



Java Remote Method Invocation

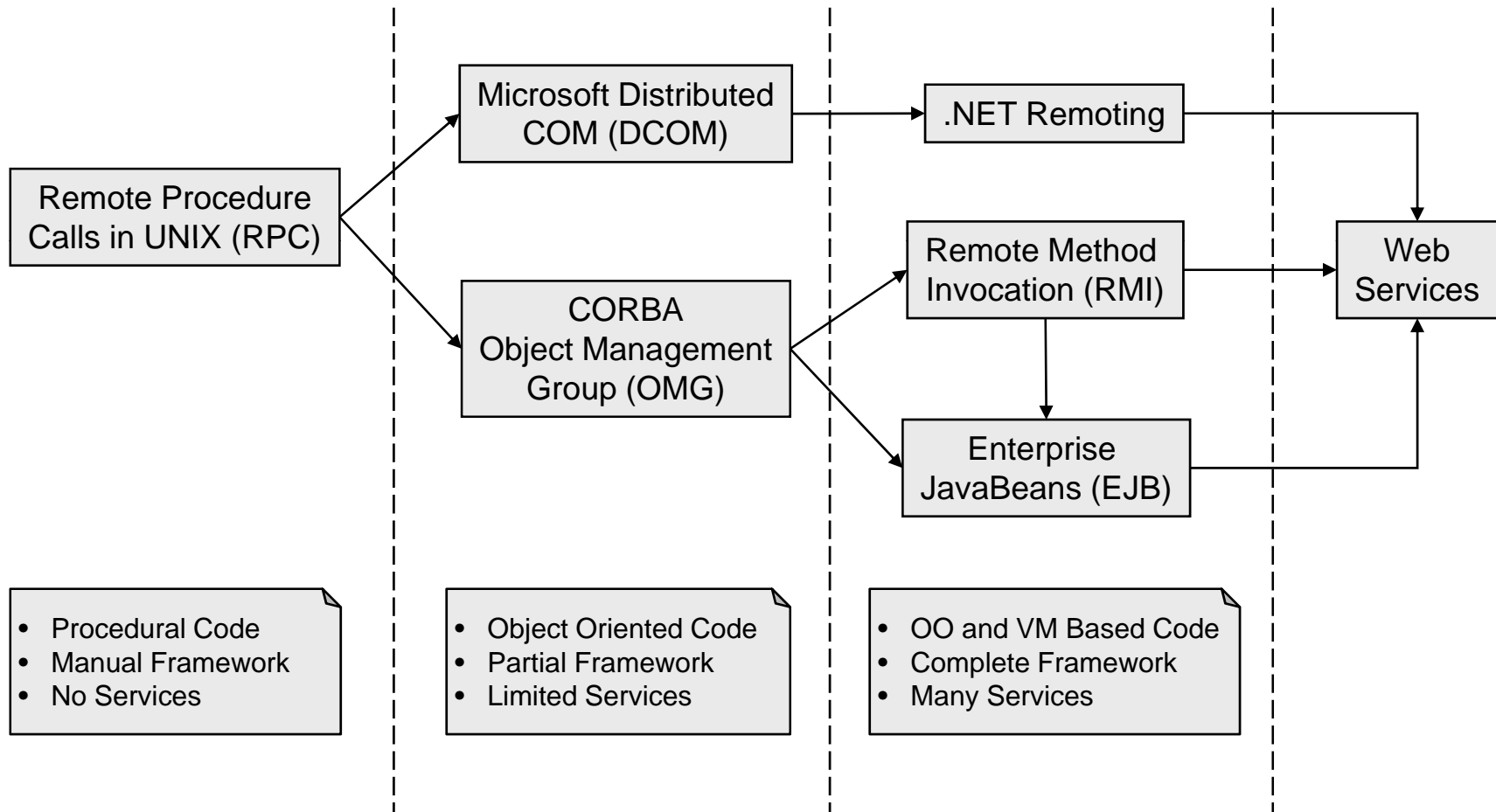
Using Remote References



Introducing Java RMI

- Java apps frequently need to talk to remote systems
 - A Java GUI might need to call business logic on a server
 - One server side system might need functionality in another
- Remoting is the general term for distributed method calls
 - It has replaced the older term 'Remote Procedure Calls' (RPC)
- An object in VM1 has a 'remote reference' to one in VM2
 - In theory the two objects can pretend they are in the same VM
 - In practice remote method calls must be used with caution
- Method calls are made over the network via 'marshalling'
 - The call is transformed into a message that can be transmitted
 - The content of the message may be binary, XML or a mixture

