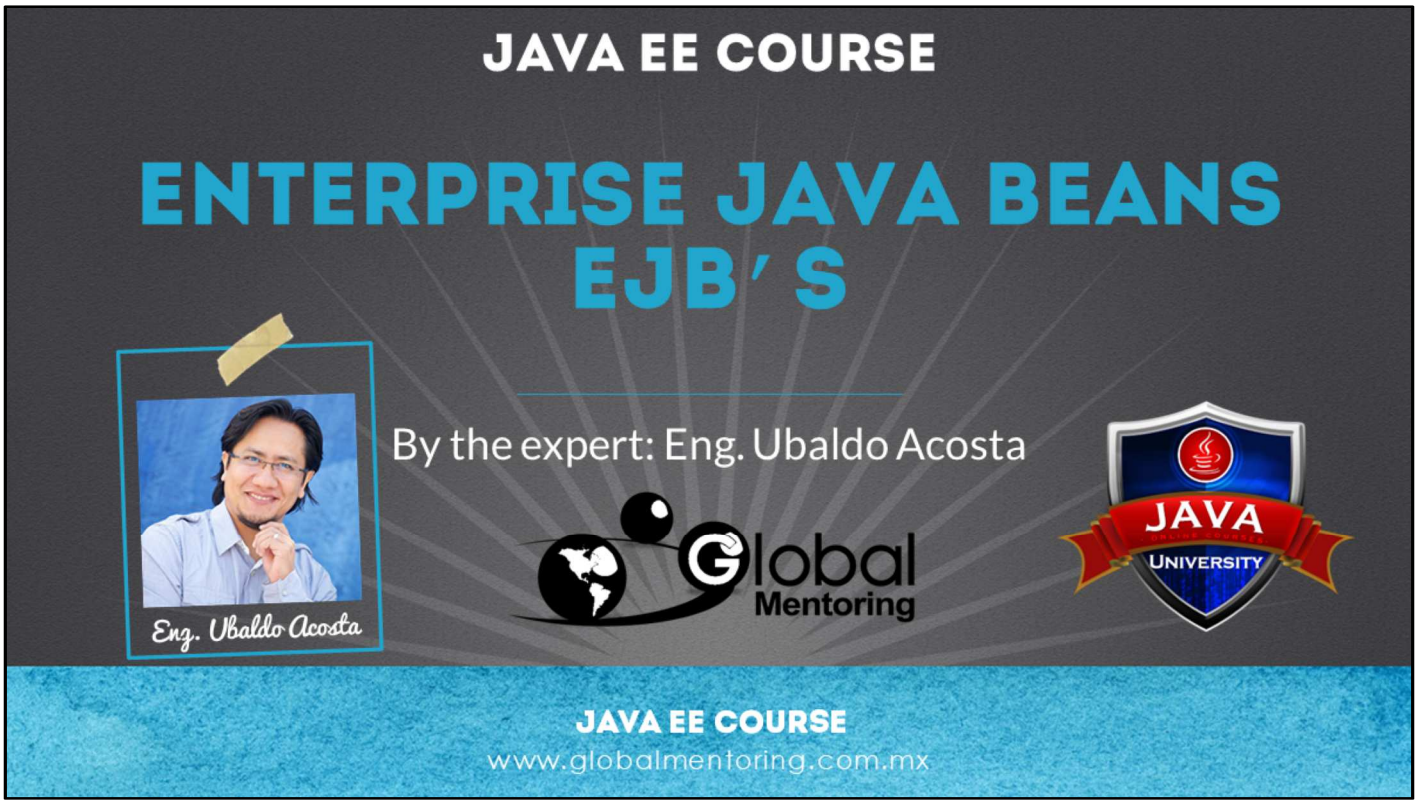
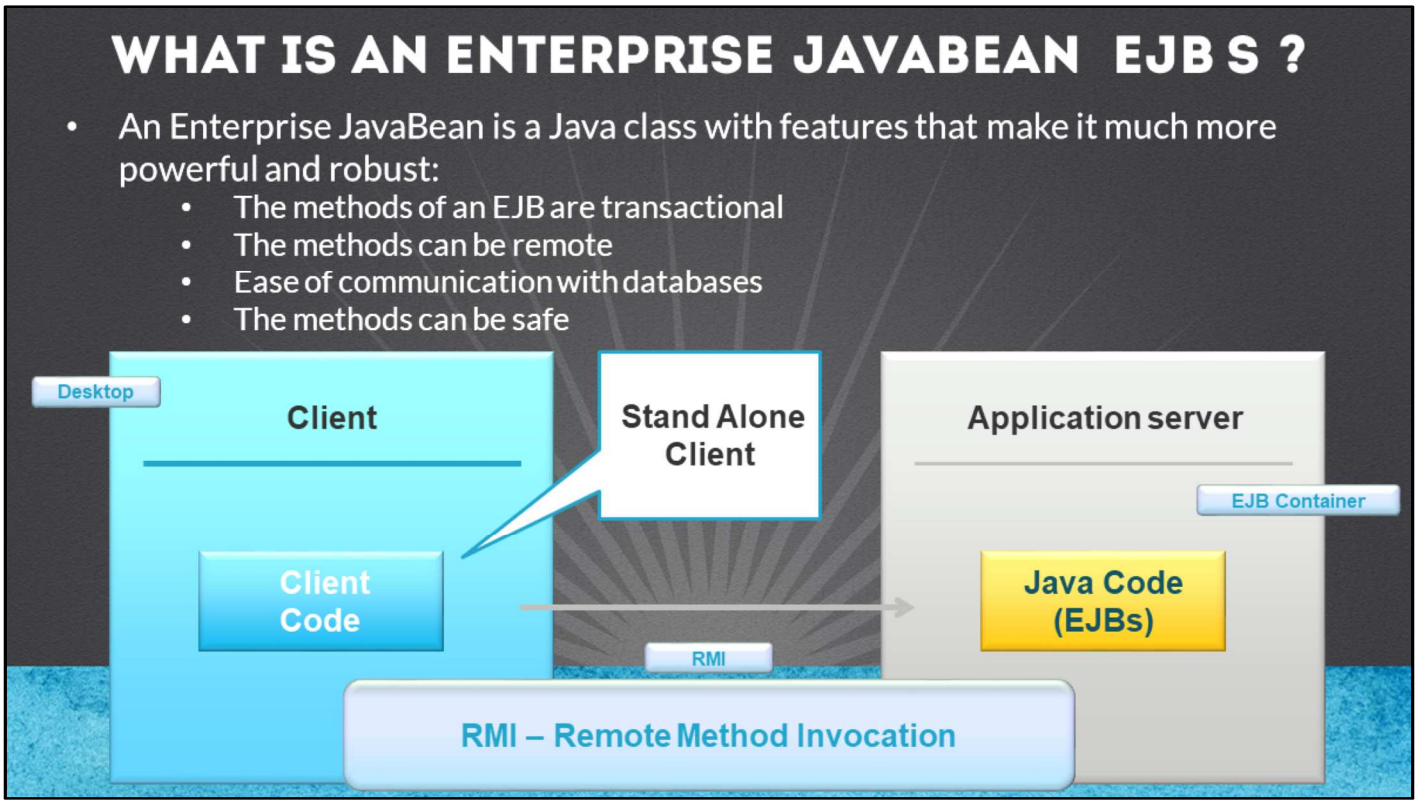


