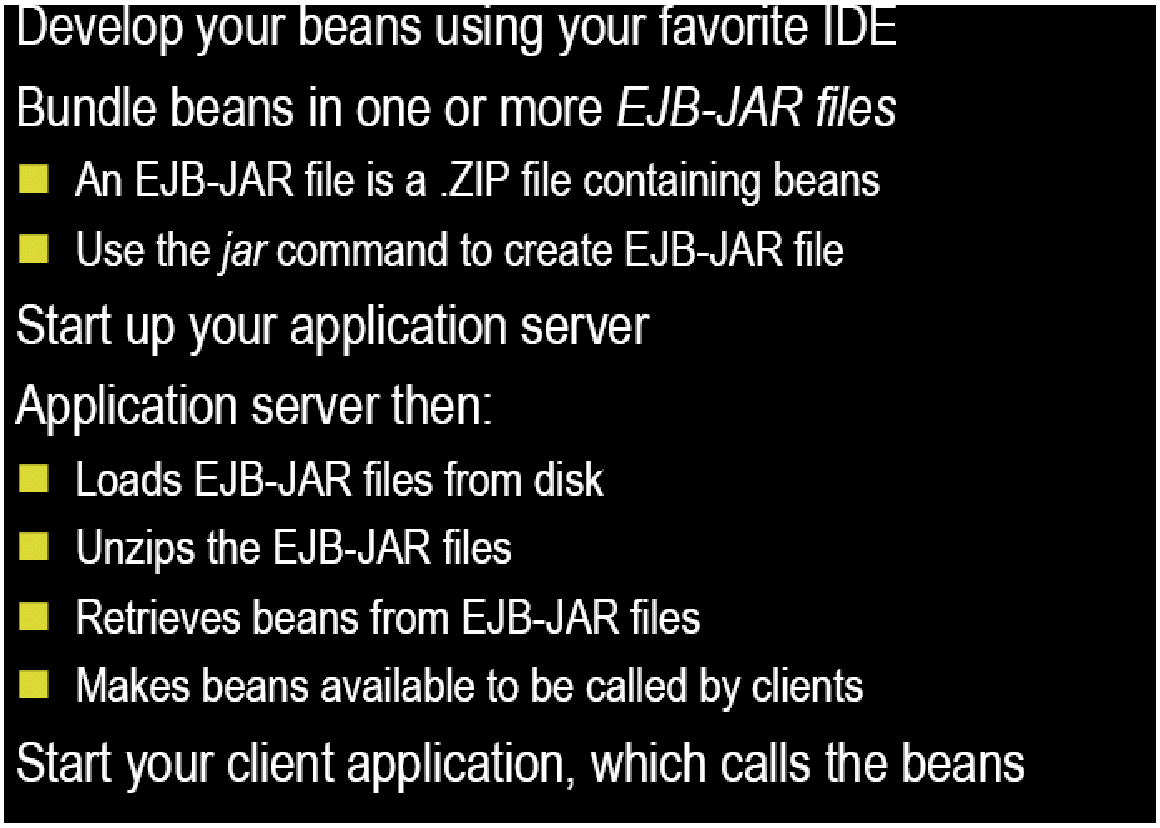
By Middleware Company , 2003



Develop your beans using your favorite IDE

Bundle beans in one or more *EJB-JAR files*

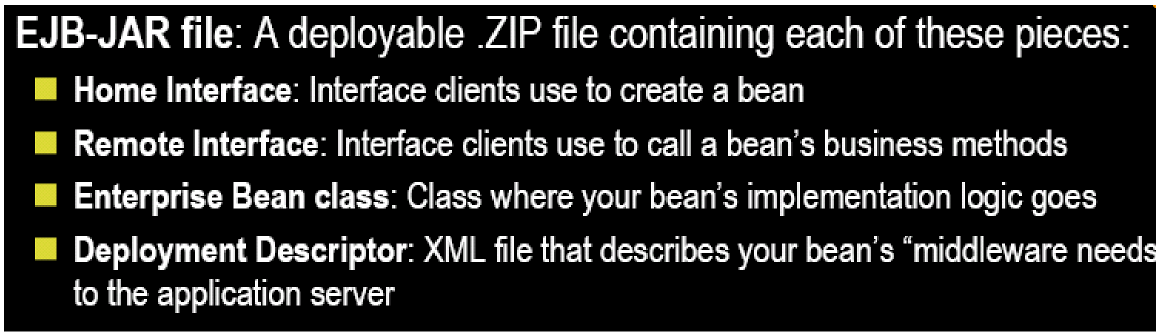
- An EJB-JAR file is a .ZIP file containing beans
- Use the *jar* command to create EJB-JAR file

Start up your application server

Application server then:

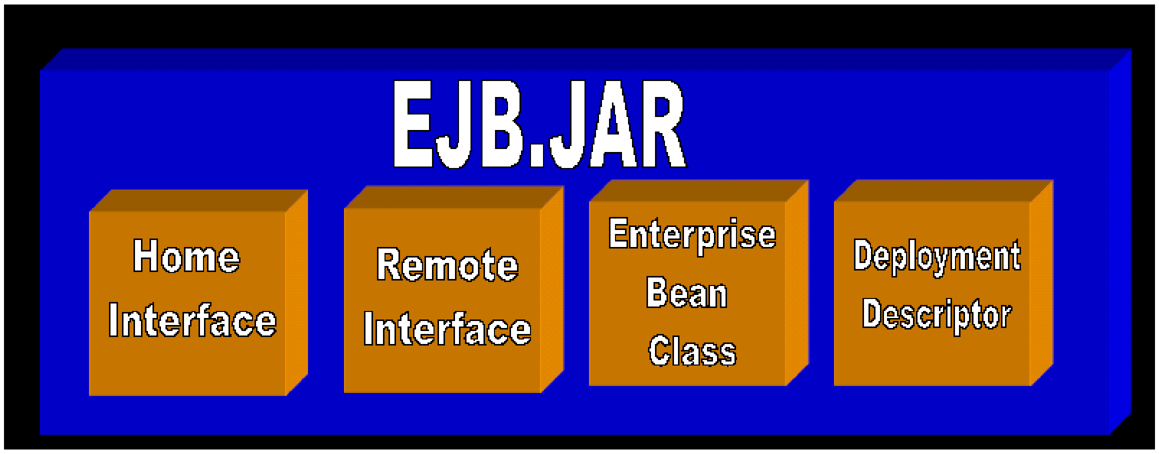
- Loads EJB-JAR files from disk
- Unzips the EJB-JAR files
- Retrieves beans from EJB-JAR files
- Makes beans available to be called by clients

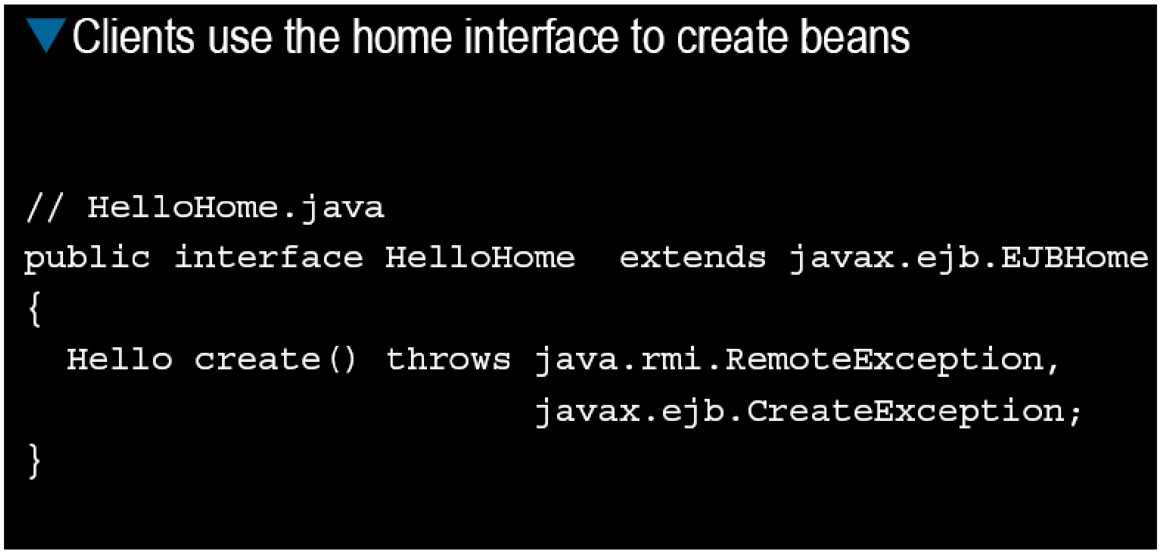
Start your client application, which calls the beans



EJB-JAR file: A deployable .ZIP file containing each of these pieces:

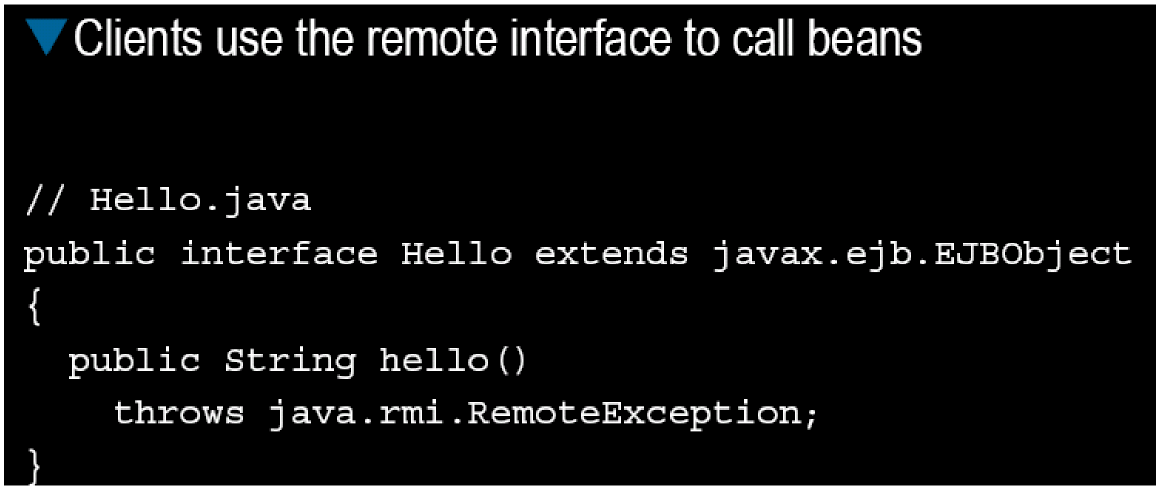
- **Home Interface**: Interface clients use to create a bean
- **Remote Interface**: Interface clients use to call a bean's business methods
- **Enterprise Bean class**: Class where your bean's implementation logic goes
- **Deployment Descriptor**: XML file that describes your bean's "middleware needs to the application server




Clients use the home interface to create beans

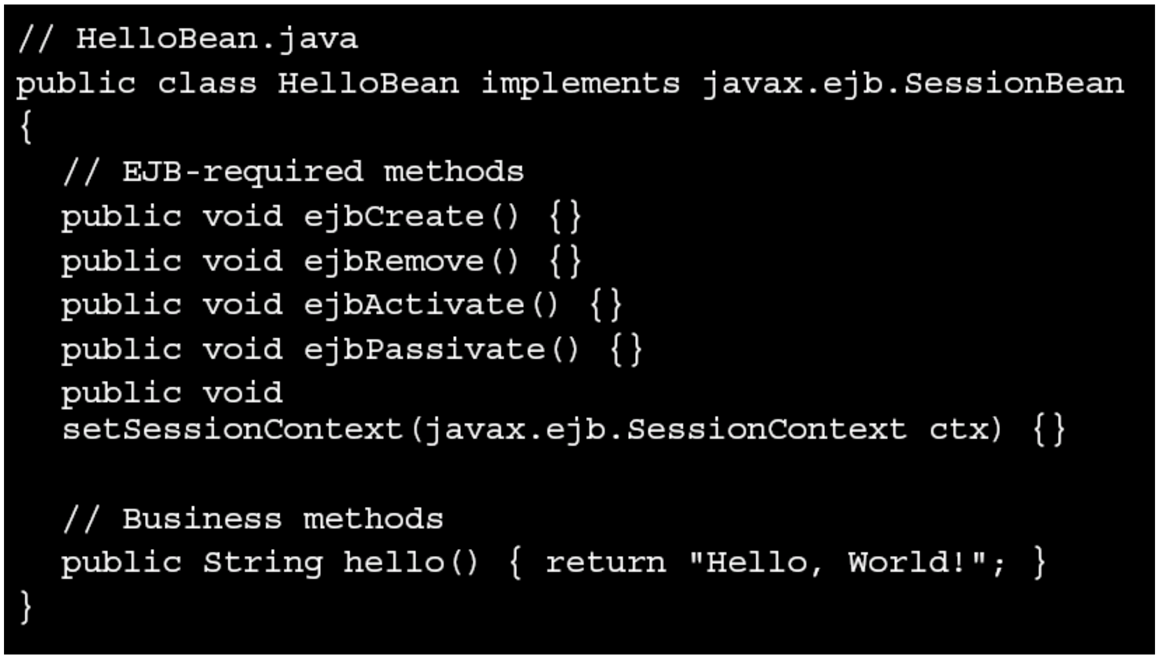
```
// HelloHome.java
public interface HelloHome  extends javax.ejb.EJBHome
{
  Hello create() throws java.rmi.RemoteException,
                        javax.ejb.CreateException;
}
```

HOME object


Clients use the remote interface to call beans

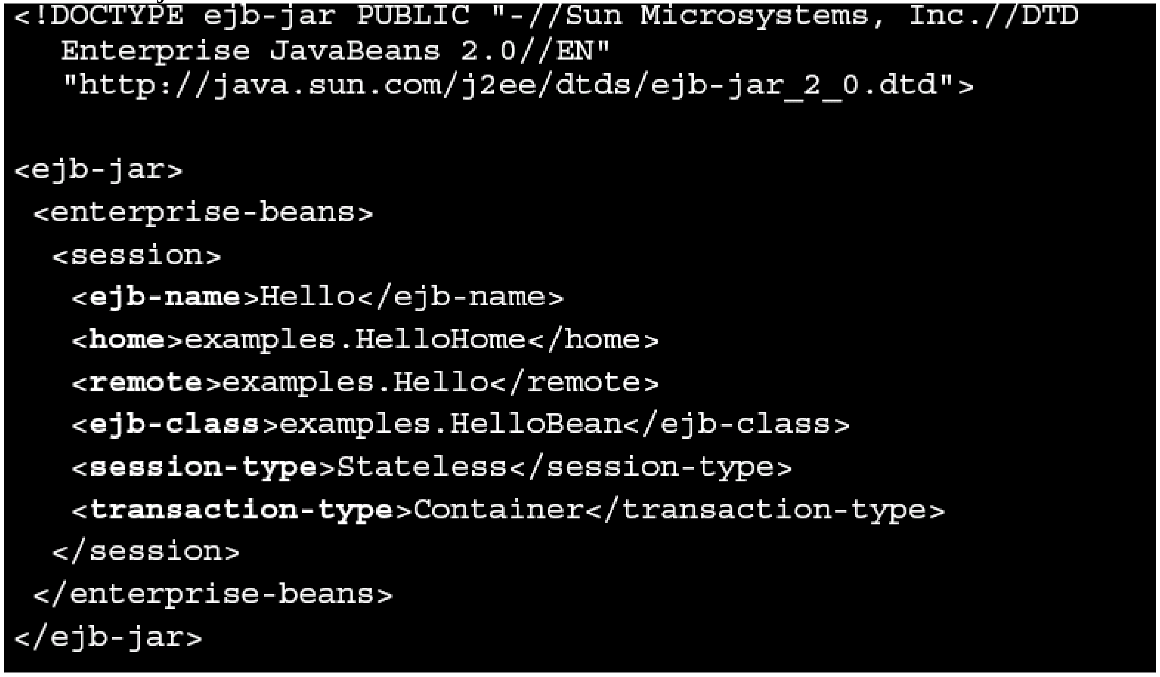
```
// Hello.java
public interface Hello extends javax.ejb.EJBObject
{
  public String hello()
    throws java.rmi.RemoteException;
}
```