The Evolution of Remote Methods

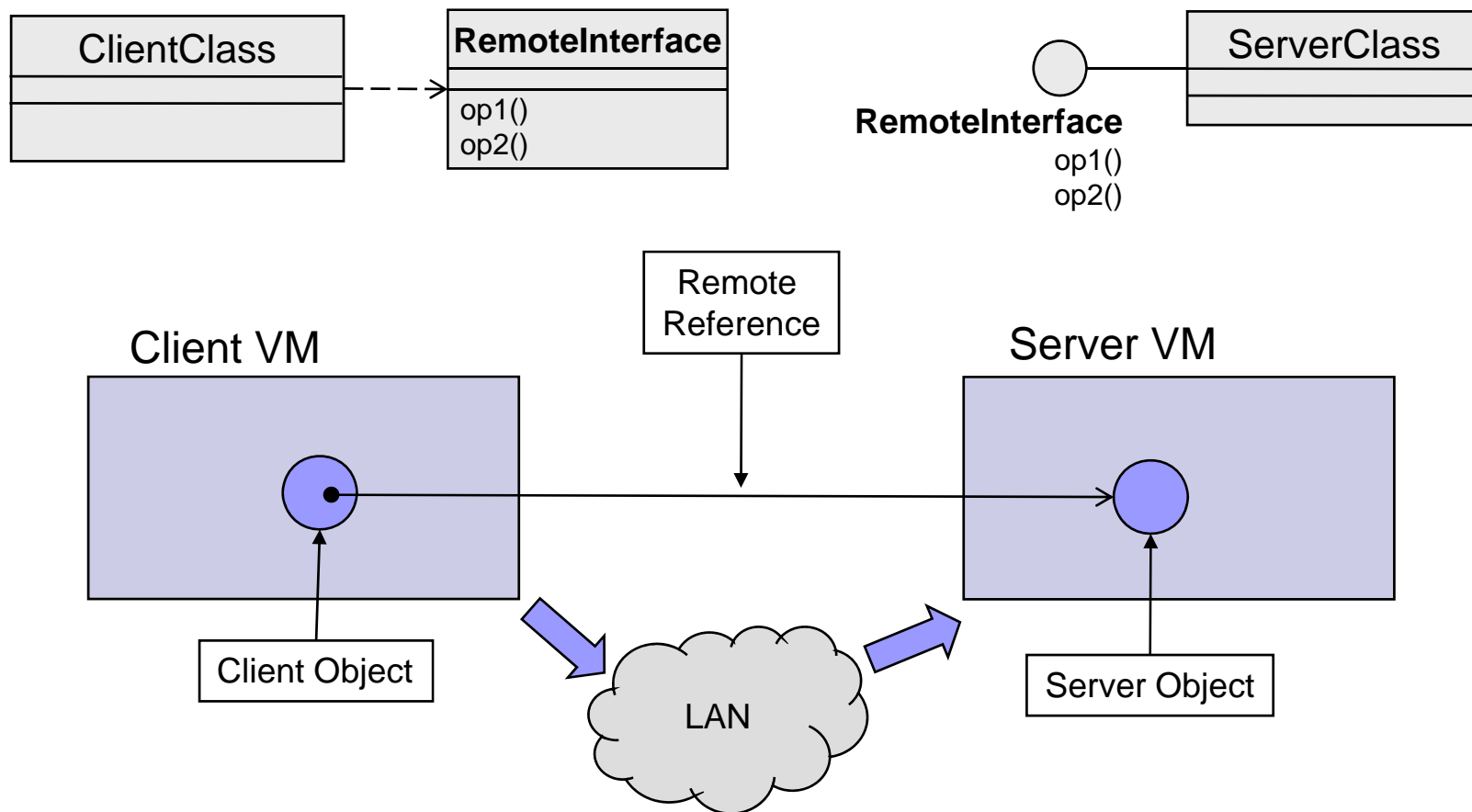




The Logical View of Java Remoting

- Java VM's support remote references
 - A reference in one VM which points to an object in another
 - Clients can use the reference as normal
 - Obviously much more work is being done by the VM
- A remote reference must have an interface type
 - The client uses an interface implemented by the remote object
 - The client should never know the class type of the remote object
 - This is good practice when building distributed systems
- Garbage collection works with remote references
 - One VM notifies another as references are made and broken
 - VM's use a leasing system to cope with server crashes