Hello, Ubaldo Acosta greets you again. I hope you're ready to start with this lesson.

We are going to study the topic of Enterprise Java Beans (EJB's).

Are you ready? Let's go!



The Enterprise Java Beans (EJB) is Java code on the server side. Normally they have the business logic of our application, and therefore they cover the role of the service layer of our Java applications, as we reviewed previously.

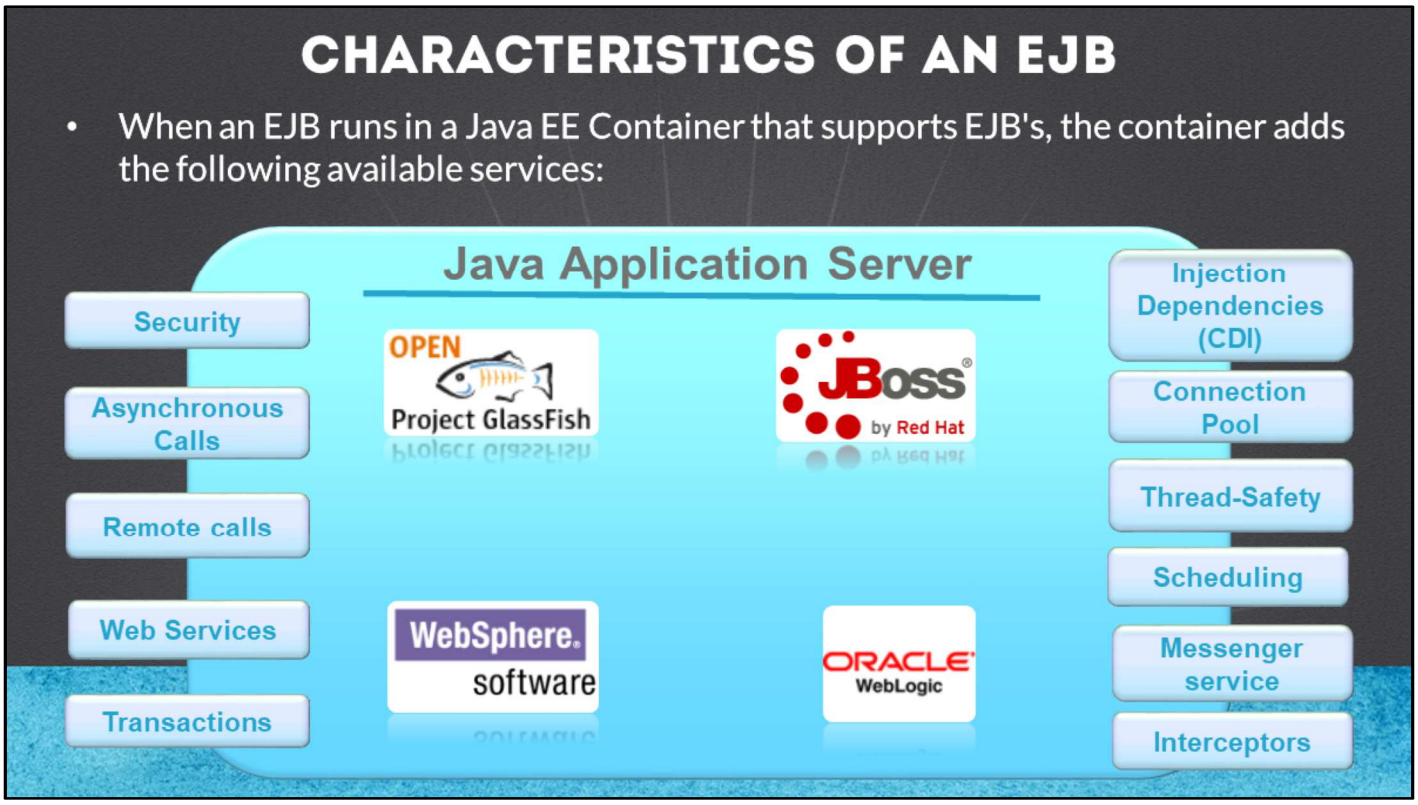
Today, EJBs are pure Java classes (POJO's) which, when deployed in an Application Server, reduce the complexity of programming, adding robustness, reusability and scalability to our mission-critical business applications.

Today more than ever EJBs can be programmed once and run on any Java application server that supports the Java EE standard. Today's EJBs are a proven technology that offers benefits such as security, transactionality, multi-threading, among many other features, all with the help of a Java application server.

Unlike a JavaBean, which is a pure Java class, an Enterprise JavaBean is a Java class with features that make it much more powerful and robust:

- The methods of an EJB are transactional.
- The methods can be remote.
- Ease of communication with databases.
- The methods can be safe.
- The methods can be asynchronous.
- Among many more features.

In the figure we can see that the Java code on the server side is an EJB, which can be executed by an application, known as Client. This client makes a request to the EJB component, which can be a local call (if it is on the same server) or a remote call (if it is outside the application server). If the call is remote, the RMI protocol (Remote Method Invocation) is used.