Remote object



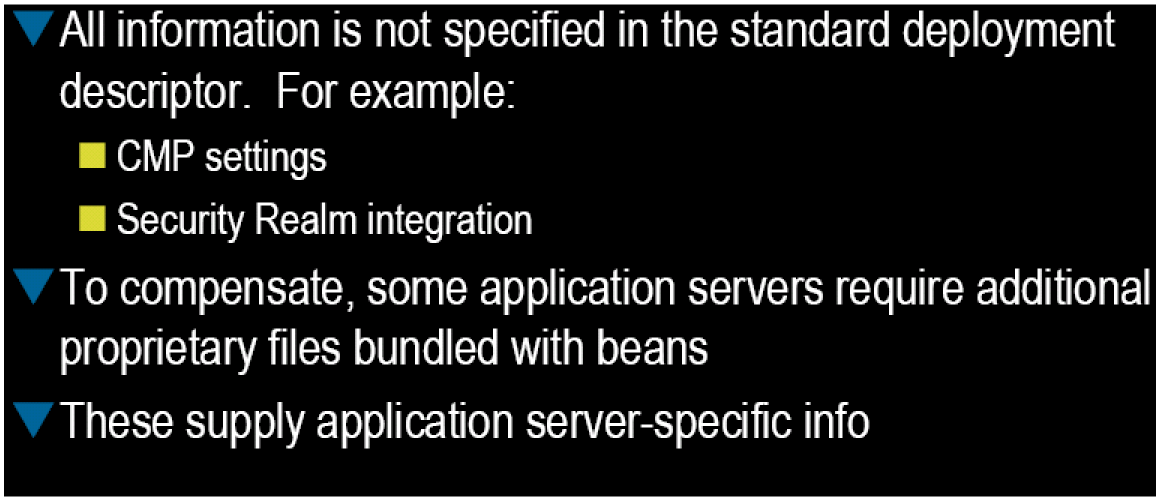
```java
// HelloBean.java
public class HelloBean implements javax.ejb.SessionBean
{
   // EJB-required methods
   public void ejbCreate() {}
   public void ejbRemove() {}
   public void ejbActivate() {}
   public void ejbPassivate() {}
   public void
   setSessionContext(javax.ejb.SessionContext ctx) {}

   // Business methods
   public String hello() { return "Hello, World!"; }
}
```

Business object

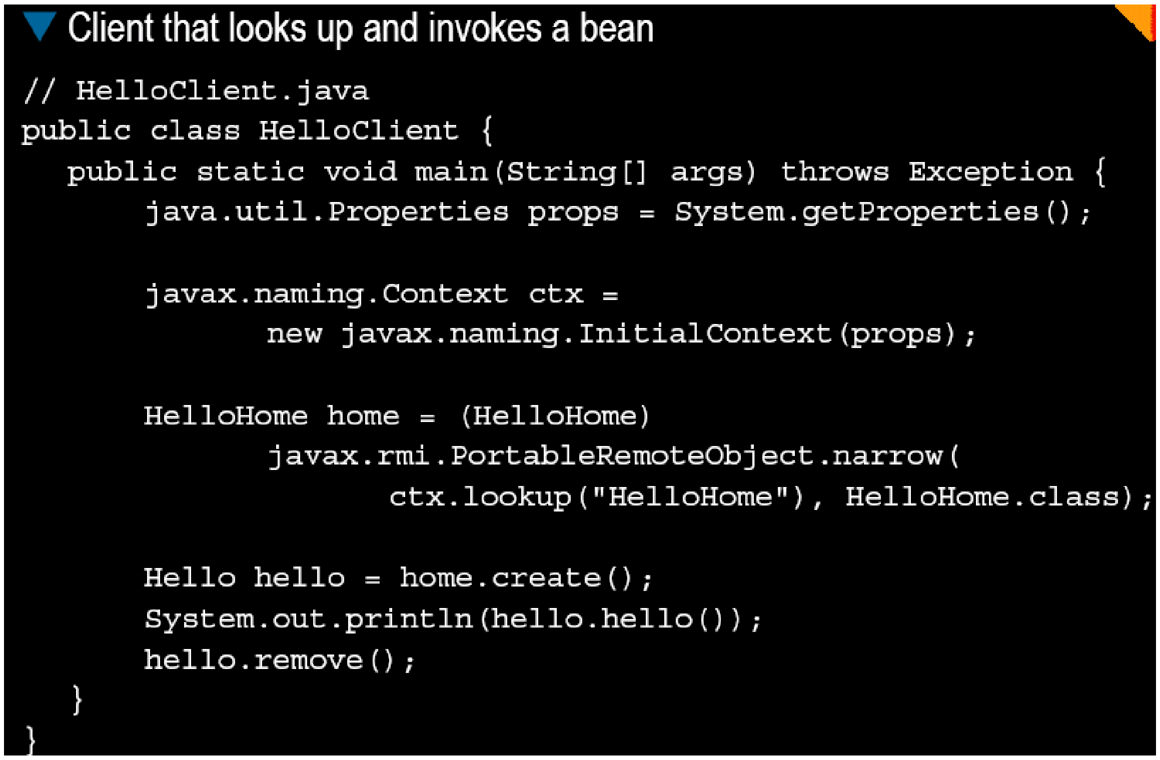


```xml
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD
   Enterprise JavaBeans 2.0//EN"
   "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">

<ejb-jar>
 <enterprise-beans>
  <session>
   <ejb-name>Hello</ejb-name>
   <home>examples.HelloHome</home>
   <remote>examples.Hello</remote>
   <ejb-class>examples.HelloBean</ejb-class>
   <session-type>Stateless</session-type>
   <transaction-type>Container</transaction-type>
  </session>
 </enterprise-beans>
</ejb-jar>
```

ejb-jar.xml Deployment Descriptor


- All information is not specified in the standard deployment descriptor. For example:
  - CMP settings
  - Security Realm integration
- To compensate, some application servers require additional proprietary files bundled with beans
- These supply application server-specific info

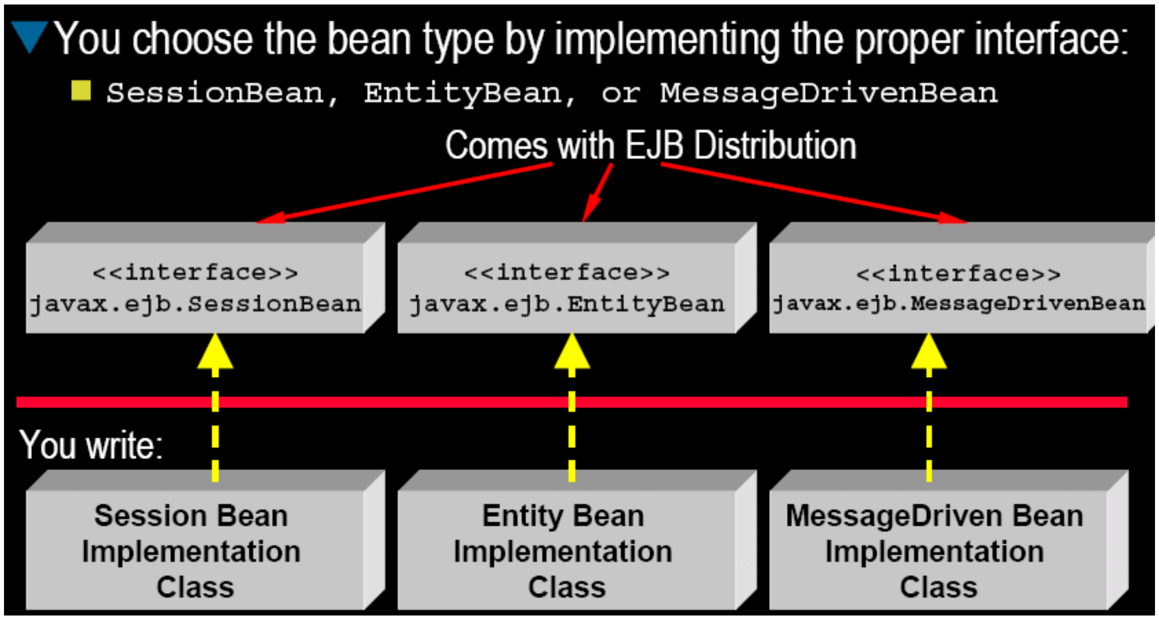
Also Application Server specific descriptors (vendor specific, caching nature, db connection mapping and pooling etc)


- Client that looks up and invokes a bean

```
// HelloClient.java
public class HelloClient {
   public static void main(String[] args) throws Exception {
       java.util.Properties props = System.getProperties();

       javax.naming.Context ctx =
              new javax.naming.InitialContext(props);

       HelloHome home = (HelloHome)
              javax.rmi.PortableRemoteObject.narrow(
                     ctx.lookup("HelloHome"), HelloHome.class);

       Hello hello = home.create();
       System.out.println(hello.hello());
       hello.remove();
   }
}
```

EJB Client

1. Creates Properties object to reach JNDI server
2. Perform lookup for HOME object
3. Create object
4. Invoke method on object
5. Remove cleans

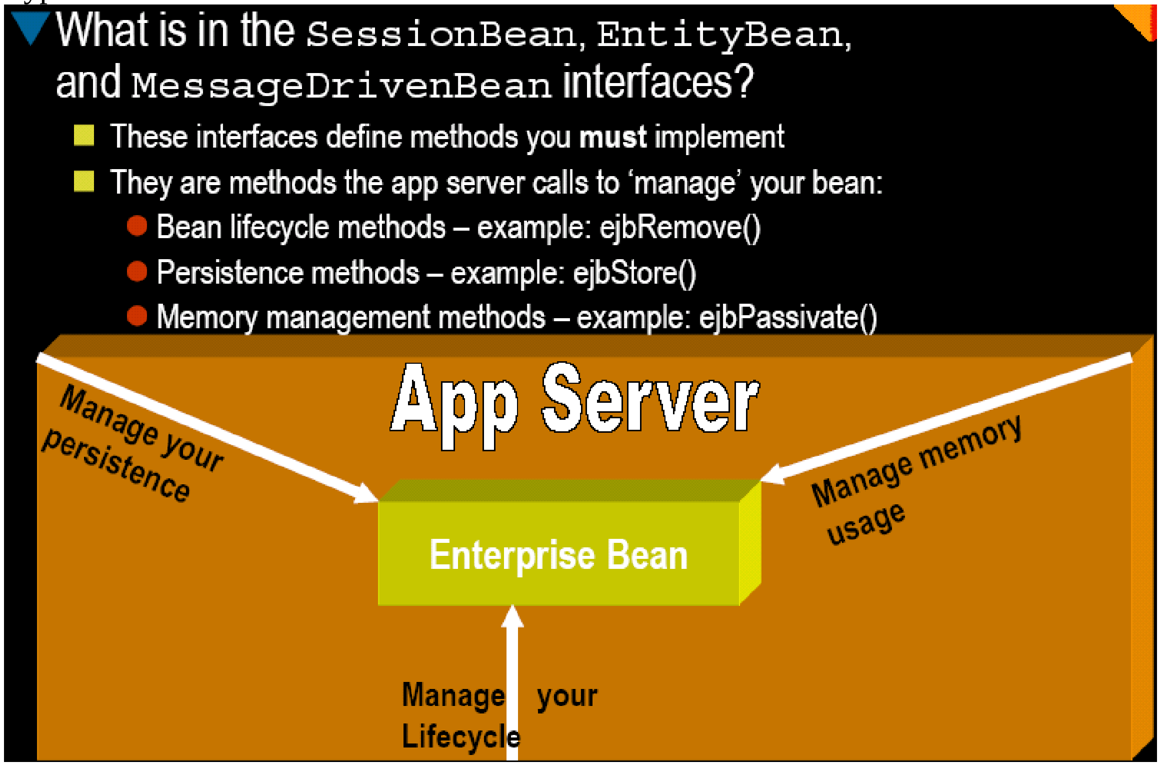


You choose the bean type by implementing the proper interface:

- SessionBean, EntityBean, or MessageDrivenBean

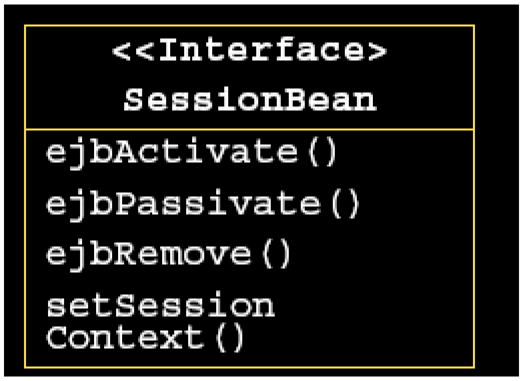
Comes with EJB Distribution
<<interface>> javax.ejb.SessionBean
<<interface>> javax.ejb.EntityBean
<<interface>> javax.ejb.MessageDrivenBean
You write:
Session Bean Implementation Class
Entity Bean Implementation Class
MessageDriven Bean Implementation Class

Types of Beans

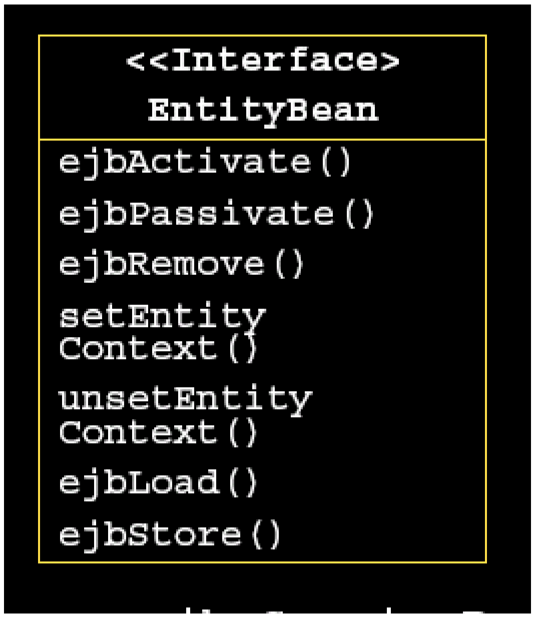


What is in the SessionBean, EntityBean, and MessageDrivenBean interfaces?

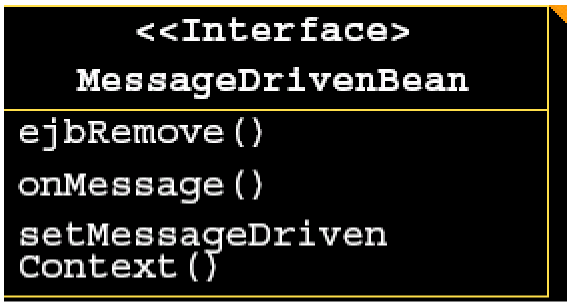
- These interfaces define methods you **must** implement
- They are methods the app server calls to 'manage' your bean:
  - Bean lifecycle methods – example: ejbRemove()
  - Persistence methods – example: ejbStore()
  - Memory management methods – example: ejbPassivate()



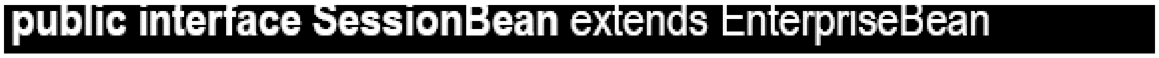
Session Bean API

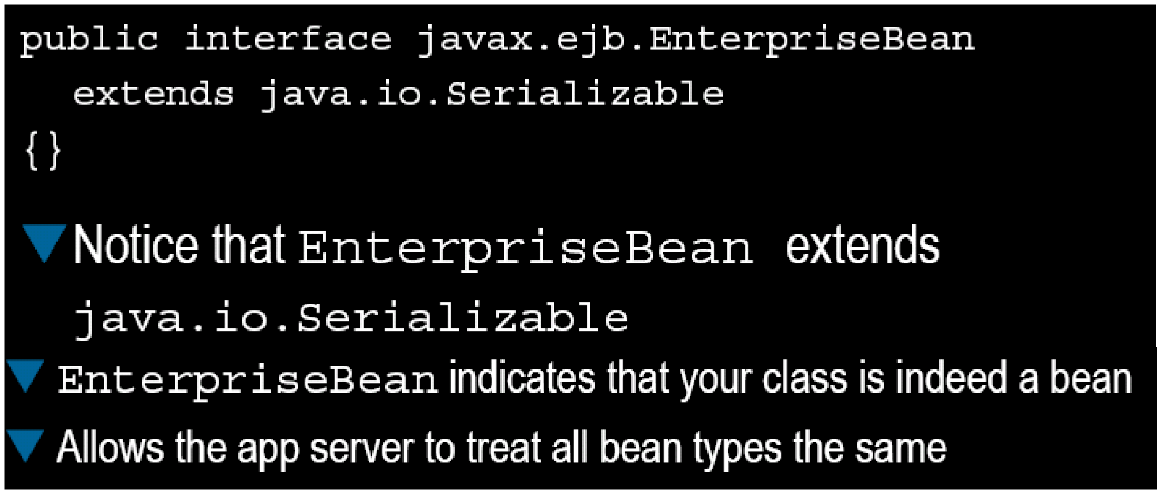


Entity Bean API
ejbLoad and ejbStore to refresh state



MDB API

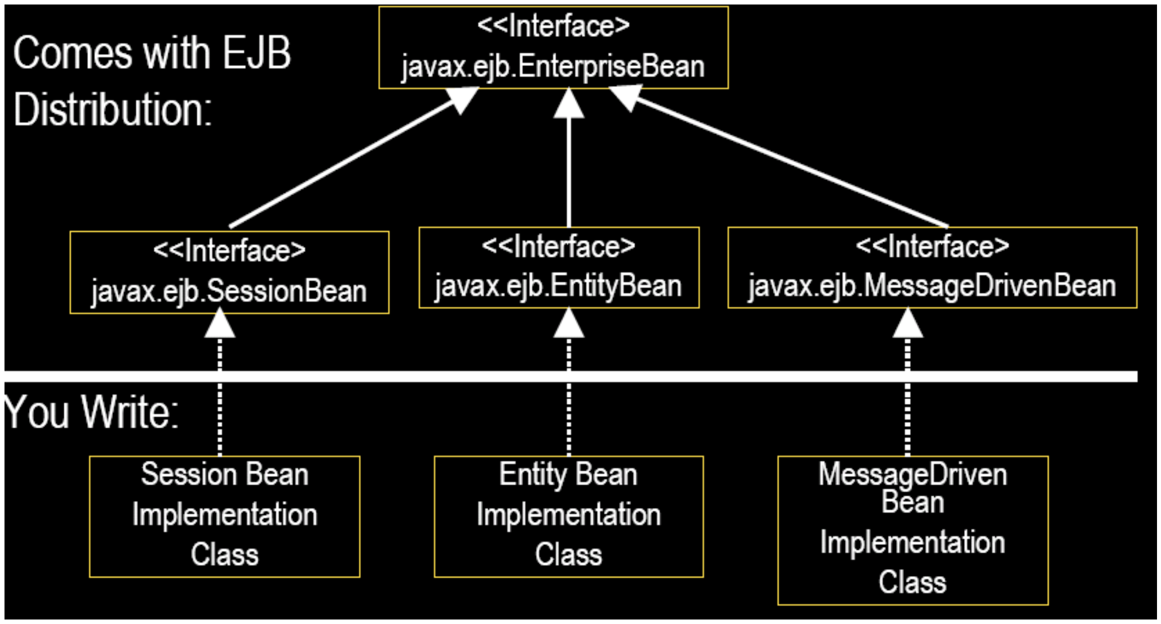

```
public interface SessionBean extends EnterpriseBean
```
And it is empty