The Logical View of Java Remoting

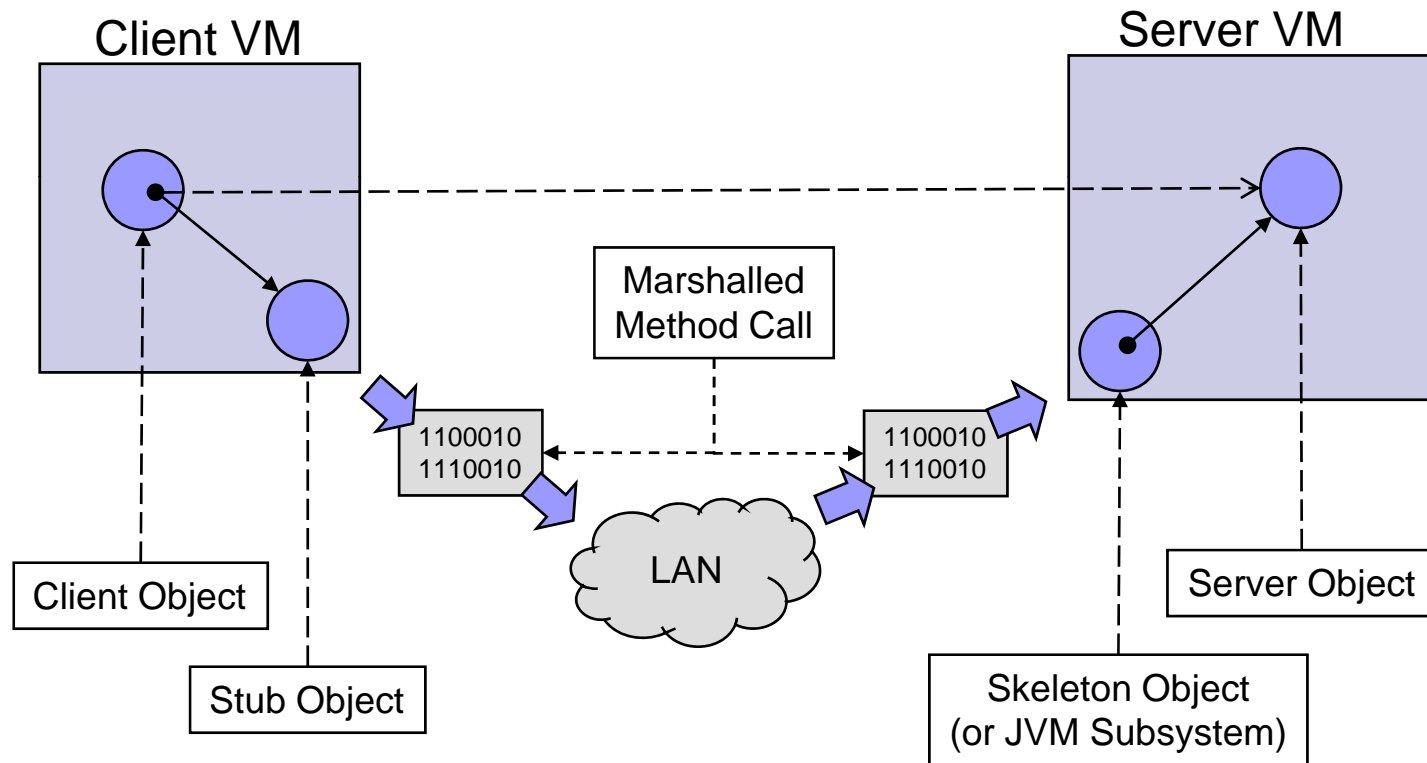




The Architecture of Remoting

- A 'remote reference' is a high level concept
 - It cannot be implemented simply via memory addresses
- Instead a supporting infrastructure is required:
 - The client object actually references a 'stub'
 - When a call is made the stub marshals it into a data structure that can be sent over the network to the server
 - On the server a 'skeleton' receives and un-marshals the message
 - A local call is then made by the skeleton on the server object
 - Finally the procedure is reversed to carry back the return value

The Architecture of Remoting





The Architecture of Remoting

- The helper classes are built by the RMI compiler
 - Running 'rmic' on the class file for a server side object causes stub and (optionally) skeleton classes to be generated
 - As of Java 2 only stub classes are required
- The final piece of architecture is the naming system
 - The means by which a client obtains the first remote reference
 - After which clients typically use factory methods to get others
- Several possible naming systems are available
 - RMI provides a simple service call the 'RMI Registry'
 - JEE developers can make use of JNDI



Remote References

- Interfaces used for remote references are special
 - They must extend the marker interface 'java.rmi.Remote'
 - All methods must throw 'java.rmi.RemoteException'
 - However good your code network faults are always possible
 - When overriding these methods don't declare this exception
 - Unless you actually need to throw a 'RemoteException' yourself
 - NB This is a Java language feature and not RMI specific
- These interfaces should be designed with care
 - Remote methods should always be idempotent
 - Consider round-tripping, network loading and security
 - Beware the '10 feet of cable' and 'Slashdot effect' problems



Working With Remote Interfaces

```
public interface PizzaShop extends Remote {  
    public void listOrders() throws RemoteException;  
