean can compose one or more Java objects because a component may be more than just a simple object.
- Regardless of an enterprise bean's composition, the clients of the bean deal with a single exposed component interface. This interface, as well as the enterprise bean itself, must conform to the EJB specification. The specification requires that your beans expose a few required methods; these required methods allow the EJB container to manage beans uniformly, regardless of which container your bean is running in.
- Note that the client of an enterprise bean could be anything—a servlet, an applet, or even another enterprise bean. In the case of an enterprise bean, a client request to a bean can result in a whole chain of beans being called. This is a very powerful idea because you can subdivide a complex bean task, allowing one bean to call on a variety of prewritten beans to handle the subtasks. This hierarchical concept is quite extensible.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49

Types of Beans

- EJB 2.1 defines three different kinds of enterprise beans:
 - Session beans.** Session beans model business processes. They are like *verbs* because they perform actions. The action could be anything, such as adding numbers, accessing a database, calling a legacy system, or calling other enterprise beans. Examples include a pricing engine, a workflow engine, a catalog engine, a credit card authorizer, or a stocktrading engine.
 - Entity beans.** Entity beans model business data. They are like *nouns* because they are data objects—that is, Java objects that cache database information. Examples include a product, an order, an employee, a credit card, or a stock. Session beans typically harness entity beans to achieve business goals, such as a stock-trading engine (session bean) that deals with stocks (entity beans).

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50

Types of Beans

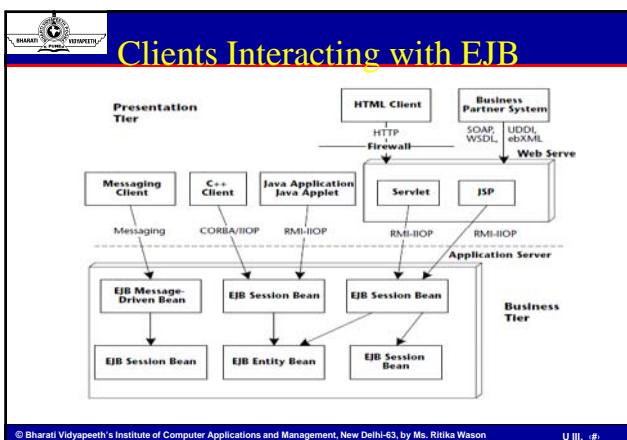
- **Message-driven beans.** Message-driven beans are similar to session beans in that they perform actions. The difference is that you can call message-driven beans only implicitly by sending *messages to those* beans. Examples of message-driven beans include beans that receive stock trade messages, credit card authorization messages, or workflow messages. These message-driven beans might call other enterprise beans as well.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48

Session Bean calling Entity Bean

ENTITY BEAN	SESSION BEAN
Bank teller	Bank account
Credit card authorizer	Credit card
DNA sequencer	DNA strand
Order entry system	Order, Line item
Catalog engine	Product
Auction broker	Bid, Item
Purchase order Approval router	Purchase order

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49



Distributed Objects: The Foundation for EJB

- EJB components are based on *distributed objects*.
- A *distributed object* is an object that is callable from a remote system. It can be called from an in-process client, an out-of-process client, or a client located elsewhere on the network.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 13

Distribution Transparency

- Distribution transparency is the holy grail in distributed systems technology and very hard to achieve. Perfect distribution transparency would mean that a client never sees any differences between local and remote interactions.
- In the presence of the more complex failure modes of remote operations and network latency, this is not possible. Most of the time, the term distribution transparency is used rather loosely to mean that the syntax of the code making invocations is the same for both remote and local invocations.
- Even this is not always the case when you consider the different exceptions found in remote interfaces that in turn require different exception handling, and the subtle differences between the pass-by-reference and pass-by-value semantics that local and remote invocations sometimes exhibit.
- For these reasons, most middleware systems settle for a less ambitious form of transparency, viz. location transparency. We will come back to location transparency in a moment.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 14