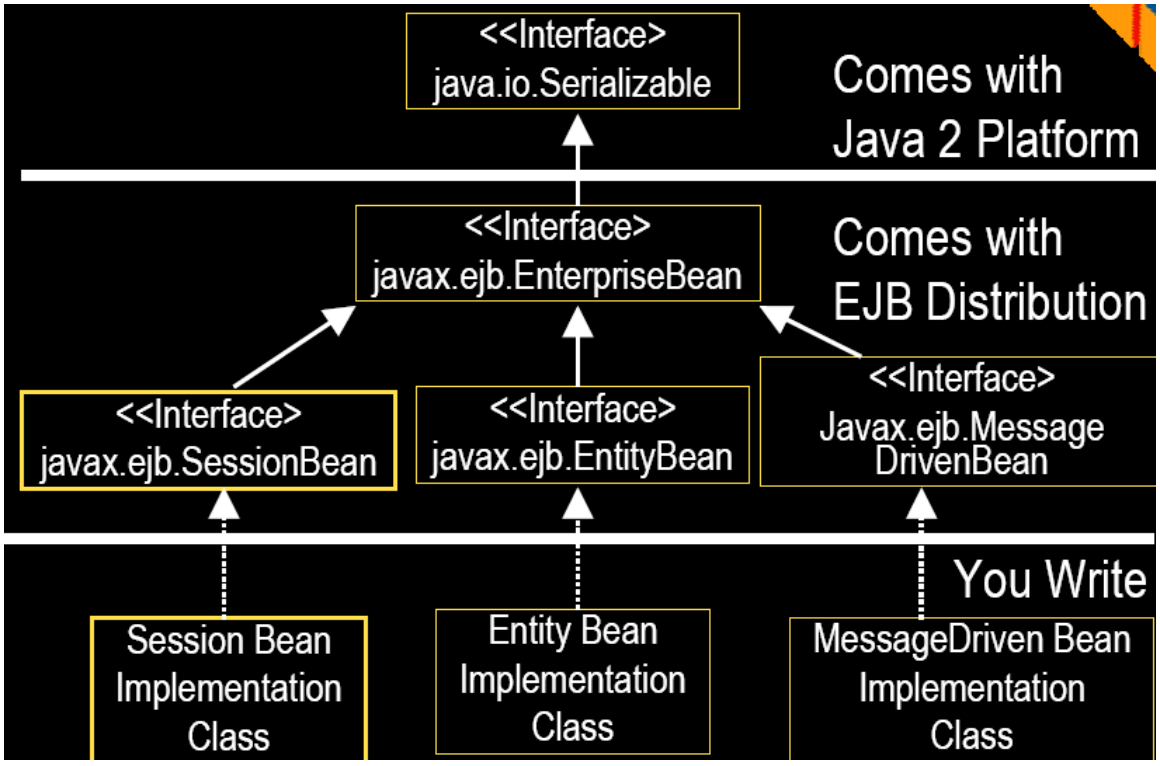
```
public interface javax.ejb.EnterpriseBean
   extends java.io.Serializable
{}
```

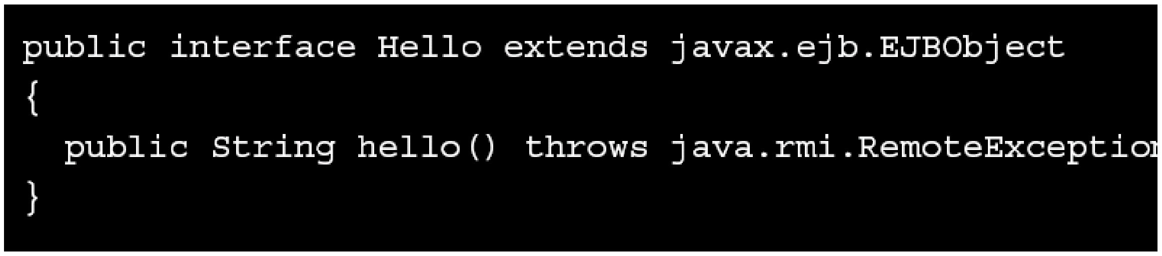
- Notice that `EnterpriseBean` extends `java.io.Serializable`
- `EnterpriseBean` indicates that your class is indeed a bean
- Allows the app server to treat all bean types the same

Serializable allows to persist object and send over the wire


Comes with EJB Distribution:
<<Interface>> javax.ejb.EnterpriseBean
<<Interface>> javax.ejb.SessionBean
<<Interface>> javax.ejb.EntityBean
<<Interface>> javax.ejb.MessageDrivenBean
You Write:
Session Bean Implementation Class
Entity Bean Implementation Class
MessageDriven Bean Implementation Class

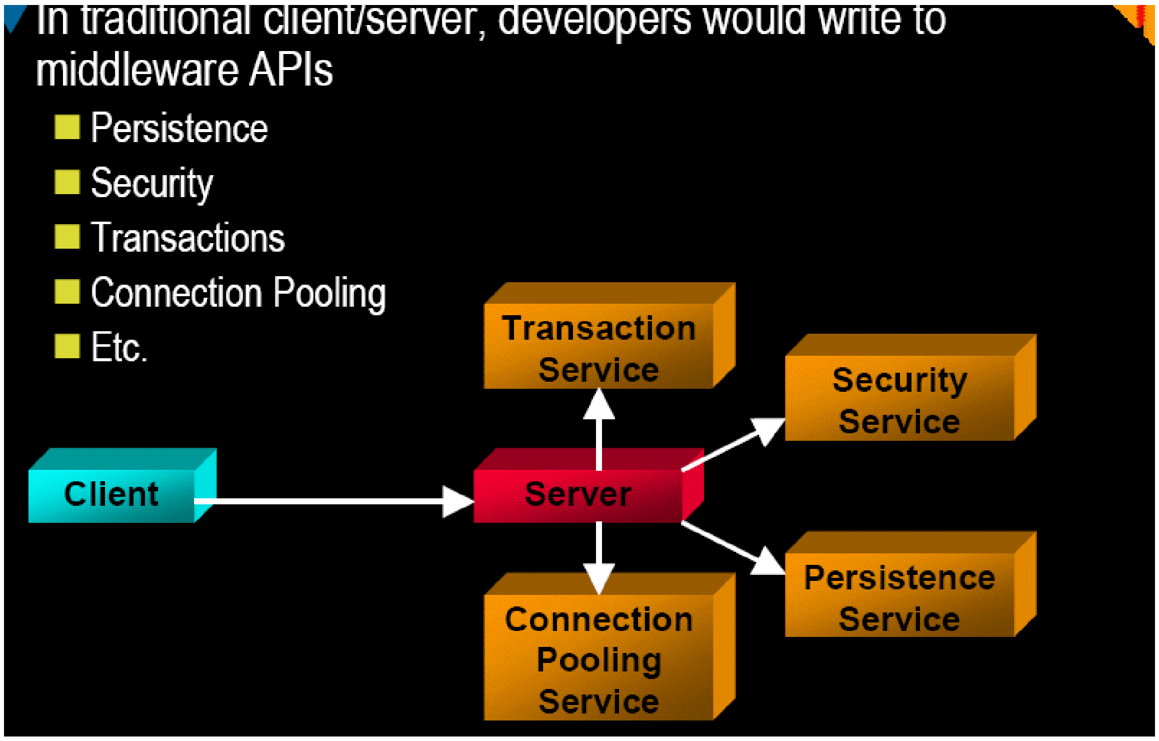
But you never directly talk to the EJB but only to its remote interface. Container can passivate and activate EJBs and it uses serializable nature of the EJBs.



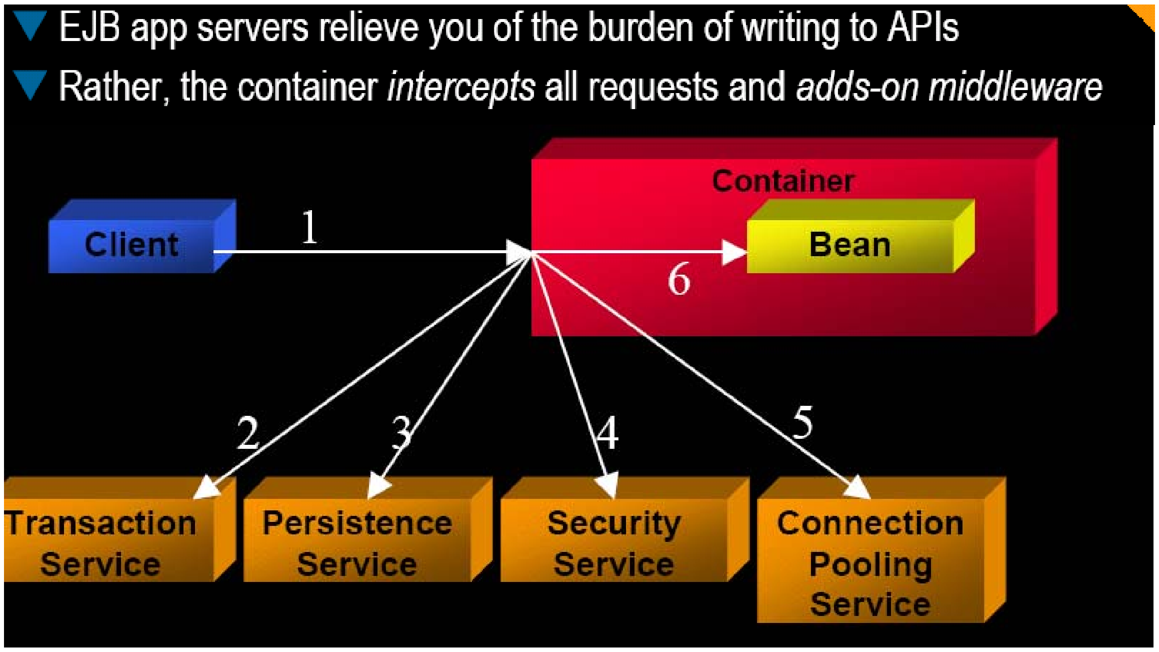


```
public interface Hello extends javax.ejb.EJBObject
{
  public String hello() throws java.rmi.RemoteExceptio
}
```

extends EJBObject!

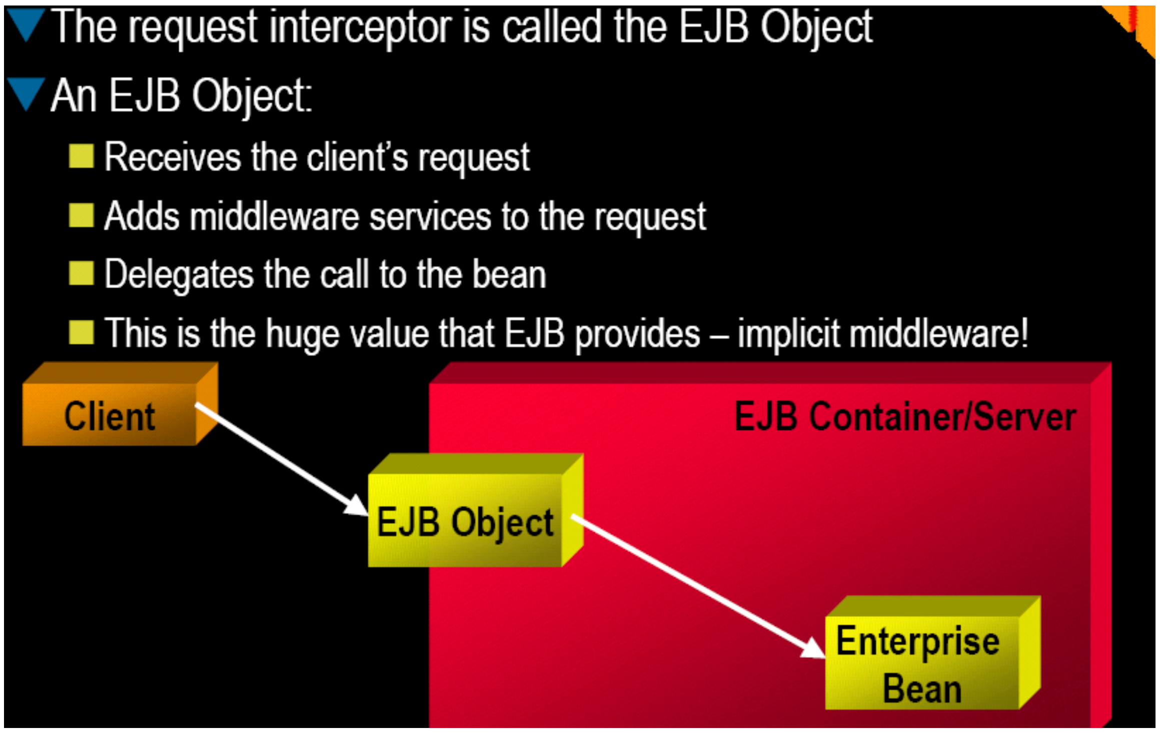


- In traditional client/server, developers would write to middleware APIs
  - Persistence
  - Security
  - Transactions
  - Connection Pooling
  - Etc.



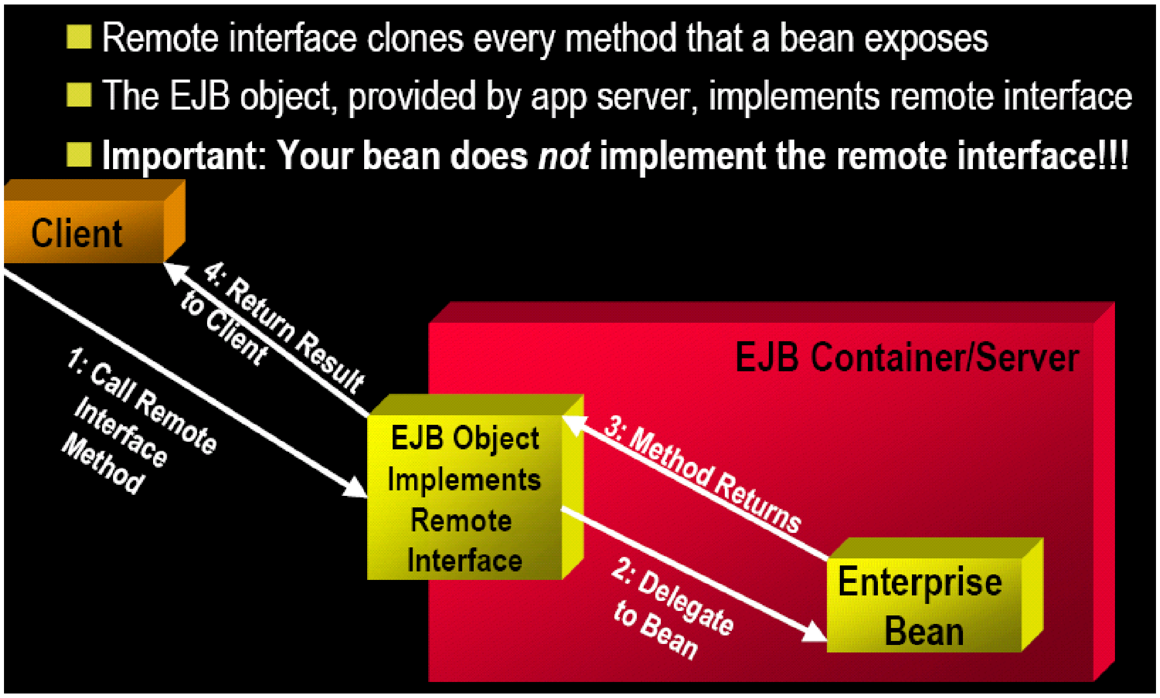
- EJB app servers relieve you of the burden of writing to APIs
- Rather, the container *intercepts* all requests and *adds-on middleware*

Intercepts client call and invokes all services before accessing the bean, including security, starting transactions etc.