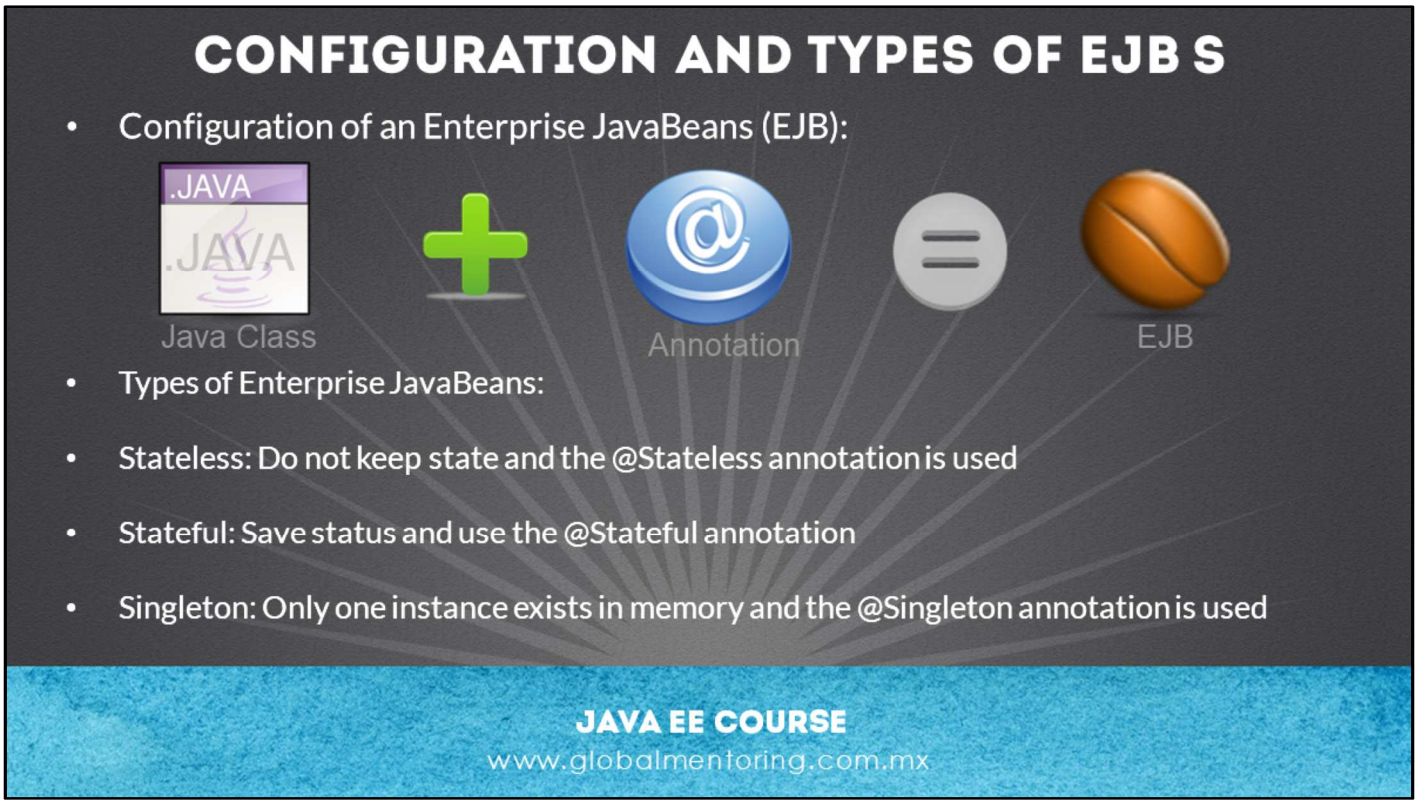
In a typical Java EE architecture, EJBs play the role of the Service layer, where it is common to find many of the business rules of our application.

A business rule is the rules or policies of the company or organization, for example, if a customer has been loyal to a product for a certain number of years, you can apply an extra discount for a certain amount of purchase. These types of decisions can be applied automatically through the system, and the business layer is one of those responsible for executing these rules.

The EJB's when running inside an EJB container and in turn within a Java application server, have several services available to them, such as:

- Security
- Asynchronous Calls
- Remote calls through RMI
- Transaction Management through JTA
- Exposure of business rules through Web Services (JAX-WS or JAX-RS)
- Dependency Injection Service through CDI
- Connection Pool Service
- Safe Concurrence Management (Tread-Safety)
- Management of Scheduled Tasks (Scheduling)
- Messaging Management through JMS
- Interceptors, allow intercepting calls to methods and add extra or complementary functionality through AOP (Aspect Oriented Programming)

Java application servers also add other features such as: clustering, load balancing and fault tolerance. This allows to create mission critical applications with operations 7 days a week, 24 hours a day, 365 days a year. So regardless of the type of application server that we use, we will have all these features available when creating and deploying our EJBs.



In versions prior to EJB 3.0, the programmer had to create several classes and interfaces to run an EJB: A local or remote interface (or both), a local or remote home interface (or both), and an xml configuration file , known as deployment descriptor.

The EJB in its version 3 promoted the use of annotations for its configuration, and continues adding and simplifying the integration of business technologies through the concept of annotations. This concept greatly facilitated the development of EJBs, and in general of all Java Enterprise technology.