Distributed Objects and Middleware

- Distributed objects are great because they enable you to break up an application across a network. However, as a distributed object application gets larger, you'll need help from middleware services, such as transactions and security.
- There are two ways to get middleware: explicitly and implicitly.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 15

Explicit Middleware

- In traditional distributed object programming (such as traditional CORBA), you can harness middleware by purchasing that middleware off the shelf and writing code that calls that middleware's API.
- For example, you could gain transactions by writing to a transaction API. We call this *explicit middleware* because you need to write to an API to use that middleware.
- The following example shows a bank account distributed object that knows how to transfer funds from one account to another. It is filled with pseudocode that illustrates explicit middleware.

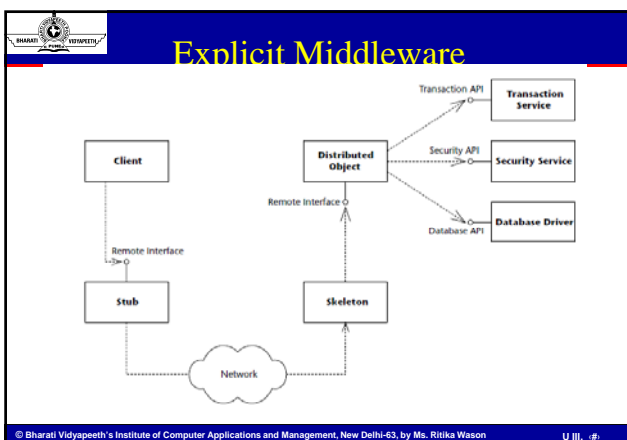
```
Ex: transfer(Account account1, Account account2, long amount) {
// 1: Call middleware API to perform a security check
// 2: Call middleware API to start a transaction
// 3: Call middleware API to load rows from the database
// 4: Subtract the balance from one account, add to the other
// 5: Call middleware API to store rows in the database
// 6: Call middleware API to end the transaction
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48

Explicit Middleware

- As you can see, we are gaining middleware, but our business logic is intertwined with the logic to call these middleware APIs, which is not without its downsides. This approach is:
 - Difficult to write. The code is bloated. We simply want to perform a transfer, but it requires a large amount of code.
 - Difficult to maintain. If you want to change the way you do middleware, you need to rewrite your code.
 - Difficult to support. If you are an Independent Software Vendor (ISV) selling an application, or an internal department providing code to another department, you are unlikely to provide source code to your customers. This is because the source code is your intellectual property, and also because upgrading your customers to the next version of your software is difficult if those customers modify source code. Thus, your customers cannot change their middleware (such as changing how security works).

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49



Implicit Middleware

- The crucial difference between systems of the past (transaction processing monitors such as TUXEDO or CICS, or traditional distributed object technologies such as CORBA, DCOM, or RMI) and the newer, component-based technologies (EJB, CORBA Component Model, and Microsoft.NET) is that in this new world, you can harness complex middleware in your enterprise applications without writing to middleware APIs.
- In outline, follow these steps to harness the power of middleware:
 - Write your distributed object to contain *only business logic*. *Do not write* to complex middleware APIs. For example, this is the code that would run inside the distributed object:


```
transfer(Account account1, Account account2, long amount) {
// 1: Subtract the balance from one account, add to the other
}
```

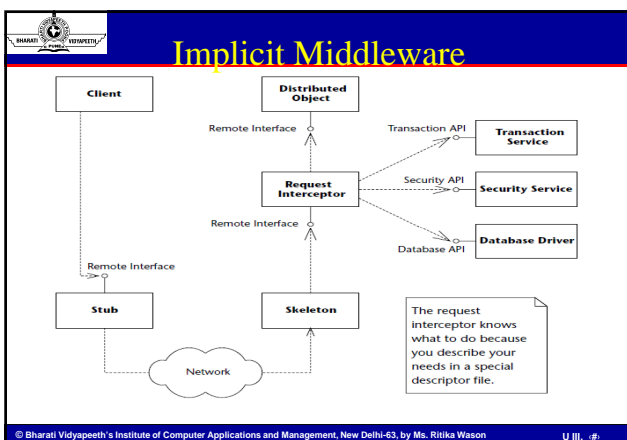
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 15

Implicit Middleware

- Declare the middleware services that your distributed object needs in a separate descriptor file, such as a plain text file. For example, you might declare that you need transactions, persistence, and a security check.
- Run a command-line tool provided for you by the middleware vendor. This tool takes your descriptor file as input and generates an object that we'll call the *request interceptor*.
- The request interceptor intercepts requests from the client, performs the middleware that your distributed object needs (such as transactions, security, and persistence), and then delegates the call to the distributed object.

- The values of *implicit middleware* (also called *declarative middleware*) are:
 - Easy to write.
 - Easy to maintain.
 - Easy to support.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 15



Developing EJB Components

- When building an EJB component, the following is a typical order of operations:
 1. Write the .java files that compose your bean: the component interfaces, home interfaces, enterprise bean class file, and any helper classes you might need.
 2. Write the deployment descriptor, or have it generated by your IDE or tools like XDoclet.
 3. Compile the .java files from Step 1 into .class files.
 4. Using the *jar* utility, create an Ejb-jar file containing the deployment descriptor and .class files.
 5. Deploy the Ejb-jar file into your container in a vendor-specific manner, perhaps by running a vendor-specific tool or perhaps by copying your Ejb-jar file into a folder where your container looks to load Ejb-jar files.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason

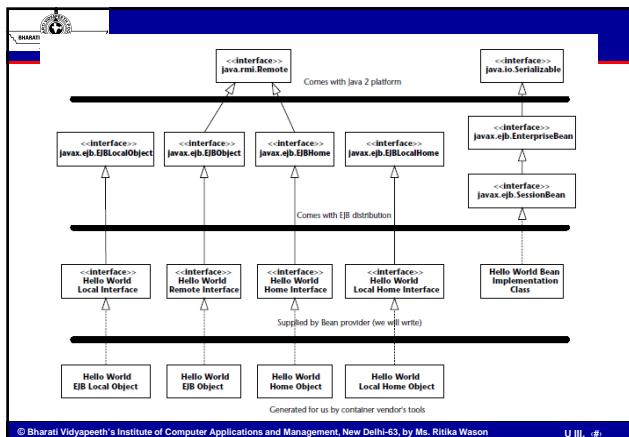
U III. (#)

Developing EJB Components

6. Configure your EJB server so that it is properly configured to host your Ejb-jar file. You might tune things such as database connections, thread pools, and so on. This step is vendor-specific and might be done through a Web-based console or by editing a configuration file.
7. Start your EJB container and confirm that it has loaded your Ejb-jar file.
8. Optionally, write a standalone test client .java file and let vendor tools generate stub classes for remote access, if required. Compile that test client into a .class file. Run the test client from the command line and have it exercise your bean's APIs.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason

U III. (#)



© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason

U III. (#)



The Remote Interface (Hello.java)

- package examples;
- /** This is the HelloBean remote interface.


* This interface is what clients operate on when they interact with EJB *objects. The container vendor will implement this interface; the

* implemented object is the EJB object, which delegates invocations to *the actual bean.

*/

- public interface Hello extends javax.ejb.EJBObject
- {
- /** The one method - hello - returns a greeting to the client.
- */
- public String hello() throws java.rmi.RemoteException;
- }

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 17



The Local Interface (HelloLocal.java)

- /** This is the HelloBean local interface. This interface is what local *clients operate on when they interact with EJB local objects.

* The container vendor will implement this interface; the implemented *object is the EJB local object, which delegates invocations

* to the actual bean. */

public interface HelloLocal extends javax.ejb.EJBLocalObject

{


/** * The one method - hello - returns a greeting to the client.

*/

public String hello();

}

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 18



Home Interface (HelloHome.java)

- /** This is the home interface for HelloBean. This interface is *implemented by the EJB Server's tools - the implemented object is *called the Home Object, and serves as a factory for EJB Objects.

* One create() method is in this Home Interface, which corresponds to the *ejbCreate() method in HelloBean. */

public interface HelloHome extends javax.ejb.EJBHome

{

/*

*This method creates the EJB Object.

* @return The newly created EJB Object.

*/

Hello create() throws java.rmi.RemoteException, javax.ejb.CreateException;

}

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 19

The Local Home Interface (HelloLocalHome.java)

- package examples;
- /* This is the local home interface for HelloBean. This interface is implemented by the EJB Server's tools - the implemented object is called the local home object, and serves as a factory for EJB local objects.*/