- The request interceptor is called the EJB Object
- An EJB Object:
  - Receives the client's request
  - Adds middleware services to the request
  - Delegates the call to the bean
  - This is the huge value that EJB provides – implicit middleware!

EJB Object is an interceptor, Remote Interface extends EJB objects.
EJBObjects has all services.
Client talks to the EJB container

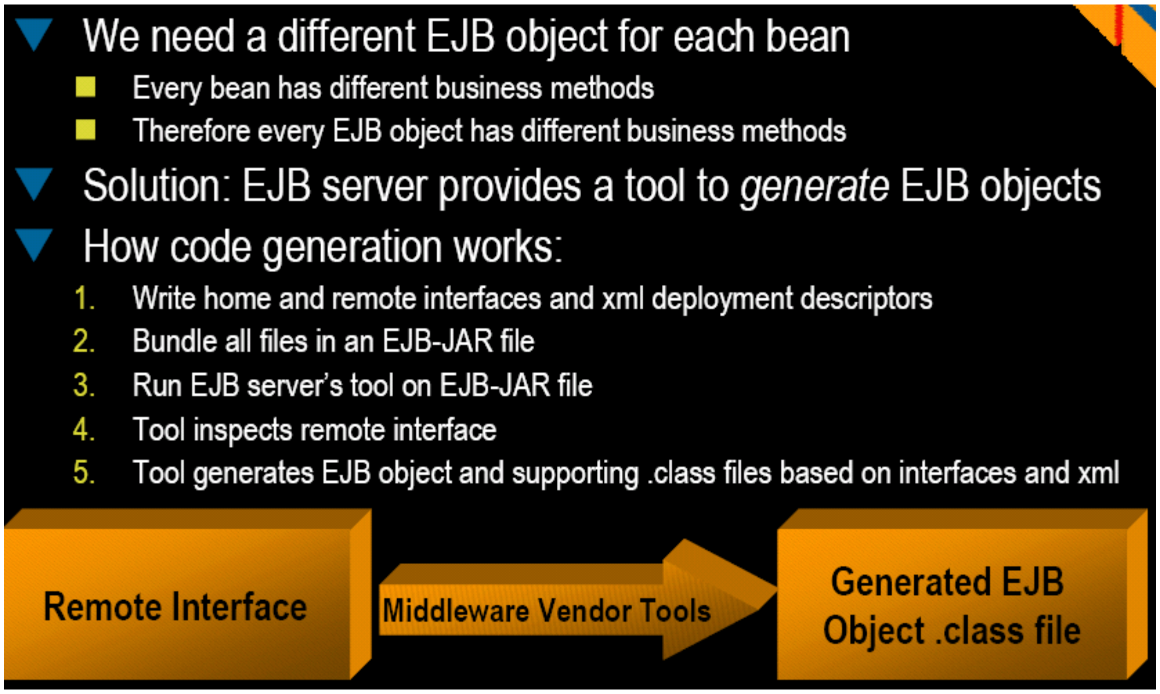


- Remote interface clones every method that a bean exposes
- The EJB object, provided by app server, implements remote interface
- **Important: Your bean does *not* implement the remote interface!!!**

Client
4: Return Result to Client
1: Call Remote Interface Method
EJB Container/Server
EJB Object Implements Remote Interface
3: Method Returns
2: Delegate to Bean
Enterprise Bean

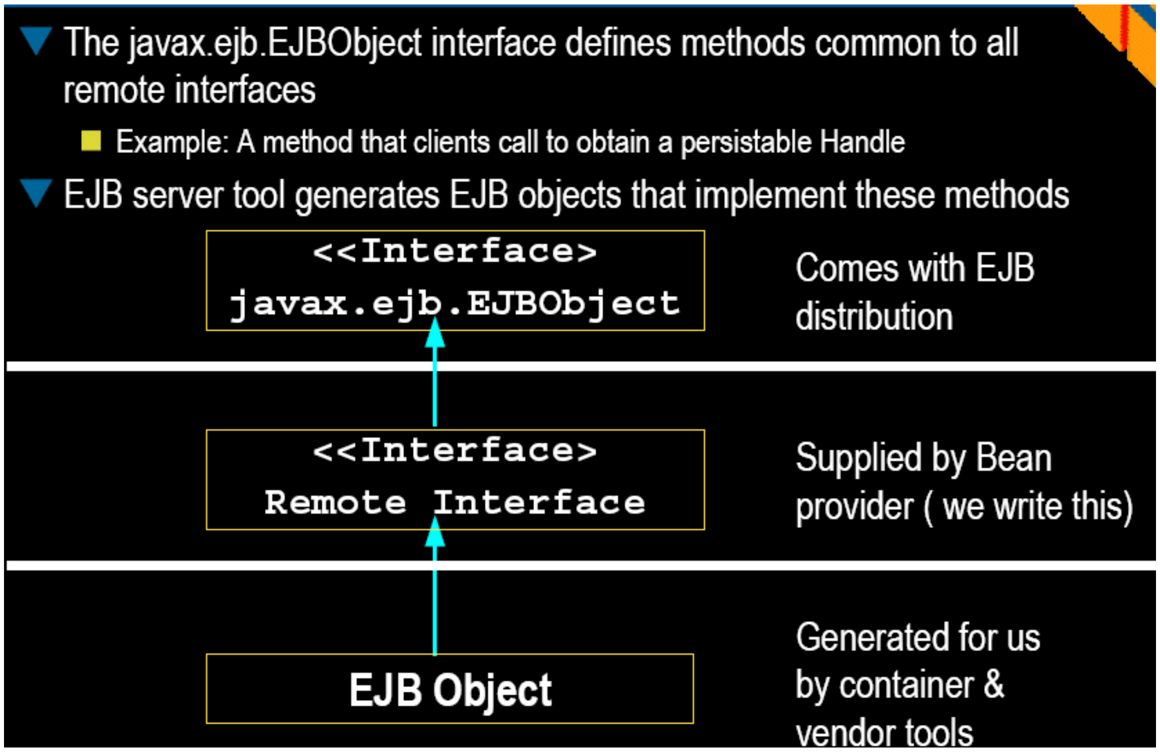
Client calls to the EJB Object via Remote Interface
Layer of indirection

EJB Objects are Generated Files!



- We need a different EJB object for each bean
  - Every bean has different business methods
  - Therefore every EJB object has different business methods
- Solution: EJB server provides a tool to *generate* EJB objects
- How code generation works:
  1. Write home and remote interfaces and xml deployment descriptors
  2. Bundle all files in an EJB-JAR file
  3. Run EJB server's tool on EJB-JAR file
  4. Tool inspects remote interface
  5. Tool generates EJB object and supporting .class files based on interfaces and xml

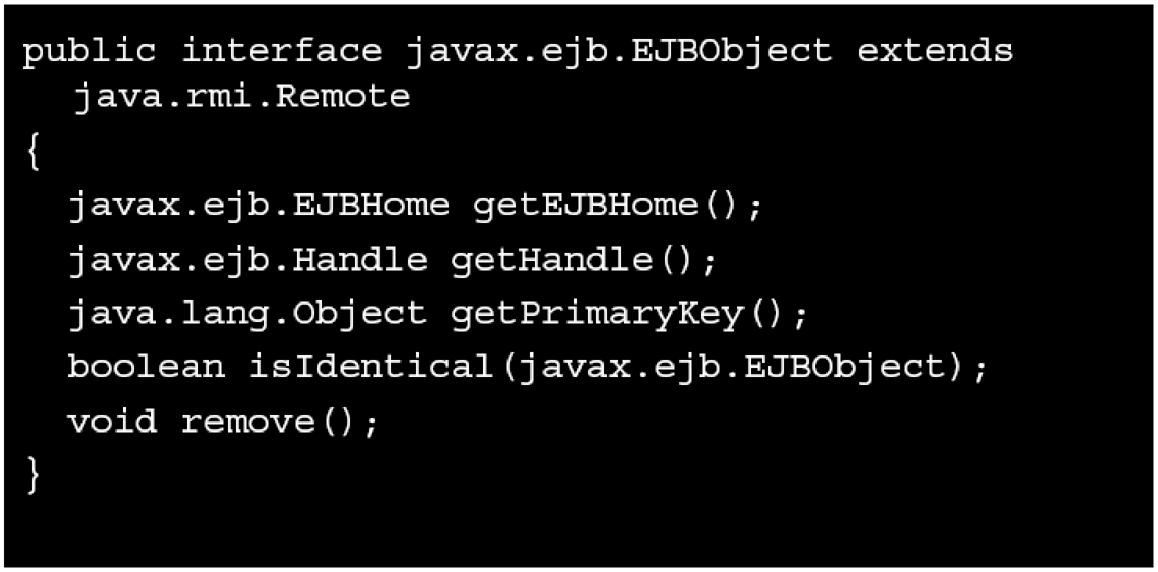
Remote Interface
Middleware Vendor Tools
Generated EJB Object .class file



- The javax.ejb.EJBObject interface defines methods common to all remote interfaces
  - Example: A method that clients call to obtain a persistable Handle
- EJB server tool generates EJB objects that implement these methods

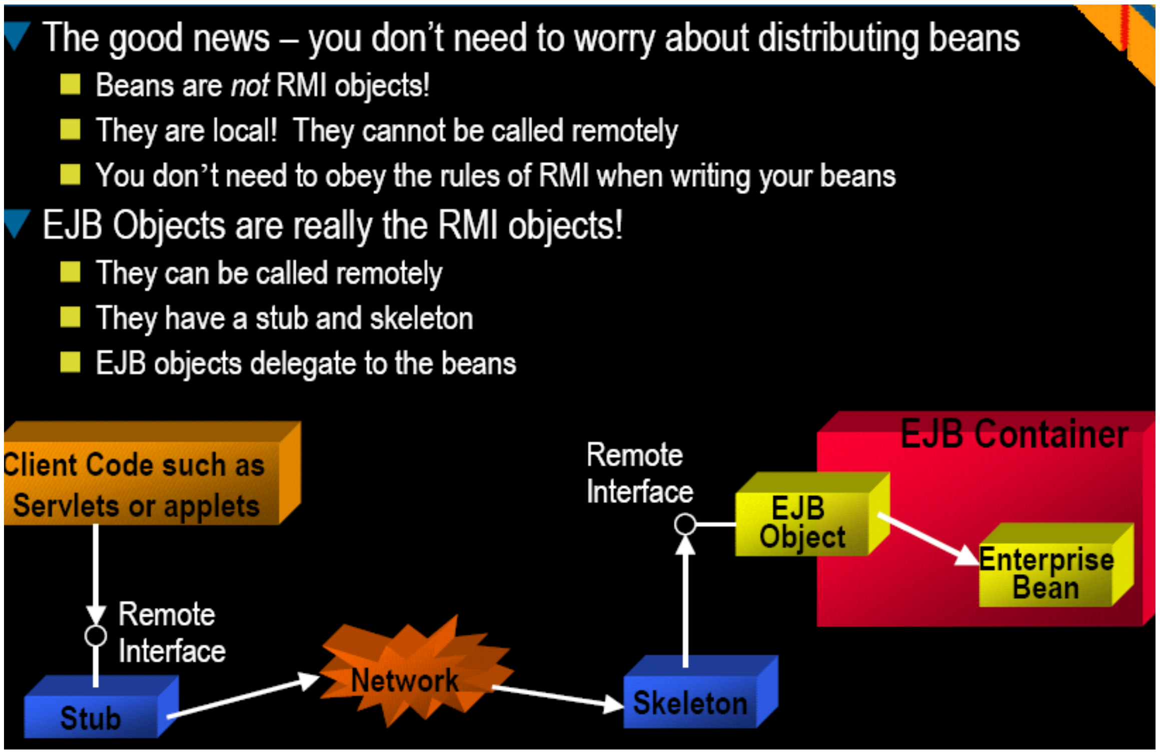
<<Interface>
javax.ejb.EJBObject
Comes with EJB distribution
<<Interface>
Remote Interface
Supplied by Bean provider (we write this)
EJB Object
Generated for us by container & vendor tools

EJB Object extends both!

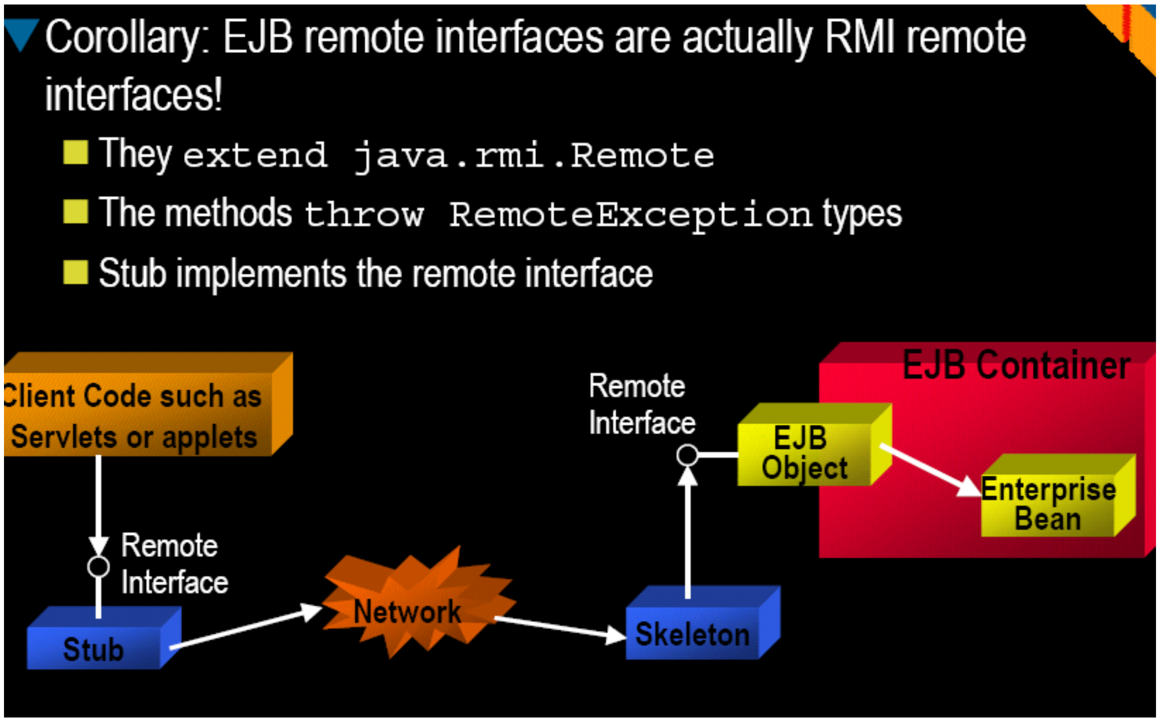


```
public interface javax.ejb.EJBObject extends
   java.rmi.Remote
{
   javax.ejb.EJBHome getEJBHome();
   javax.ejb.Handle getHandle();
   java.lang.Object getPrimaryKey();
   boolean isIdentical(javax.ejb.EJBObject);
   void remove();
}
```

EJBs are called remotely via RMI (RMI-IIOP for CORBA compatibility)