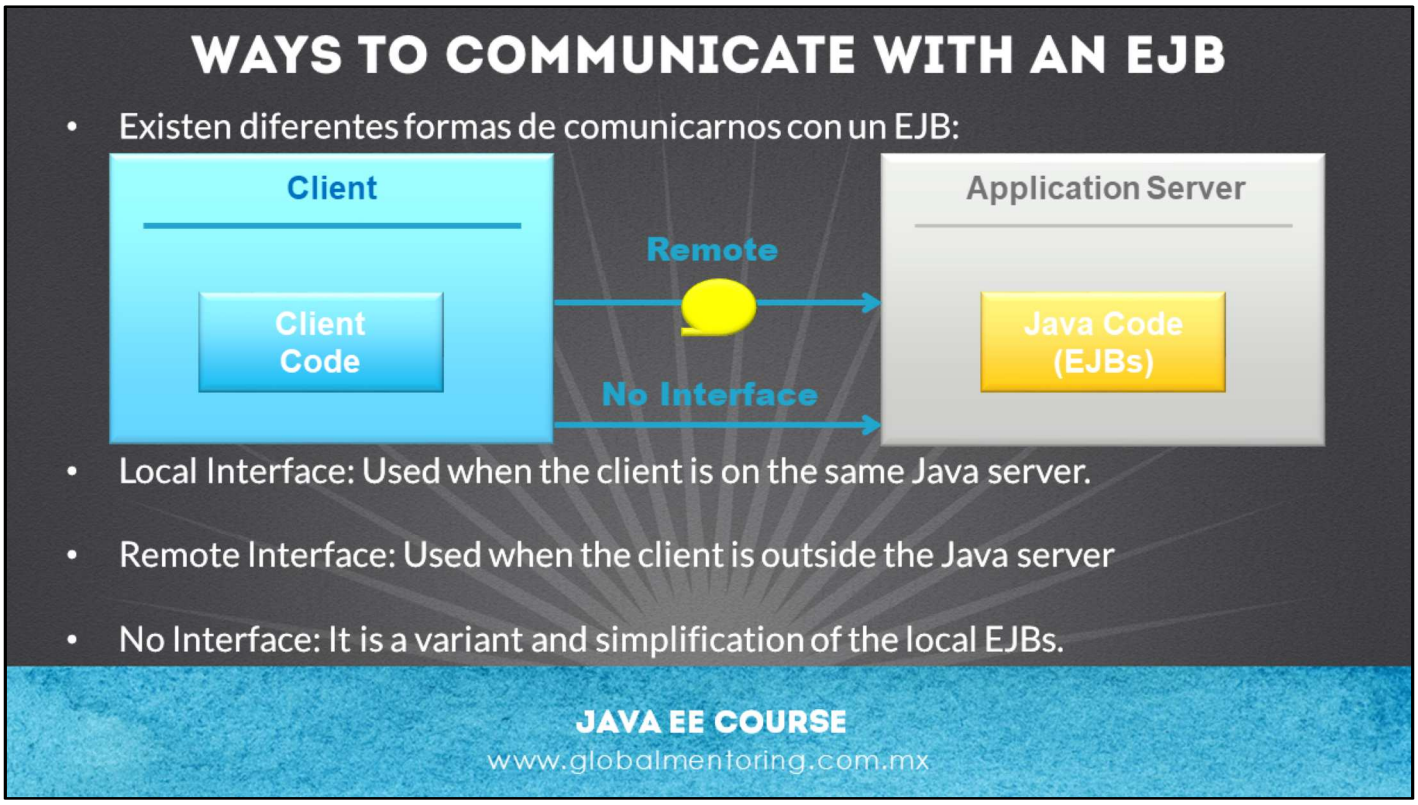
There are different types of beans, depending on the function that is added to a Java multilayer architecture. In addition, this organization allows you to better understand the configuration of a business application. Because business applications are often complex, the following types of EJBs have been defined, according to the requirements to be covered.

Session EJB: A session bean is invoked by the client to execute a specific business operation.

- **Stateless**: This type of EJB does not maintain any state of the user, that is, it does not remember any type of information after a transaction is completed.
- **Stateful**: This type of EJB maintains a status of the client's activity, for example, if you have a shopping cart. This status can be remembered even after the transaction is completed, but if the server is restarted this information is lost. The similar to the Session scope of a Web application.
- **Singleton**: This type of beans uses the Singleton design pattern, in which there is only one instance in memory of this class.

Other classifications that we can find are:

- **EJB Timer**: This is a feature that can be added to beans, so that they run at a specified time (scheduling).
- **Message-driven beans (MDBs)**: This type of beans is used to send messages using JMS technology. The study of this type of beans is beyond the scope of this course.
- **Entity Beans**: This is a classification prior to version 3 of the EJB, however today the standard JPA (Java Persistance API) has replaced this type of beans. So, unless we are using an earlier version than version 3, we should use JPA instead of Entity Beans.



The EJB 2.x version included more concepts and more programming complexity. In version 3 these concepts have been greatly simplified.

The EJBs can be configured in the following way, in order to allow communication with their methods:

- **Business Interfaces**: These interfaces contain the declaration of the business methods that are visible to the client. These interfaces are implemented by a Java class.
- **A Java class (bean)**: This class implements the business methods and can implement zero or more Business Interfaces. Depending on the type of EJB, this class must be annotated with @Stateless, @Stateful or @Singleton depending on the type of EJB that we want to create.

As we can see in the figure, we have different ways of communicating with our EJB component.

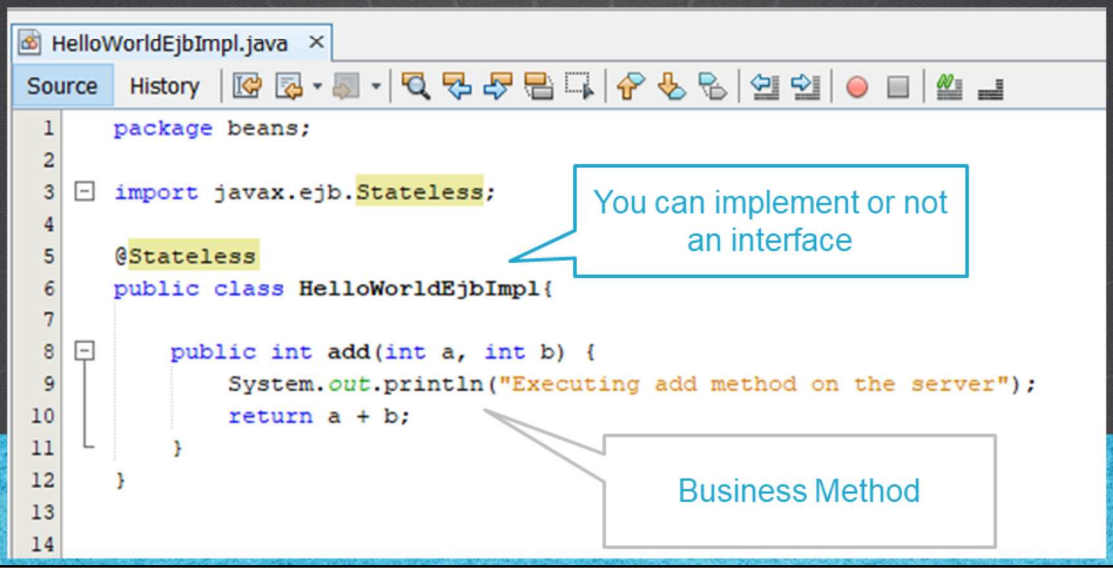
- **Local Interface**: Used when the client is inside the same Java server, thus avoiding the processing overload when using remote calls via RMI.

- **Remote Interface**: Used when the client code is outside the Java application server (in a different Java Virtual Machine) and therefore we must make remote calls in order to execute the EJB methods.