```
public interface HelloLocalHome extends javax.ejb.EJBLocalHome
{
    /*
    * This method creates the EJB Object.
    * @return The newly created EJB Object.
    */
    HelloLocal create() throws javax.ejb.CreateException;
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48

The Remote Interface


- The remote interface supports every business method that our beans expose. Things to notice about our remote interface include the following:
- We extend *javax.ejb.EJBObject*. This means that the container-generated EJB object, which implements the remote interface, will contain every method that the *javax.ejb.EJBObject* interface defines. This includes a method to compare two EJB objects, a method to remove an EJB object, and so on.
- We have one business method—*hello()*—which returns the String “Hello, World!” to the client. We need to implement this method in our enterprise bean class. Because the remote interface is an RMI-IIOP remote interface (it extends *java.rmi.Remote*), it must throw a remote exception. This is the only difference between the remote interface's *hello()* signature and our bean's *hello()* signature. The exception indicates a networking or other critical problem.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49

//Hello.java

- package examples;
- /*
- * This is the HelloBean remote interface.
- *
- * This interface is what clients operate on when they interact with EJB objects. The container vendor will implement this interface; the implemented object is the EJB object, which delegates invocations to the actual bean.
- */
- public interface Hello extends javax.ejb.EJBObject{
- /*
- * The one method - hello - returns a greeting to the client.
- */
- public String hello() throws java.rmi.RemoteException;
- }


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50



The Local Interface

- Local clients will use local interface, rather than the remote interface, to call our beans' methods. (//HelloLocal.java)
- package examples;
- /**
- * This is the HelloBean local interface. This interface is what local clients *operate on when they interact with EJB local objects. The container vendor *will implement this interface; the implemented object is the EJB local *object, which delegates invocations to the actual bean.
- */
- public interface HelloLocal extends javax.ejb.EJBLocalObject
- {
- /**
- * The one method - hello - returns a greeting to the client.
- */
- public String hello();


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



The Home Interface


- The home interface has methods to create and destroy EJB objects. The implementation of the home interface is the home object, which is generated by the container tools.
- /**
- * This is the home interface for HelloBean. This interface is implemented *by the EJB Server's tools – the implemented object is called the Home *Object, and serves as a factory for EJB Objects. One create() method is in *this Home Interface, which corresponds to the ejbCreate() method in *HelloBean.
- */
- public interface HelloHome extends javax.ejb.EJBHome{
- /**
- * This method creates the EJB Object. @return The newly created EJB Object.
- */
- Hello create() throws java.rmi.RemoteException,
- javax.ejb.CreateException; }

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49



- Notice the following about our home interface:
- The single *create()* is a *factory method* that clients use to get a reference to an EJB object. The *create()* method is also used to initialize a bean.
- The *create()* method throws two exceptions: *java.rmi.RemoteException* and *javax.ejb.CreateException*. Remote exceptions are necessary side effects of RMI-IIOP because the home object is a networked RMI-IIOP remote object. *CreateException* is also required in all *create()* methods. We explain this further in the Exceptions and EJB sidebar.
- Our home interface extends *javax.ejb.EJBHome*. This is required for all home interfaces. *EJBHome* defines a way to destroy an EJB object, so we don't need to write that method signature.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50