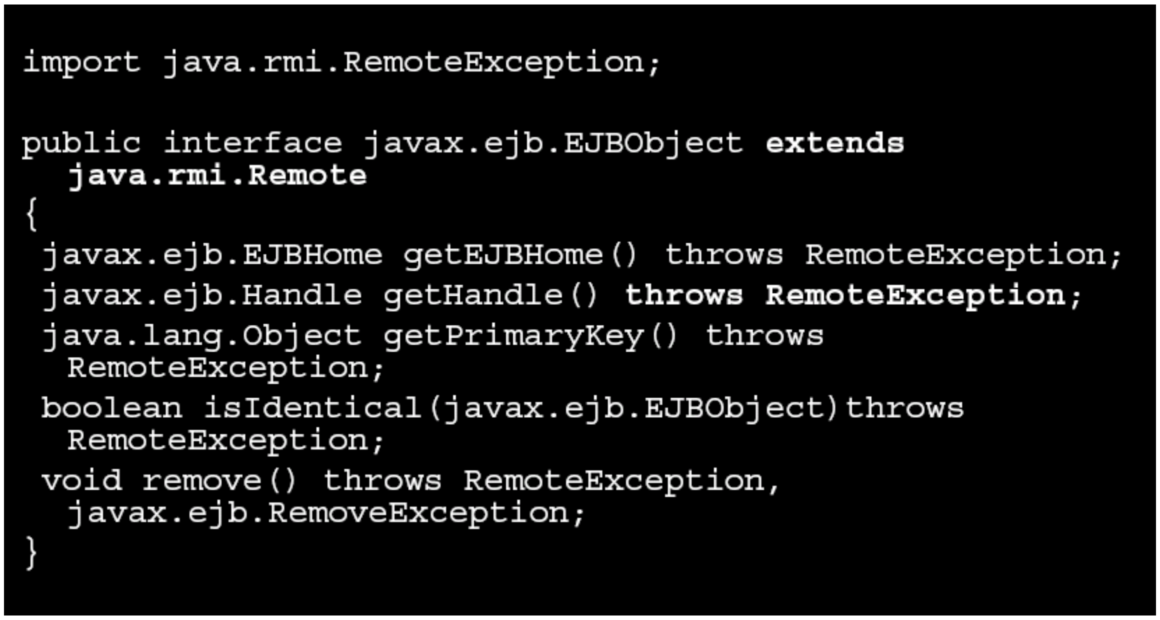


- The good news – you don't need to worry about distributing beans
  - Beans are *not* RMI objects!
  - They are local! They cannot be called remotely
  - You don't need to obey the rules of RMI when writing your beans
- EJB Objects are really the RMI objects!
  - They can be called remotely
  - They have a stub and skeleton
  - EJB objects delegate to the beans



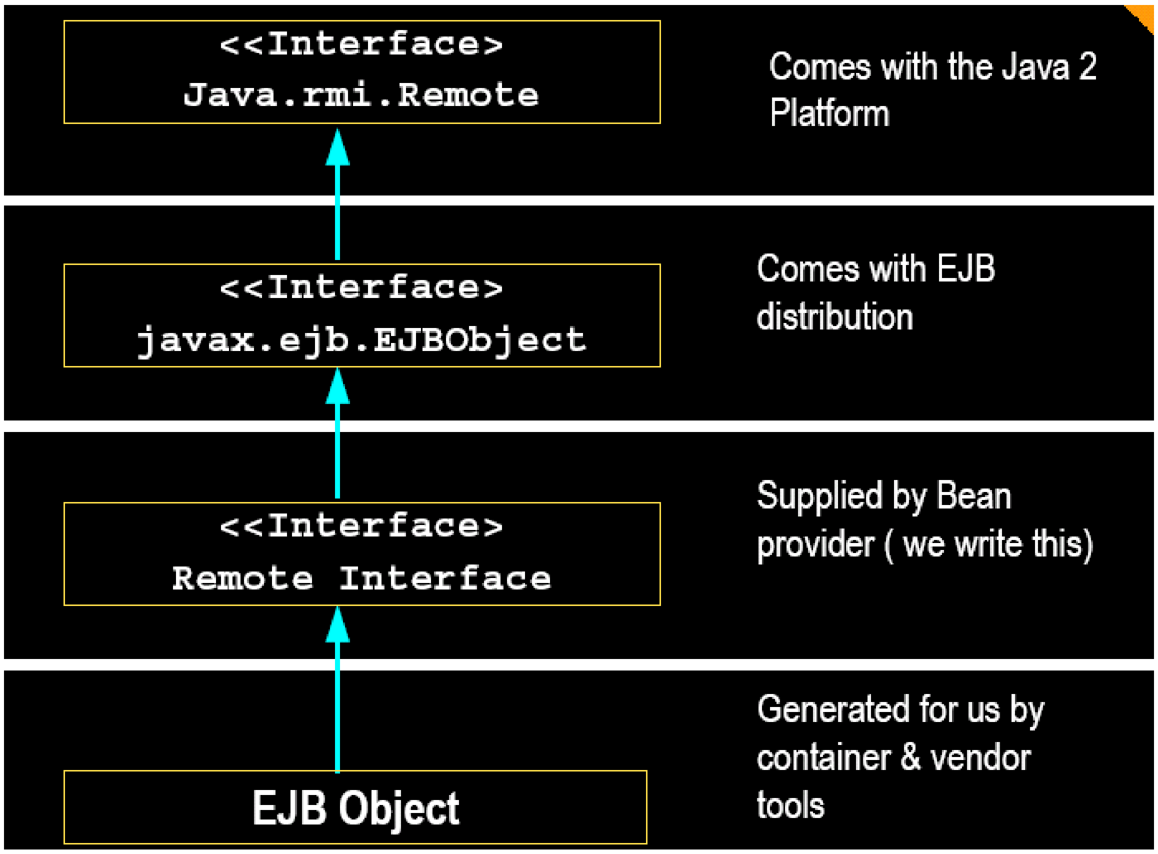
Corollary: EJB remote interfaces are actually RMI remote interfaces!

- They `extend java.rmi.Remote`
- The methods `throw RemoteException` types
- Stub implements the remote interface



```
import java.rmi.RemoteException;

public interface javax.ejb.EJBObject extends
   java.rmi.Remote
{
 javax.ejb.EJBHome getEJBHome() throws RemoteException;
 javax.ejb.Handle getHandle() throws RemoteException;
 java.lang.Object getPrimaryKey() throws
   RemoteException;
 boolean isIdentical(javax.ejb.EJBObject)throws
   RemoteException;
 void remove() throws RemoteException,
   javax.ejb.RemoveException;
}
```

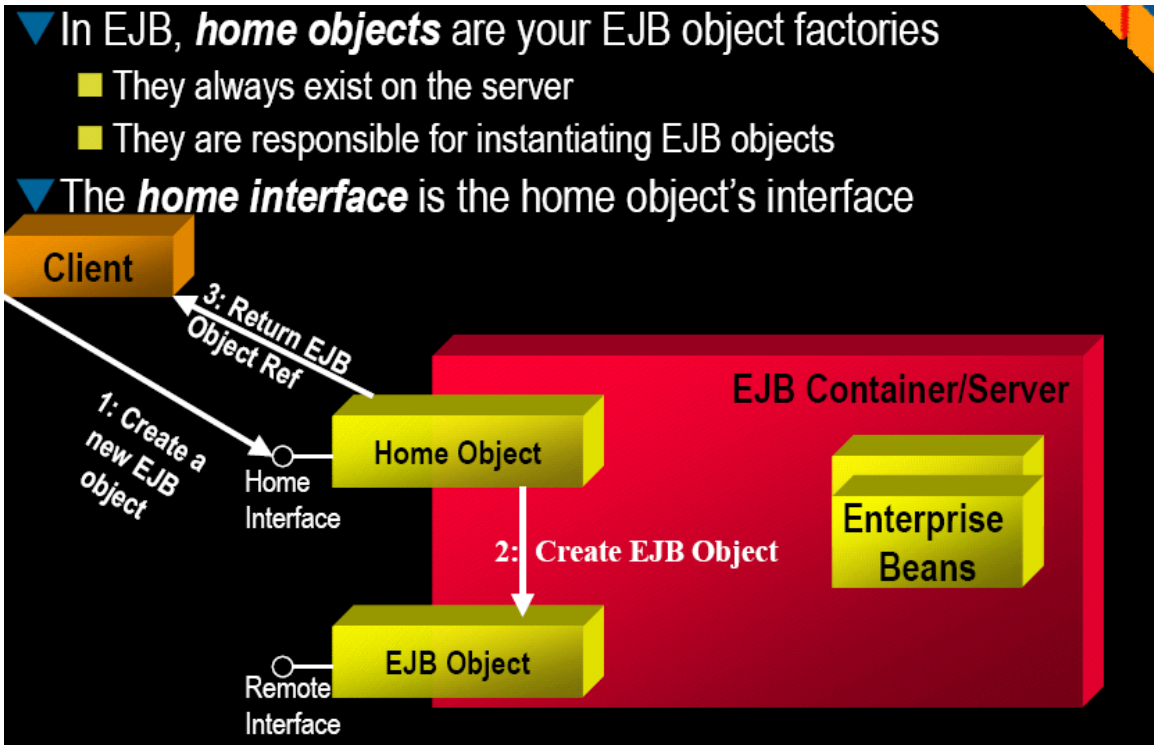
Remote Exception



<<Interface>>
Java.rmi.Remote
Comes with the Java 2 Platform
<<Interface>>
javax.ejb.EJBObject
Comes with EJB distribution
<<Interface>>
Remote Interface
Supplied by Bean provider (we write this)
EJB Object
Generated for us by container & vendor tools

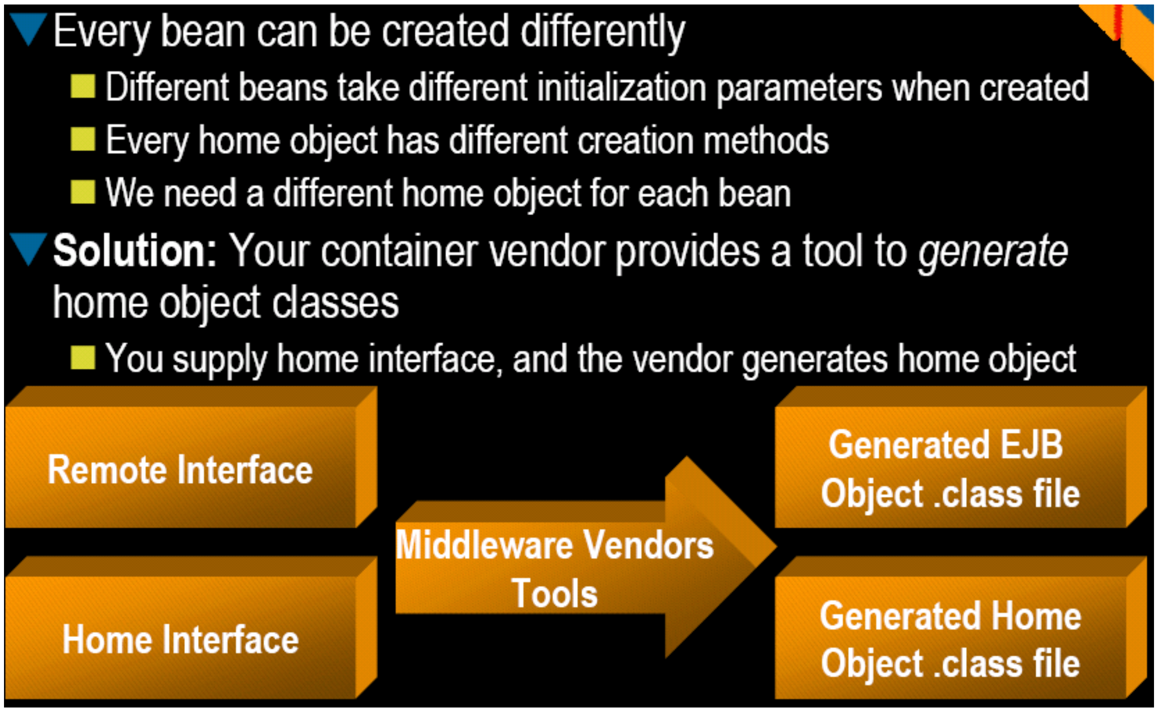
Remote Interface Summary

Home Interface is a factory to get proxy for the magic EJBObject
We cannot just instantiate EJB, it is container business



- In EJB, ***home objects*** are your EJB object factories
  - They always exist on the server
  - They are responsible for instantiating EJB objects
- The ***home interface*** is the home object's interface

We access HOME object via JNDI



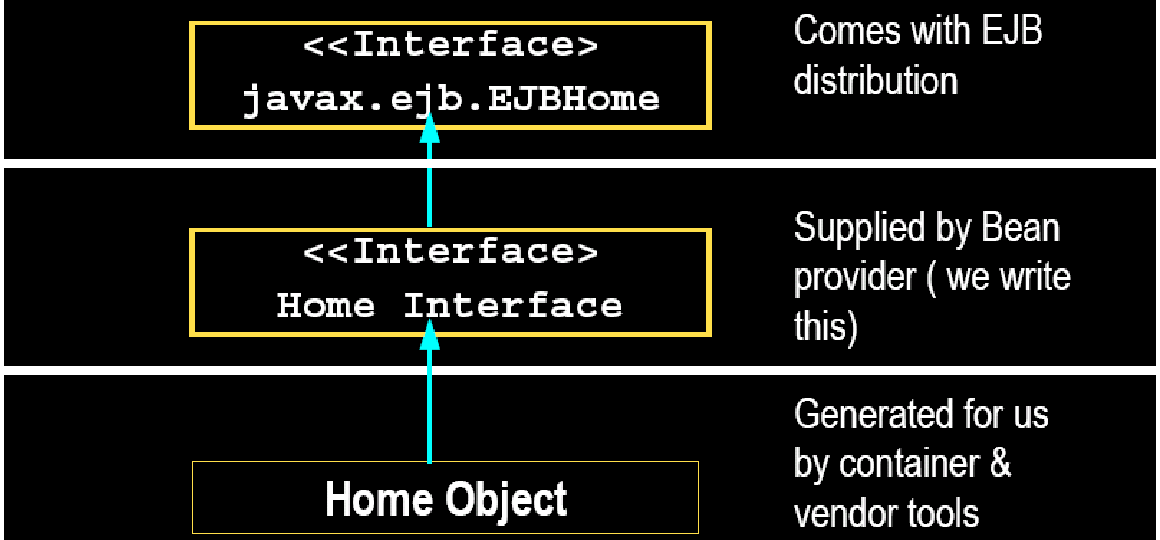
- Every bean can be created differently
  - Different beans take different initialization parameters when created
  - Every home object has different creation methods
  - We need a different home object for each bean
- **Solution:** Your container vendor provides a tool to *generate* home object classes
  - You supply home interface, and the vendor generates home object

Remote Interface
Home Interface
Middleware Vendors Tools
Generated EJB Object .class file
Generated Home Object .class file

Home Objects are also generated

Just like with EJB objects...

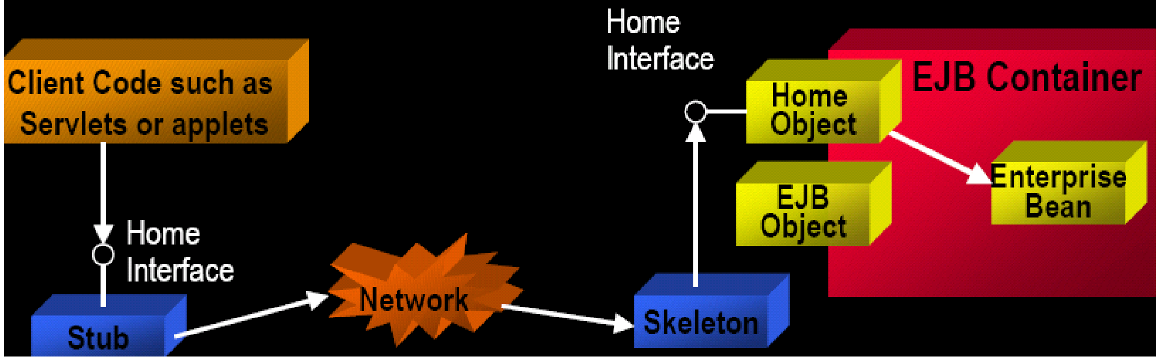
- There are some methods that all home objects should have
- Those methods are defined in javax.ejb.EJBHome

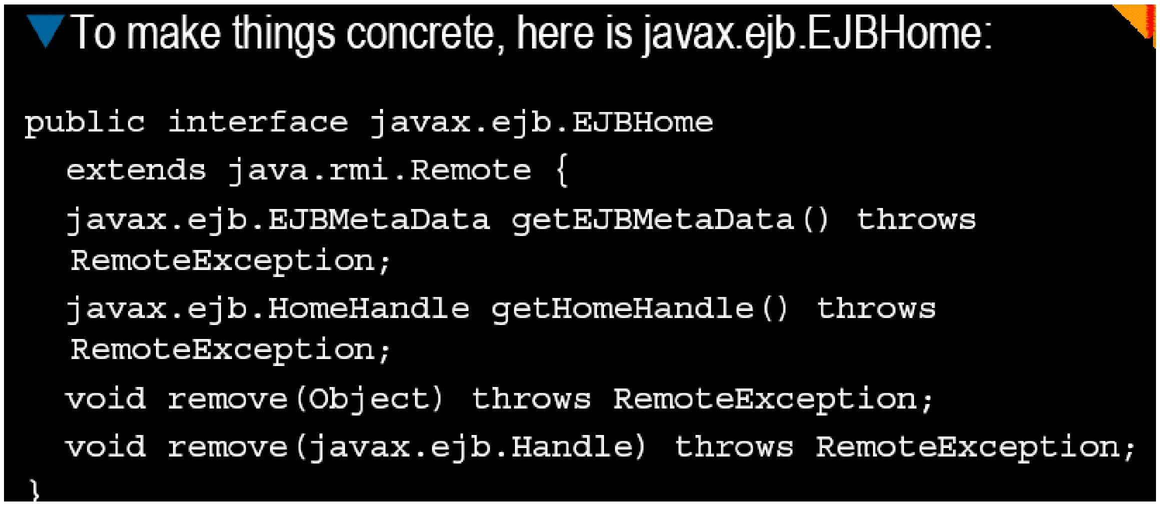


<<Interface>>
javax.ejb.EJBHome
Comes with EJB distribution
<<Interface>>
Home Interface
Supplied by Bean provider (we write this)
Home Object
Generated for us by container & vendor tools

Just like with EJB objects..