- **No Interface**: It is a simplification in version 3, since it does not require an interface to establish communication, as long as the calls are local, that is, within the same Java application server.

ANATOMY OF AN EJB

- In the following figure we can see the general structure of an EJB, which may or may not implement an interface (local or remote), and may have one or more business methods:



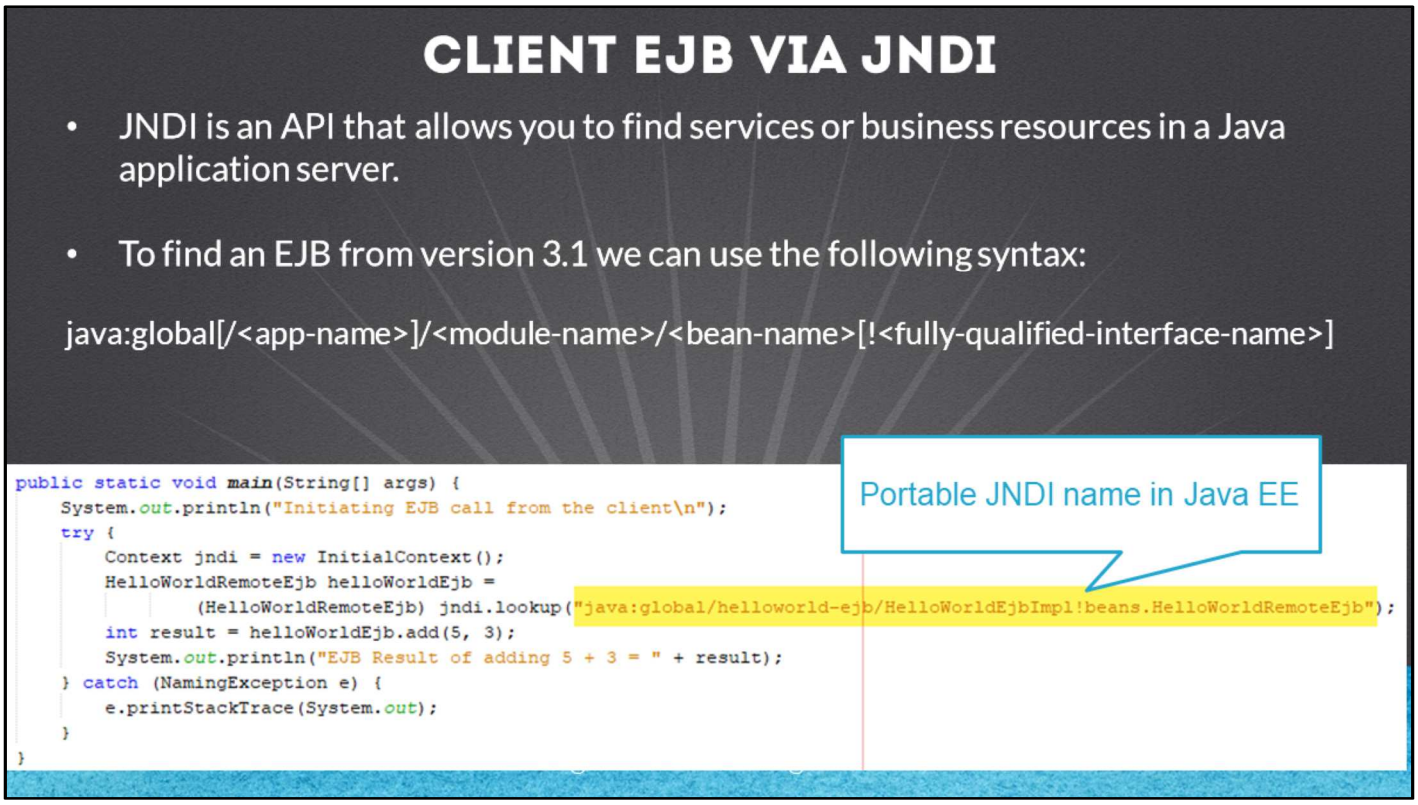
In the J2EE version (predecessor of Java EE), it was required to create several classes to run an EJB: a local or remote interface (or both), a local or remote home interface (or both) and an xml deployment descriptor. From the Java EE 5 and EJB 3 version, this configuration was dramatically simplified by adding the concept of annotations, however it was still necessary to add an interface to the EJB, local or remote.

As seen in the figure, the new versions of Java EE and EJB allow converting a pure Java class (POJO: Plain Old Java Object) into an EJB by simply adding the corresponding bean notation, for example @Stateless. This automatically makes this class have features such as transactional methods, methods with security, and can access the entity manager (entity manager) and thus persist information in the database, among many other features. All this simply by adding the EJB annotation.

Another way to configure an EJB is by using the descriptor file ejb-jar.xml, which is already optional today. This descriptor file overwrites the behavior added with the annotations in the Java classes.

Although the code shown in the figure is very simple, we must emphasize and remember that an EJB is a component that runs in a Java container. This execution environment is what allows adding business features to our Java classes allowing remote calls, dependency injection, states and life cycle management, object pooling, messaging handling, transaction management, security, concurrency support , interceptors, handling of asynchronous methods, among several other features.

All this happens simply by deploying this Java EJB class to the application server. This allows the Java programmer to focus on the business methods and delegate all these non-functional requirements characteristics to the Java application servers, such as Glassfish, JBoss, WebLogic, WebSphere, etc.



JNDI (Java Naming and Directory Interface) is an API that allows us to find services or resources in a Java application server.

In a JNDI startup it was the only way to find the EJB components, but as the concept of local EJBs and annotations management was introduced there were other ways to locate and provide a reference of the business components that are needed, this concept is known as dependency injection.

Prior to the JEE 6 version, there was no standard name for locating the EJBs through the JNDI API, so each Java server provided different syntaxes for locating the business components. However, starting with the Java EE 6 version, a global name was introduced to locate the EJB components.