The Local Home Interface

- Is the higher-performing home interface used by local clients.
- package examples;
- `/*`
- `* This is the local home interface for HelloBean. This interface is implemented by the EJB Server's tools - the implemented object is called the local home object, and serves as a factory for EJB local objects.`
- `*/`
- `public interface HelloLocalHome extends javax.ejb.EJBLocalHome`
- `{`
- `/*`
- `* This method creates the EJB Object.`
- `* @return The newly created EJB Object.`
- `*/`
- `HelloLocal create() throws javax.ejb.CreateException;`
- `}`


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



Remote versus Local Interface

- The local home interface extends *EJBLocalHome* rather than *EJBHome*. The *EJBLocalHome* interface does not extend *java.rmi.Remote*. This means that the generated implementation will not be a remote object.
- The local home interface does not throw *RemoteExceptions*.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49



The Bean Class

```
package examples;
/** Demonstration stateless session bean.*/
public class HelloBean implements javax.ejb.SessionBean {
    private SessionContext ctx;
    // EJB-required methods
    public void ejbCreate() {System.out.println("ejbCreate()");}
    public void ejbRemove() {System.out.println("ejbRemove()");}
    public void ejbActivate() {System.out.println("ejbActivate()");}
    public void ejbPassivate() {System.out.println("ejbPassivate()");}
    public void setSessionContext(javax.ejb.SessionContext ctx) {
        this.ctx = ctx;}
    // Business methods
    public String hello() {System.out.println("hello()");
        return "Hello, World!";}
}
```


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50



The Bean Class

- This is just about the most basic bean class possible. Notice:
- Our bean implements the *javax.ejb.SessionBean* interface, which makes it a session bean. This interface defines a few required methods that you must fill in.
- The container uses these management methods to interact with the bean, calling them periodically to alert the bean to important events. For example, the container will alert the bean when it is being initialized and when it is being destroyed. These callbacks are not intended for client use, so you will never call them directly—only your EJB container will. The bean has an *ejbCreate()* method that matches the home object's *create()* method, and takes no parameters.
- We have one business method, *hello()*. It returns *Hello, World!* to the client.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



The Bean Class

- The *ejbActivate()* and *ejbPassivate()* methods do not apply to stateless session beans, and so we leave these methods empty.
- When we destroy the bean, there's nothing to clean up, so we have a very simple *ejbRemove()* method.


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49



The Deployment Descriptor

- Next, we need to generate a *deployment descriptor*, which describes our bean's middleware requirements to the container. Deployment descriptors are one of the key features of EJB because they enable you to *declaratively specify attributes* on your beans, rather than program this functionality into the bean itself.
- Physically, a deployment descriptor is an XML document. Your EJB container, IDE environment, or other tool (such as a UML editor that can generate EJB code) should supply tools to help you generate such a deployment descriptor.
- Many different settings make up a deployment descriptor.
- Here is an explanation of our session bean descriptor:


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 50



The Deployment Descriptor

- **<ejb-name>** The nickname for this particular bean. Can be used later in the deployment descriptor to refer back to this bean to set additional settings.
- **<home>** The fully qualified name of the home interface.
- **<remote>** The fully qualified name of the remote interface.
- **<local-home>** The fully qualified name of the local home interface.
- **<local>** The fully qualified name of the local interface.
- **<ejb-class>** The fully qualified name of the enterprise bean class.
- **<session-type>** Whether the session bean is a stateful or stateless session bean.
- **<transaction-type>**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 48



- `<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise Æ JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">`
- `<ejb-jar> <enterprise-beans>`
- `<session>`
- `<ejb-name>Hello</ejb-name>`
- `<home>examples.HelloHome</home>`
- `<remote>examples.Hello</remote>`
- `<local-home>examples.HelloLocalHome</local-home>`
- `<local>examples.HelloLocal</local>`
- `<ejb-class>examples.HelloBean</ejb-class>`
- `<session-type>Stateless</session-type>`
- `<transaction-type>Container</transaction-type>`
- `</session> </enterprise-beans> </ejb-jar>`

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U III. 49