- Home Objects are also RMI objects
- Home Interfaces are also RMI remote interfaces

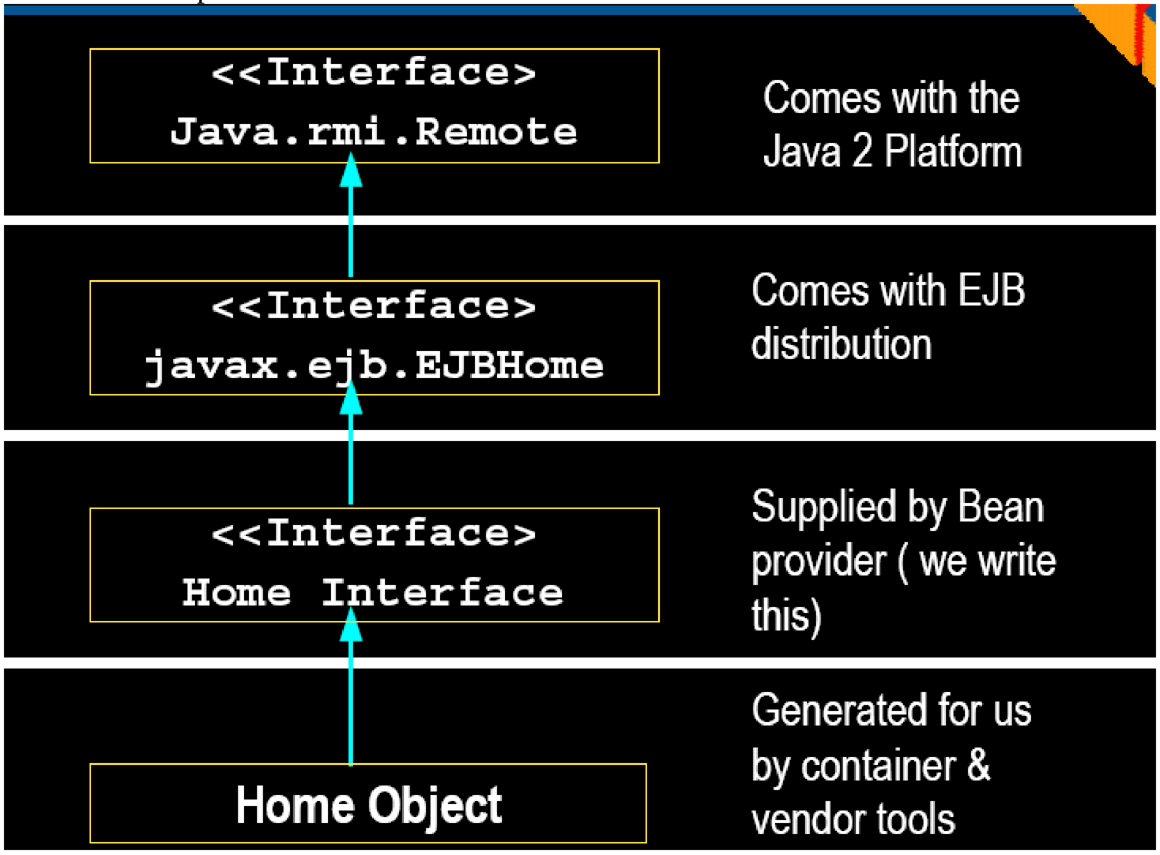




To make things concrete, here is javax.ejb.EJBHome:

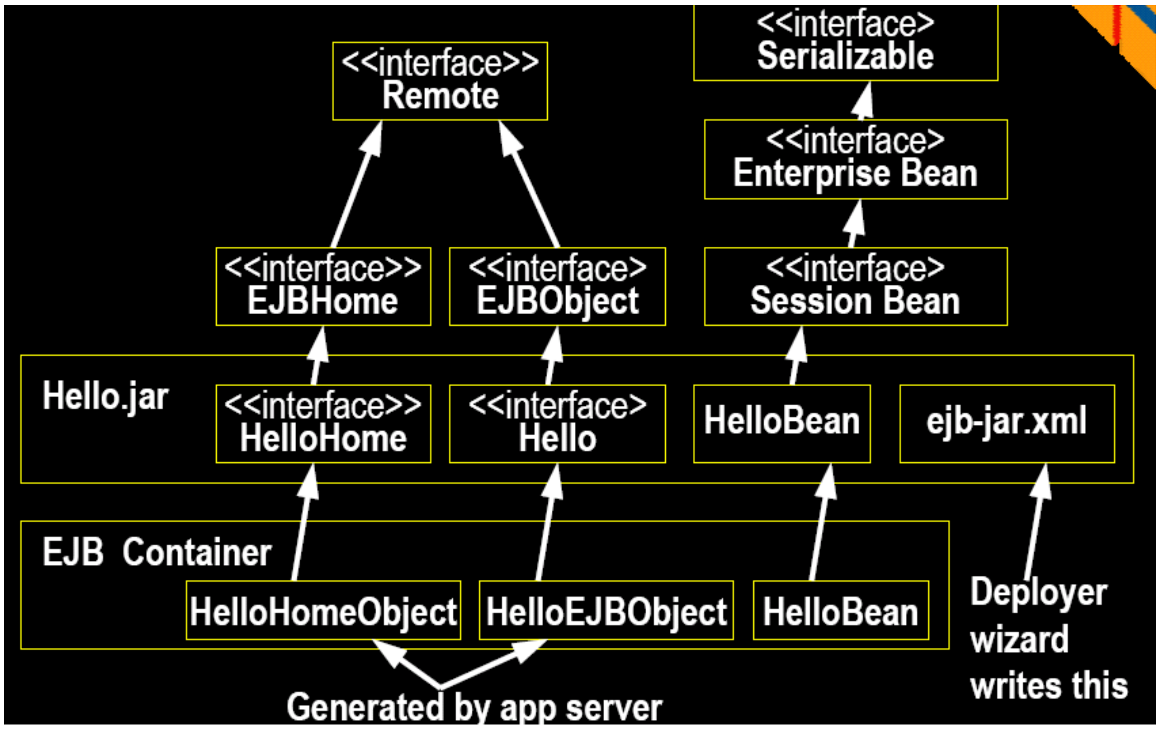
```
public interface javax.ejb.EJBHome
   extends java.rmi.Remote {
   javax.ejb.EJBMetaData getEJBMetaData() throws
    RemoteException;
   javax.ejb.HomeHandle getHomeHandle() throws
    RemoteException;
   void remove(Object) throws RemoteException;
   void remove(javax.ejb.Handle) throws RemoteException;
}
```

Each home is specific

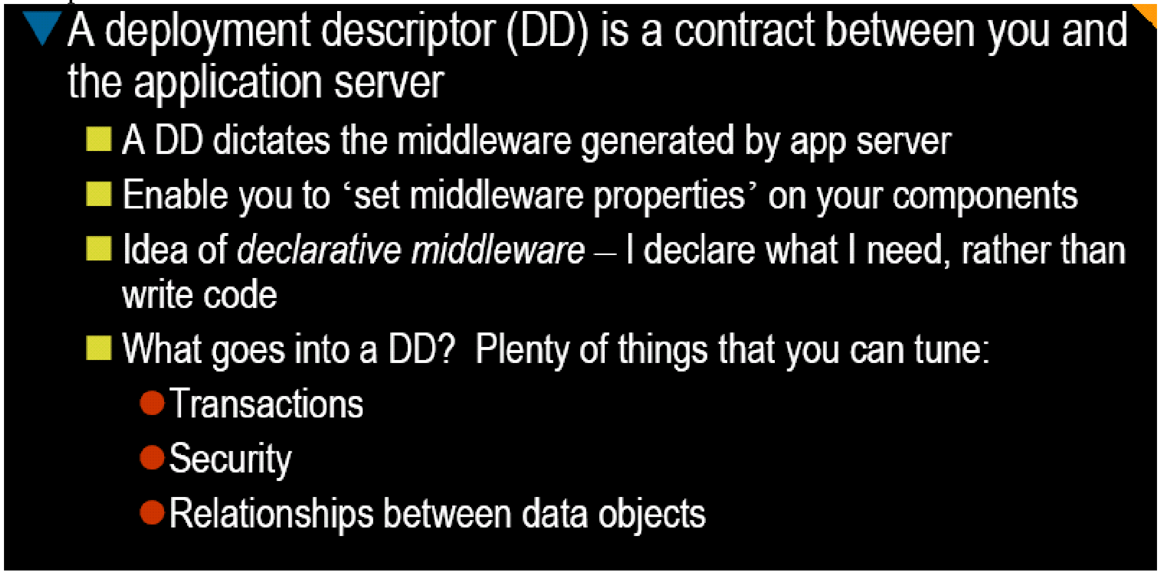


<<Interface>>
Java.rmi.Remote
Comes with the Java 2 Platform
<<Interface>>
javax.ejb.EJBHome
Comes with EJB distribution
<<Interface>>
Home Interface
Supplied by Bean provider (we write this)
Home Object
Generated for us by container & vendor tools

Home Interface

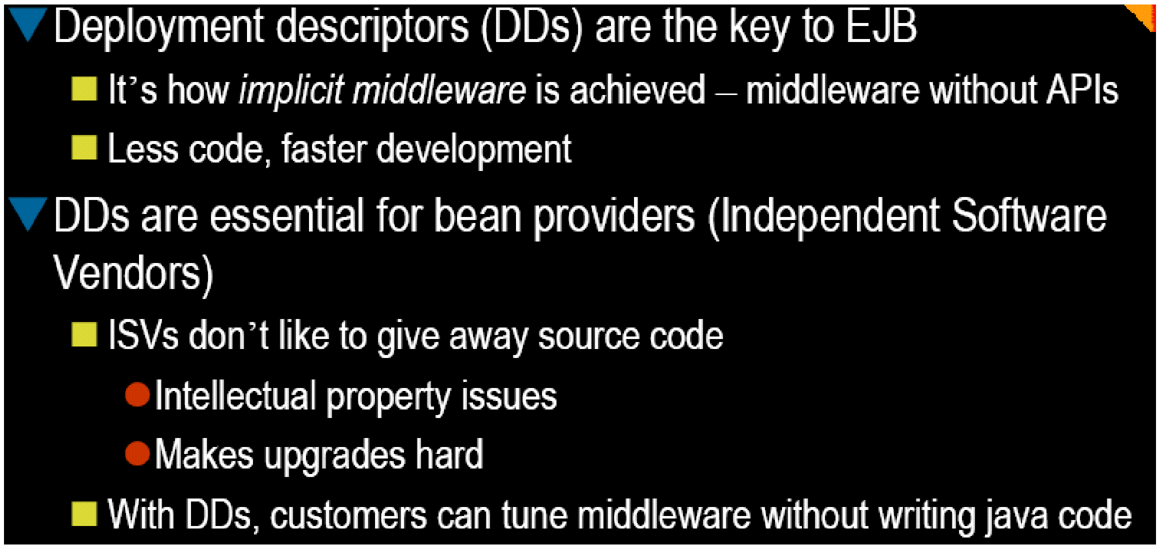


BIG picture

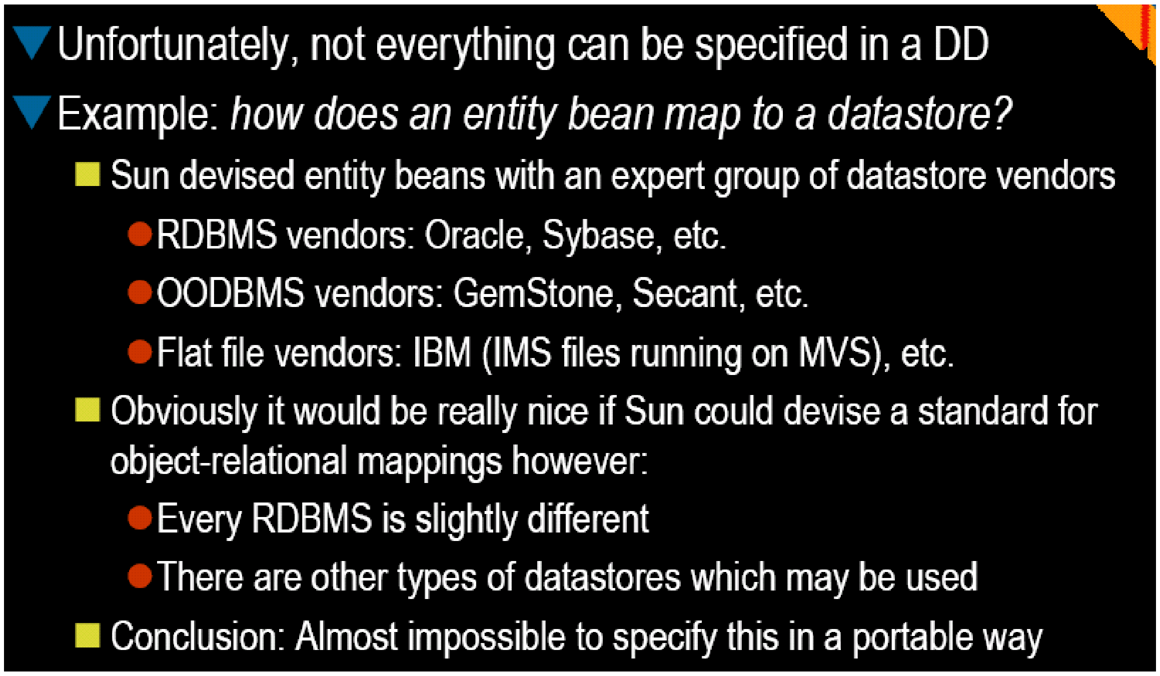


- A deployment descriptor (DD) is a contract between you and the application server
  - A DD dictates the middleware generated by app server
  - Enable you to 'set middleware properties' on your components
  - Idea of *declarative middleware* – I declare what I need, rather than write code
  - What goes into a DD? Plenty of things that you can tune:
    - Transactions
    - Security
    - Relationships between data objects

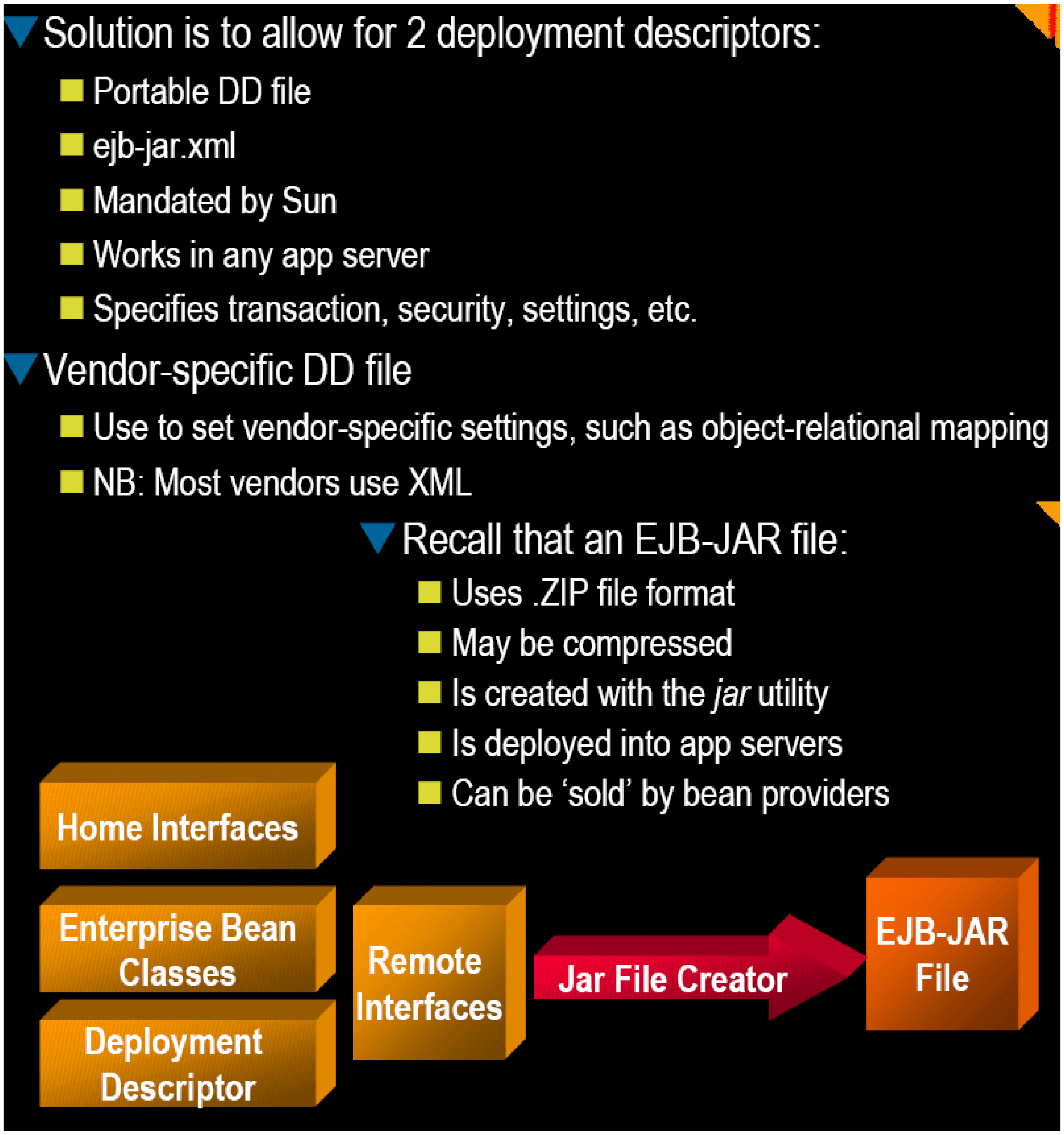
Code is not changed, but behavior changed!