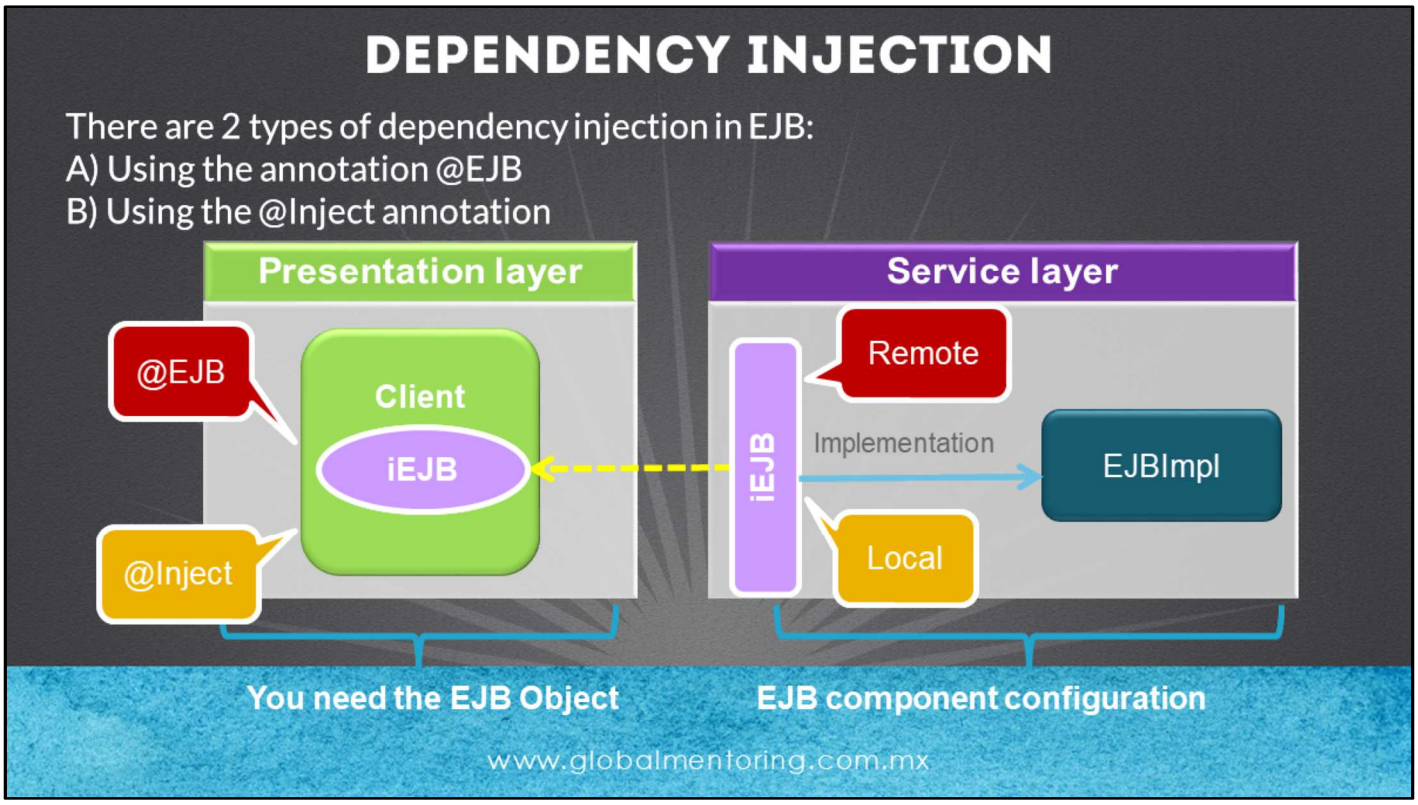
java: global [/<app-name>] /<module-name>/<bean-name>[!<fully-qualified-interface-name>]

This allows to locate any EJB in any Java application server. The basic code for finding an EJB using JNDI is:

HelloWorldEJB ejb = (HelloWorldEJB) context.lookup ("java: global / classes / HolaMundoEJB");

Or including the name of the Java package:

HelloWorldEJB ejb = (HelloWorldEJB) context.lookup ("java: global / classes / HelloWorldEJB! Beans.HelloWorldEJB");



In the figure shown we can see a layered example of a business architecture. In this example, the Client class in the presentation layer needs the EJB component of the service layer, which can be located on the same server (local call) or outside it (remote call). In order for the Client class to be able to use the EJB component, the application server can provide a reference for that component, this is known as Dependency Injection.

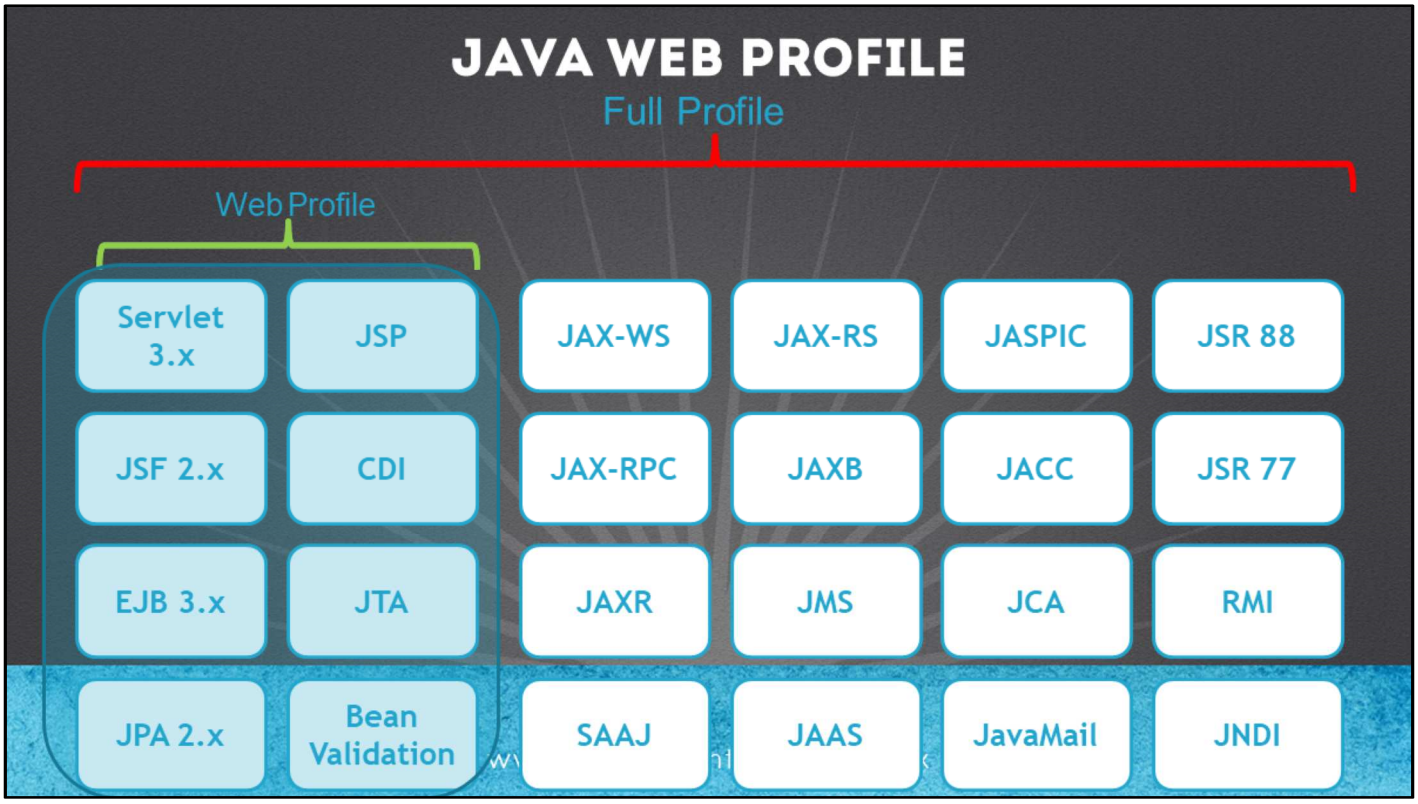
The injection of dependencies checks whether an EJB exists in memory with the same type or with the same name, as specified, and if that object exists, the Java application server returns a reference so that it can be used. In the Java EE enterprise version, there are two ways to perform dependency injection.