- Deployment descriptors (DDs) are the key to EJB
  - It's how *implicit middleware* is achieved – middleware without APIs
  - Less code, faster development
- DDs are essential for bean providers (Independent Software Vendors)
  - ISVs don't like to give away source code
    - Intellectual property issues
    - Makes upgrades hard
  - With DDs, customers can tune middleware without writing java code

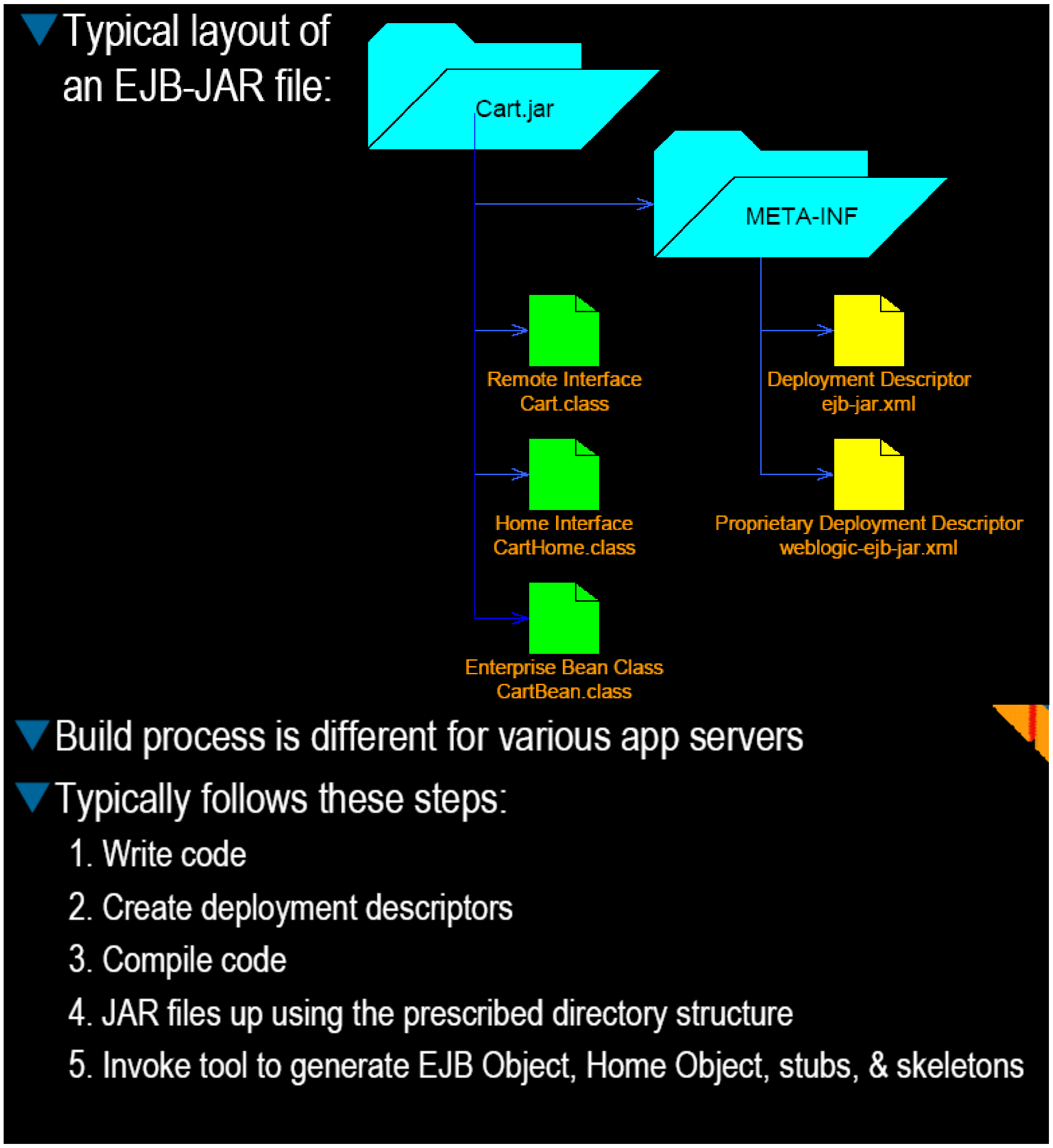


- Unfortunately, not everything can be specified in a DD
- Example: *how does an entity bean map to a datastore?*
  - Sun devised entity beans with an expert group of datastore vendors
    - RDBMS vendors: Oracle, Sybase, etc.
    - OODBMS vendors: GemStone, Secant, etc.
    - Flat file vendors: IBM (IMS files running on MVS), etc.
  - Obviously it would be really nice if Sun could devise a standard for object-relational mappings however:
    - Every RDBMS is slightly different
    - There are other types of datastores which may be used
  - Conclusion: Almost impossible to specify this in a portable way

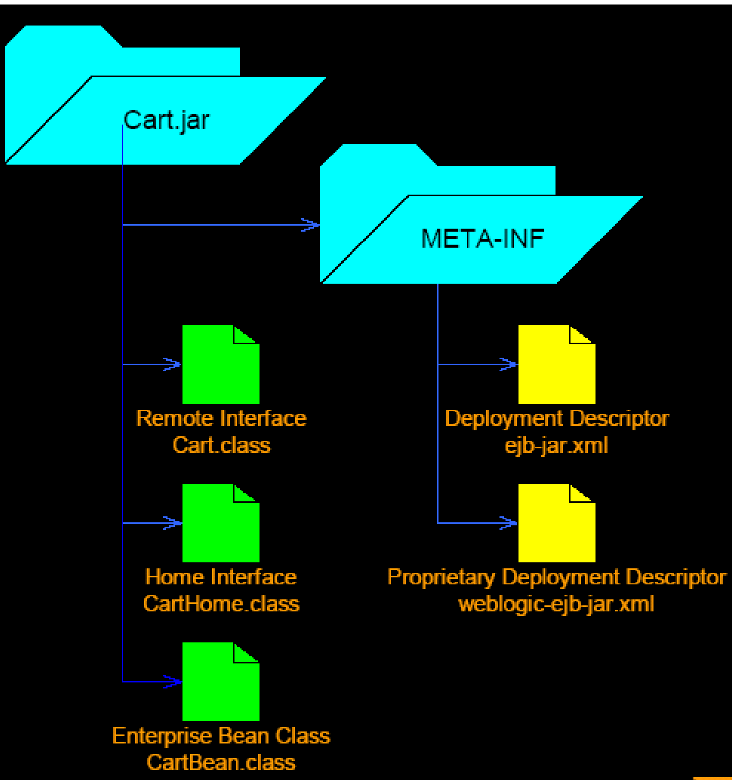


- Solution is to allow for 2 deployment descriptors:
  - Portable DD file
  - ejb-jar.xml
  - Mandated by Sun
  - Works in any app server
  - Specifies transaction, security, settings, etc.
- Vendor-specific DD file
  - Use to set vendor-specific settings, such as object-relational mapping
  - NB: Most vendors use XML
- Recall that an EJB-JAR file:
  - Uses .ZIP file format
  - May be compressed
  - Is created with the *jar* utility
  - Is deployed into app servers
  - Can be ‘sold’ by bean providers

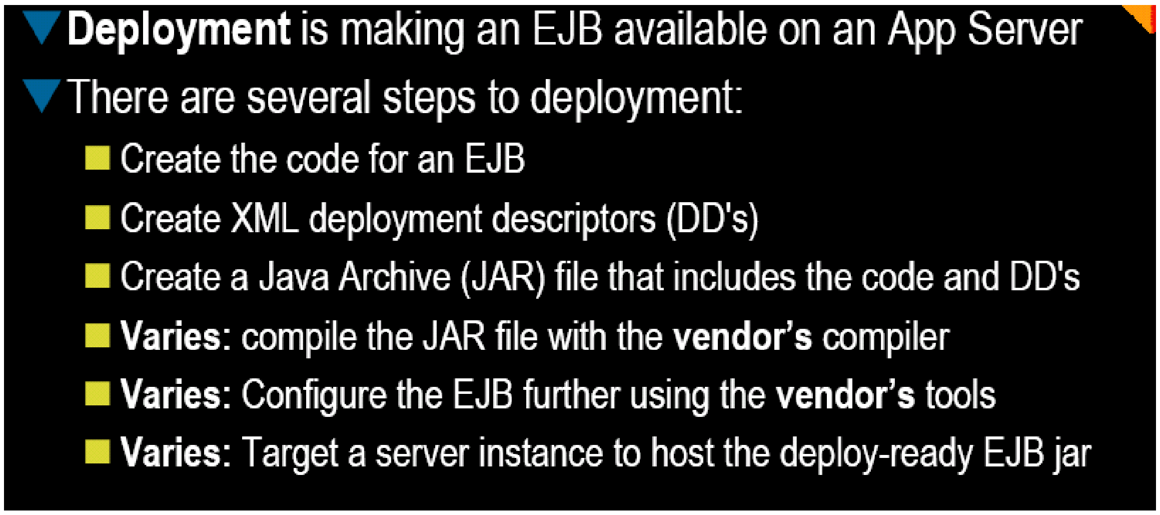
Stubs will be generated



- Typical layout of an EJB-JAR file:



- Build process is different for various app servers
- Typically follows these steps:
  1. Write code
  2. Create deployment descriptors
  3. Compile code
  4. JAR files up using the prescribed directory structure
  5. Invoke tool to generate EJB Object, Home Object, stubs, & skeletons



- **Deployment** is making an EJB available on an App Server
- There are several steps to deployment:
  - Create the code for an EJB
  - Create XML deployment descriptors (DD's)
  - Create a Java Archive (JAR) file that includes the code and DD's
  - **Varies:** compile the JAR file with the **vendor's** compiler
  - **Varies:** Configure the EJB further using the **vendor's** tools
  - **Varies:** Target a server instance to host the deploy-ready EJB jar

EJB roles

The EJB specification defines six roles for EJB development and deployment.

Bean provider

The bean provider is the **developer** of the code, and is responsible for converting the business requirements into actual physical code. He or she must provide the necessary **classes** that constitute an EJB, as well as providing the deployment file describing the runtime settings for the bean. **The bean provider's final product is an EJB JAR file.**

Application assembler

This may be the same as the bean provider, or a senior team lead. The application assembler's responsibility is to **package all the different EJB components,** along with any other application components, the **final product being an enterprise application EAR file.**

Deployer

This person is responsible for deploying the EJB application on the target runtime environment. The deployer should be familiar with all aspects of the environment, including transaction and security support.

Server provider

The job of the server provider is to provide a runtime EJB server environment that is compliant with the EJB specification. For example, IBM is a server provider of the WebSphere Application Server.

Container provider

The job of the container provider is to provide a runtime EJB container environment that is compliant with the EJB specification, **and is generally the same as the server provider.** The server and container cooperate to provide the quality-of-service runtime environment for the EJBs.

Systems administrator

The system administrator's task is to ensure that the runtime environment is configured in such a way that the application can function correctly, and is integrated with all of the required external components.

Main EJB framework components

There are six main framework components of EJB technology (Figure 1):

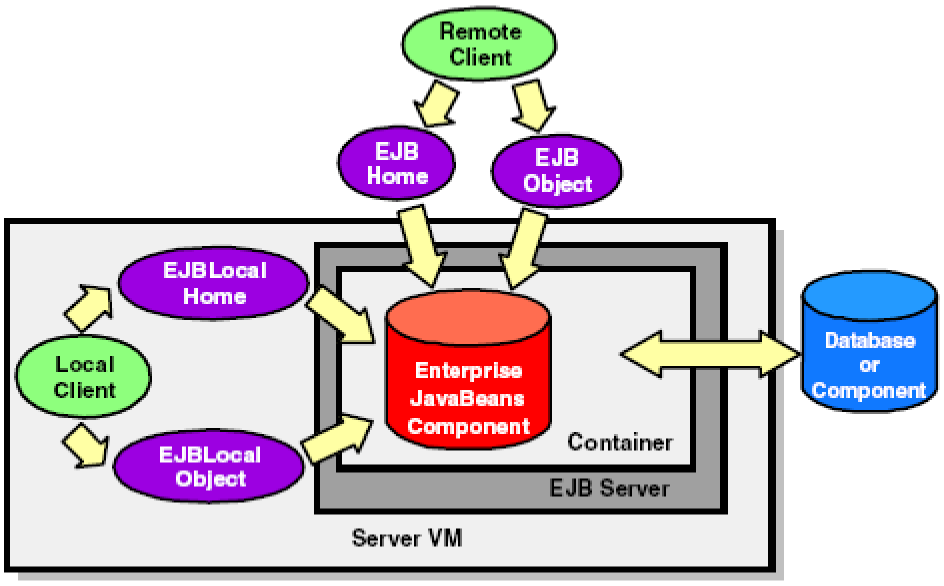
EJB server—Provides the actual primary services to all EJBs. An EJB server may host one or more EJB containers. It is also called generically an Enterprise Java Server (EJS). WebSphere Application Server is an EJS.
EJB container—Provides the runtime environment for the enterprise bean instances, and is the intermediary between the EJB component and the server.

EJB component—These represent the **actual EJBs** themselves. There are three types of enterprise beans: **entity, session, and message-driven beans.**

EJB interfaces and EJB bean—The **interfaces for client access (EJB home and EJB object)** and the EJB bean class.

EJB deployment descriptor—The descriptor defines the runtime quality of service settings for the bean when it is deployed. Many of the settings of an EJB, such as the transactional settings, are defined in the deployment descriptor. The deployment descriptor is an external entity from the bean, and therefore **decouples the bean component itself from the runtime characteristics.**

EJB client—A client that accesses EJBs.



EJB container

The EJB container is a system that functions as **a runtime environment for enterprise beans** by managing and applying the primary services that are needed for bean management at runtime. In addition to being an intermediary to the seven services above provided by the EJB server, the EJB container will also provide for EJB instance life-cycle management, and EJB instance identification. **EJB containers create bean instances, manage pools of instances, and destroy them.** The container provides the service level specified at deployment time in the deployment descriptor. **For example, it will start a transaction or check for security policies.** All the services are well defined within the EJB framework. The framework is implemented through the bean callback methods. These methods are purely for system management purposes and they are called only by the container when it is interacting with the deployed beans, and are invoked when the appropriate life-cycle event occurs. The specification is clear on what actions trigger what events.

Remote accessibility

Remote accessibility enables remote invocation of a native component by converting it into a network component. EJB containers use the Java RMI interfaces to specify remote accessibility to clients of the EJBs.

Primary services

The responsibilities that an EJB container must satisfy can be defined in terms of the primary services. Specific EJB container responsibilities are as follows:

Naming—**A client can invoke an enterprise bean by looking up the name of the bean in a centralized naming space called a naming service, and this is accessed via the Java Naming and Directory Interface (JNDI) API.** The container is responsible for registering the (**unique**) lookup name in the JNDI namespace when the server starts up, and binding the appropriate object type into the JNDI namespace.