a) Using the @EJB annotation: This option is available from the Java EE 5 version, however it is the most basic form of dependency injection. The @EJB annotation is recommended when we use remote calls to the EJB, inject a resource (JDBC DataSource, JPA, Web Service, etc) or if we want to maintain compatibility with Java EE 5. Example of code in the client:

```
@EJB
  private PersonEJB personEJB;
```

b) Using the @Inject annotation: This form of dependency injection is based on the CDI concept (Context and Dependency Injection), and is available from the Java EE version 6. This form is more flexible and robust, since many of the concepts they were taken from the experience of other frameworks like Spring, which have more powerful and robust dependency injection methods. In order for the Java application server to recognize the concept of CDI, a file called beans.xml must be added. It is recommended to use the @Inject annotation on @EJB in all cases, except when we have remote EJBs or we want to maintain compatibility with Java EE 5. Example of code in the client.

```
@Inject
  private PersonEJB personEJB;
```



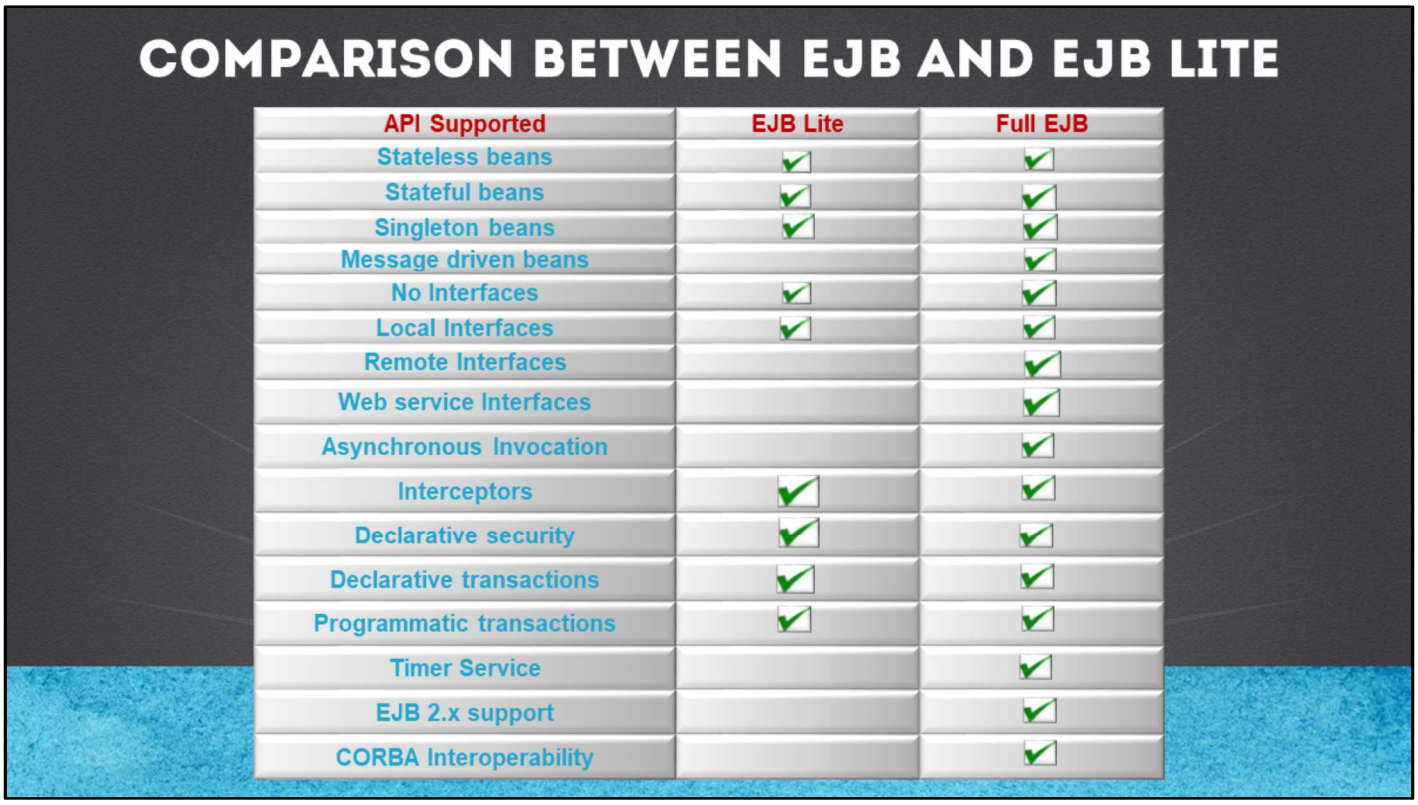
In the figure we can see the Java EE API and in particular the relationship with the Web profile, which has access only to certain APIs.

This arose because many of the Java EE applications did not need all the power nor the robust APIs they have, therefore only the most common APIs were added to this Web profile. The good news is that we can use EJBs 3.x in our Web applications without adding the configuration complexity of the EJBs in previous versions.

In fact, in the latest Java EE version it is possible to use local EJBs without the need to package them separately in an .jar file, but only use a .war file. The subject of packaging will be reviewed later. However, what we must highlight from this figure is to observe that we have access to the EJB, JPA, JTA, CDI, as the most common APIs that we will use in our business applications.

If we need other APIs such as Java Mail, Web Services, etc., it will be necessary to use a complete application server (full).

Selecting a type of profile will depend on the current and future requirements of our business application, so it is up to the Architect / Programmer to select the Java EE profile that best suits their needs.



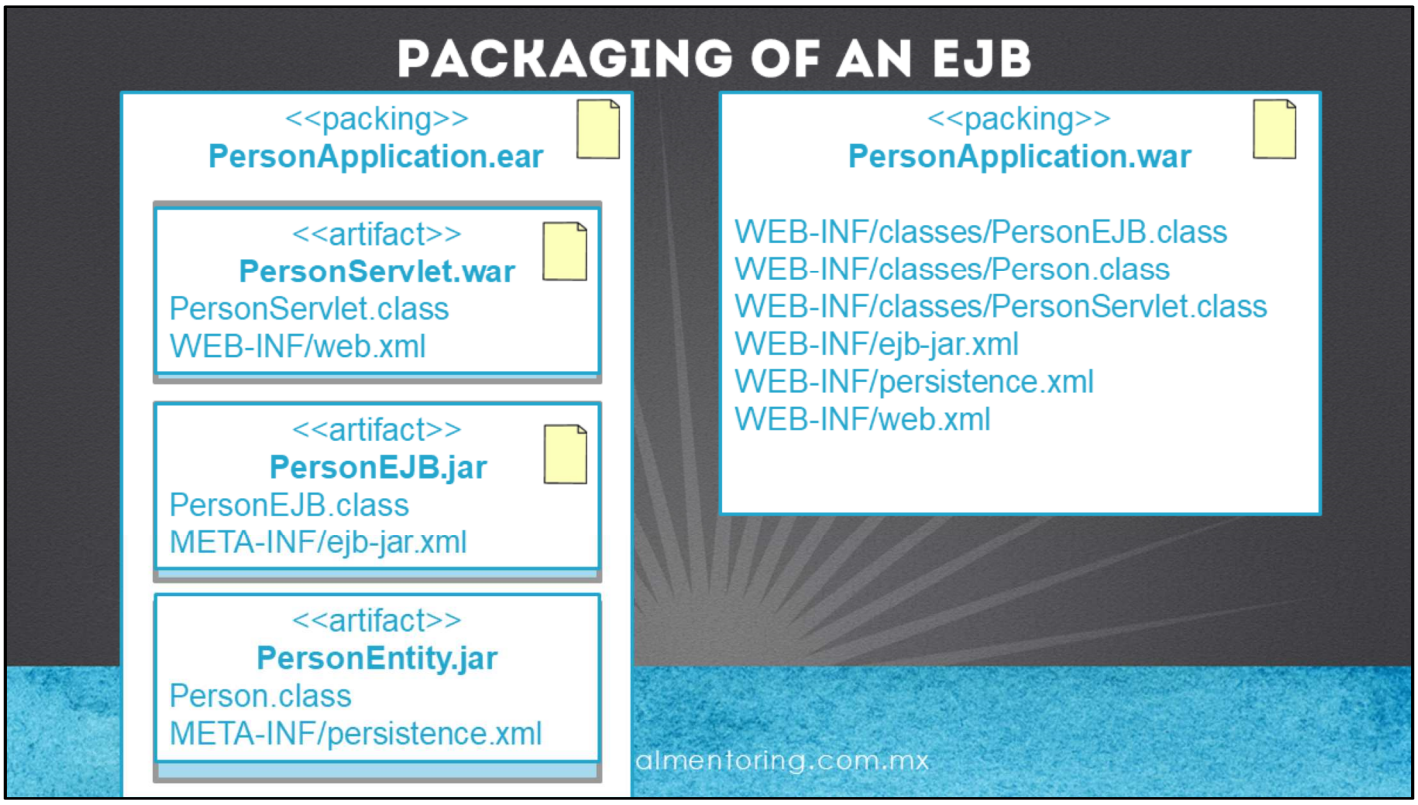
The predominant components in Java EE, without a doubt are the EJBs, which add very simple transactionality, security, among more features that we have already commented.

According to the previous sheet, we can use the Java EE Web profile and use EJBs. The minimum specification of APIs that we can use in a Web profile is known as EJB Lite. The limitations of APIs that we have in the Web profile are the limitations that we have when we use EJB, for that reason the name of lite.

We can see in the figure that if we use the Web profile and therefore EJB Lite, we will only have access to the listed API, excluding JMS, remote calls to EJB, exposure of EJB methods such as Web Service, Asynchronous calls, support for EJB 2.x , among other features that will NOT be available.

However we can see that many of the most common business requirements will have access. For example: Security, Transaction Management, declaration of EJB Local Stateless type, Stateful, Singleton, with or without Local Interface, etc.

This greatly simplified the Web applications that need this type of business requirements, without sacrificing the performance or performance of our Java application.



Because a Java Enterprise application includes different types of components, such as: Servlets, JSF pages, Web Services, EJB, etc., these components must be packaged to be deployed in the Java application server.

The EJB modules are placed in META-INF / ejb-jar.xml and in WEB-INF / ejb-jar.xml for the Web modules. EJB lite can be packaged directly in a .war (Web Archive File) or .jar (Java Archive File) file.

If your requirements use the complete specification of EJBs (remote calls, JMS, asynchronous calls, Web Services, etc), then it should be packaged in a .jar file and not in a .war file.

An .ear file (Enterprise Archive File) is used to package one or more modules, either .jar or .war., In a single file, which is recognized by the application server and it is responsible for correctly deploying each module packaged in the .ear file.

As we can see in the figure, if we need to deploy a Web application, we can package the EJBs and the Entity classes in separate .jar files, Servlets, and JSP or JSF pages within the .war file, and these files add them to a file .ear, which packages all the components into one.

From the EJB 3.x specification, the EJB Lite concept can package EJB components directly into a .war file, without the need for the .jar file.