Transaction—**A transaction is defined as a set of tasks that must execute together;** either all must work or all must be undone.

The EJBs transaction behavior is described in the deployment descriptor, and this is also referred to as **container-managed transactions (CMT)**. Because the container manages the transactions, applications can be written without explicit transaction demarcation.

Security—The container is responsible for enforcing the security policies defined at deployment time whenever there is a method call, through access control lists (ACL). An **ACL is a list of users, the groups they belong to, and their rights,** and it ensures that users access only those resources and perform only those tasks for which they have been given permission.

Persistence—The container is also responsible for managing the persistence of a bean (all storage and retrieval of data) by synchronizing the state of the bean instance in memory with the respective record in the data source. Concurrent access to the data from multiple clients is managed through the concurrency and transaction services (entity beans only).

Concurrency—Concurrency is defined as access by two or more clients to the same bean, and the container manages concurrency according to the rules of the bean type. Additionally, for message-driven beans, concurrency is defined as managing the processing of the same message.

Life cycle—The container is responsible for controlling the life cycle of the deployed components. **As EJB clients start giving requests to the container, the container dynamically instantiates, destroys, and reuses the beans as appropriate.** The specific life-cycle management that the container performs is dependent upon the type of bean.

Bean state-management is a type of life-cycle management, where activating and passivating of beans based on usage can be achieved. These techniques become part of the bean life cycle that the container must manage.

Messaging—The container must provide for asynchronous messaging. Specifically, it must provide for the reliable routing of messages from JMS clients to message-driven beans.

EJB component (the actual EJB)

New EJB 2.0 - messaging service: Messaging is new in the EJB 2.0 specification. It essentially requires container providers to provide two things:

Support of message-driven EJBs
Support for internal messaging service

To be compliant with EJB 2.0, WebSphere Application Server Version 5 now provides both of these. The internally embedded JMS messaging that is provided with the application server is actually a lightweight implementation of IBM WebSphere MQ. Given that the messaging service is intended to only provide an internal container-level communication bus, it is not required to be a fully compliant messaging middleware implementation.

EJB types

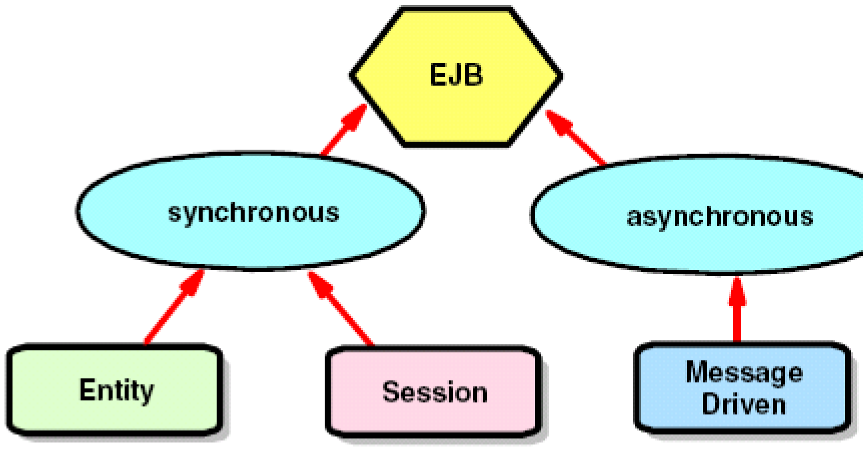
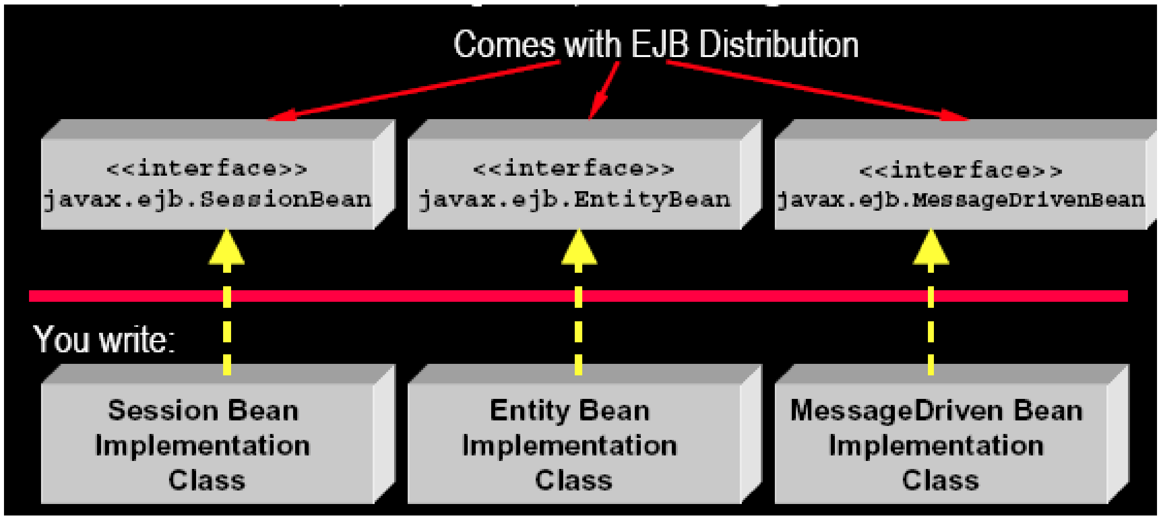


Figure 2:

Entity beans—Entity beans are modeled to represent business or domain-specific concepts, and are typically the nouns of your system, such as customers and accounts. They usually represent data (entities) stored in a database.

Session beans—A session bean is modeled to represent a task or workflow of a system, and provide coordination of those activities between beans, such as a banking service, allowing for a transfer between accounts.

Message-driven beans—Like session beans, message-driven beans (MDB) may also be modeled to represent tasks. However, they are invoked by the receipt of asynchronous messages. The bean either listens for or subscribes to messages that it is to receive.


Comes with EJB Distribution
<<interface>> javax.ejb.SessionBean
<<interface>> javax.ejb.EntityBean
<<interface>> javax.ejb.MessageDrivenBean
You write:
Session Bean Implementation Class
Entity Bean Implementation Class
MessageDriven Bean Implementation Class

Synchronous versus asynchronous invocation
Entity and session beans are accessed synchronously through a remote or local EJB interface method invocation. This is referred to as synchronous invocation, because there is a request, and a (blocking) wait for the return. Clients of EJBs invoke methods on session and entity beans. An EJB client may be an external construct like a servlet (remote) or another EJB within the same JVM (local). The message-driven bean is not accessible through a remote or local interface. The only way for an EJB client to communicate with a message-driven bean is by sending a JMS message. This is an example of asynchronous communication. The client does not invoke the method on the bean directly, but rather, uses JMS constructs to send a message. The container delegates the message to a suitable message-driven bean instance to handle the invocation.

EJB interfaces and EJB bean

An EJB component consists of the following primary elements, depending on the type of bean:

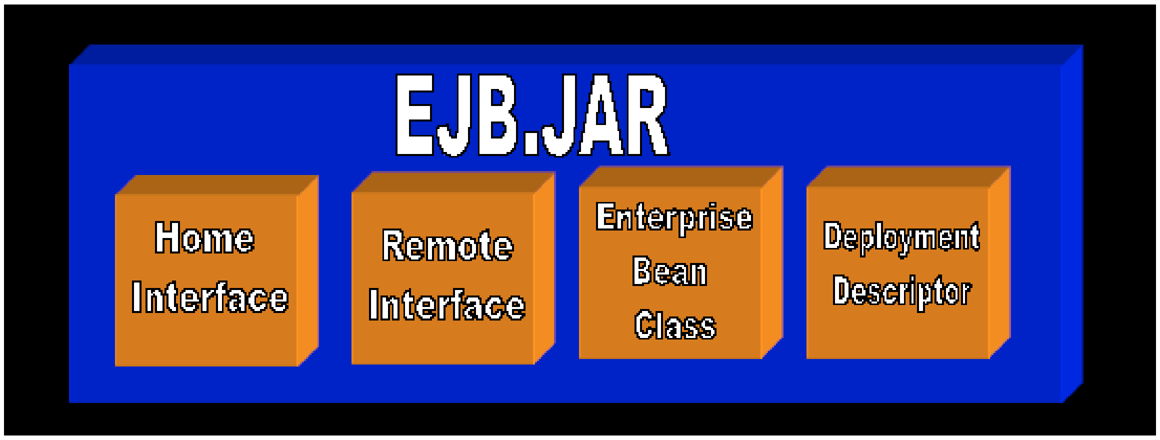
EJB component interface—Is used by an EJB client to gain access to the capabilities of the bean. This is where the business methods are defined. **The component interface is called the EJB object**.

EJB home interface—is used by an EJB client to gain access to the bean. Contains the bean life-cycle methods of create, find, or remove. The home interface is called the **EJB home.**

EJB bean class—Contains all of the actual bean business logic. Is the class that provides the business logic implementation. Methods in this bean class associate to methods in the component and home interfaces.

Entity beans also have a **primary key class**, which represents a unique entity. A bean developer often develops the classes and interfaces in the above order.

New EJB 2.0 - component interface: In EJB 1.x, the component interface was called the remote interface. This is because prior versions of EJBs supported only the notion of remote client accessibility. New in EJB 2.0 is the idea of local accessibility of enterprise beans. There are two types of component interfaces, local or remote.



zip type file that can be created using jar command

EJB component interface: EJB object

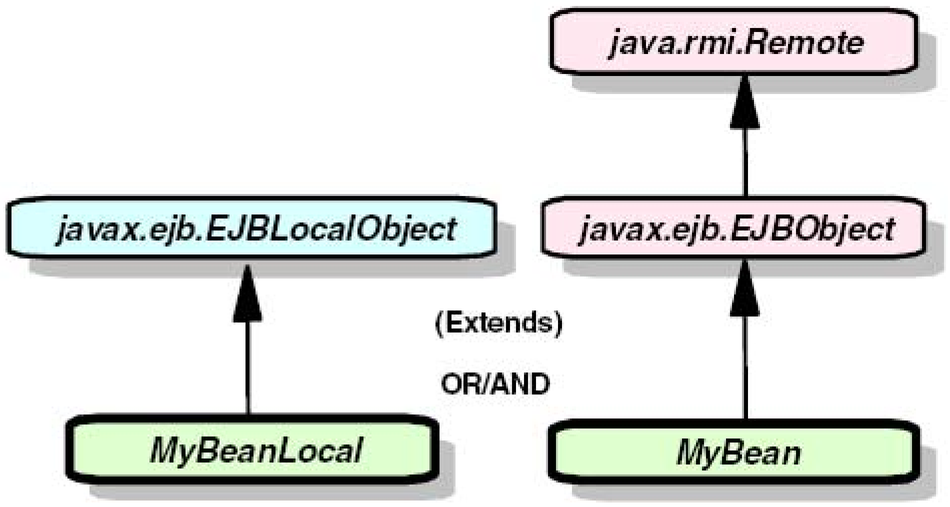
A component interface is used by the client of the enterprise bean to gain access to the capabilities of the bean. It defines the business methods that are visible to the client. T**he business methods of the component interface must have corresponding implementations in the bean class,** although the b**ean class never implements the component interface directly**.

A message-driven bean does not have a component interface, since it is not invoked directly by a client.

EJBs are distributed objects. **An EJB client never invokes an instance of an enterprise bean class directly. A method invocation is intercepted by the EJB container that provides the services to delegate the method execution to the appropriate enterprise bean instance at the right time.** An enterprise bean is therefore a passive component managed by the EJB container.

The concept of the method interceptor simplifies the component development: an EJB developer can concentrate on writing business logic. **The artifact that enables the decoupling of the EJB client with the enterprise bean class is called the EJBObject.** EJBObjects are part of the EJB container and are generated from the container tools during deployment. The EJBObject is the actual object that implements the component interface.

There are two types of component interfaces: a remote component (EJBObject) and a local component (EJBLocalObject) interface. A particular user-defined component interface may only implement one type, either local or remote,



although it is possible to have one of each type per EJB.

Local versus remote interfaces

Remote interfaces have always existed in EJBs, and provide the conventions for accessing distributed objects that are used by EJB clients that are outside of the container or JVM. **In remote invocation, method arguments and return values are essentially passed via pass-by-value, where a complete copy of the object is made and serialized before being sent over the network to the remote service.**

Both the object serialization and network overhead can be a costly proposition, ultimately reducing the response time for the request. However, with remotei nterfaces, location independence can be achieved because the same call can occur whether this client is inside or outside of the container.

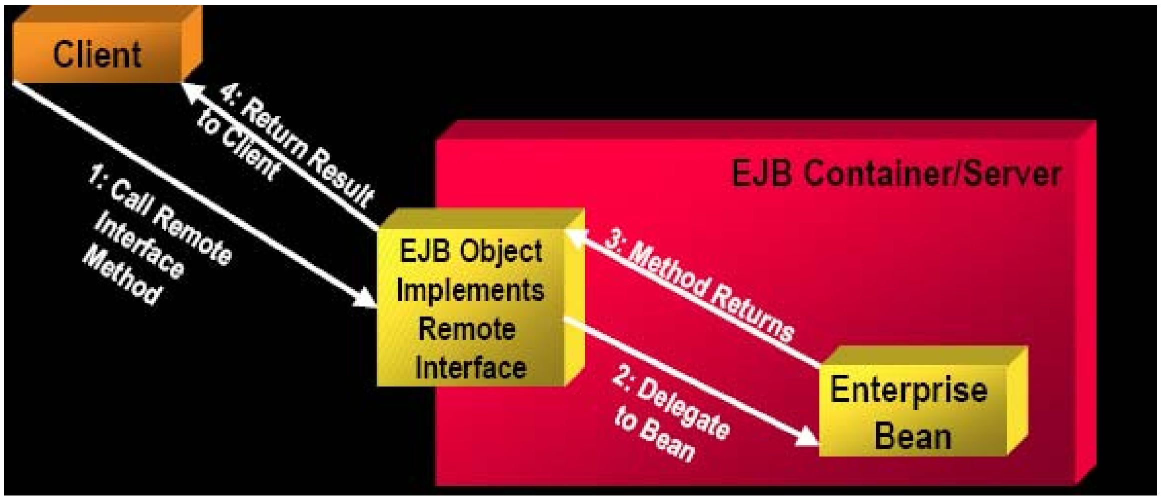
Local interfaces, which are new to EJB 2.0, provide a way for beans inside the same JVM to interact with each other locally. In local invocation, method arguments are passed by reference, and the execution is done within the same JVM, so **no serialization or network overhead is assessed. However, with local interfaces, there is now location dependence, because the type of interface used will only work from clients that execute within the same JVM.**

EJBLocalObject or EJBObject Remote interface methods must throw a RemoteException, and optionally any application exceptions. **Local interface methods may only throw optional application exceptions. Both can throw an EJB exception (runtime exception).**

EJB home interface: EJB home

As EJBs are distributed objects, a factory service is used to create and find bean instances. The home interface provides this service, and is used by the EJB client to gain access to the bean. It defines the bean's life-cycle methods, and it provides for the basic life-cycle management capabilities of the bean, such as create, remove, and find. A message-driven bean does not have a home interface, since it is not invoked directly by a client.

A client never actually has direct access to a bean class instance. What the client of the bean actually has is an EJBObject, which is the interceptor to the bean instance itself.


Client
4: Return Result to Client
1: Call Remote Interface Method
EJB Container/Server
EJB Object Implements Remote Interface
3: Method Returns
2: Delegate to Bean
Enterprise Bean

This EJBObject is the object that actually implements our component interface, which contains our beans business methods. The EjbHome object is an object that implements the home interface. As in EJBObject, it is generated from the container tools during deployment, and includes container-specific code. At startup time, the EJB container instantiates the EJBHome objects of the deployed enterprise beans and registers the home in the naming service. An EJB client accesses the EJBHome objects using JNDI (Java Naming and Directory Interface).
the EJB.

Additional flavors of EJB types

Although there are three explicit types of beans, there are two different flavors of both entity and session beans:

Flavors of an entity bean—Entity beans represent persistent data. There are two types of entity beans that govern how the persistency behavior will be managed: by the container, through container-managed persistence (CMP), or by the bean itself, through bean-managed persistence (BMP).

Flavors of a session bean—Additionally, there are two types of session beans. Stateless session beans are beans that maintain no state, and are pooled by the container to be reused. Stateful session beans are beans that represent a conversational state with a client, where the state is maintained between client invocations.

Although the classifications above (CMP versus BMP, and stateful versus stateless) are often referred to as types of EJBs, they are not really different types in the sense that there are no new classes or interfaces to represent these types (they are still just session and entity beans).