    public void placeOrder(String order) throws RemoteException;  
}
```

```
public static void main(String[] args) throws Exception {  
    //lookup the remote object  
    String host = (args.length < 1) ? "localhost" : args[0];  
    String url = "rmi://" + host + ":1099/PizzaShop";  
    PizzaShop shop = (PizzaShop)Naming.lookup(url);  
  
    //use the remote object  
    shop.placeOrder("pepperoni");  
    shop.placeOrder("napoletana");  
    shop.placeOrder("roma");  
    shop.placeOrder("quattro stagioni");  
    shop.placeOrder("isabella");  
    shop.listOrders();  
}
```



Creating Remote Objects

- For an object to be called remotely it must:
 - Implement a remote interface
 - Any number of interfaces can be implemented
 - Be registered with the RMI Runtime
 - The part of the VM which manages remote references
- Registering with the RMI Runtime is straightforward
 - Just inherit from 'java.rmi.server.UnicastRemoteObject'
 - The constructor of this class registers your object for you
 - Alternatives are available for advanced development
 - Inherit from 'RemoteServer' or 'RemoteObject'
 - Inherit from an arbitrary base class and use the 'UnicastRemoteObject.exportObject' method



Creating Remote Objects

```
public class PizzaShopImpl extends UnicastRemoteObject implements PizzaShop {
    public PizzaShopImpl() throws RemoteException {
        //export happens here
        super();
    }
    public void listOrders() {
        Iterator i = list.iterator();
        System.out.println("---- Current orders are: ----");
        while(i.hasNext()) {
            System.out.print("\t");
            System.out.println((String)i.next());
        }
    }
    public void placeOrder(String order) {
        list.add(order);
    }
    private ArrayList list = new ArrayList();
}
```



Publishing Remote Objects

```
public class LaunchServer {
    public static void main(String[] args) {
        try {
            //start the rmi registry
            Registry r = LocateRegistry.createRegistry(1099);

            //bind the object in registry with name "PizzaShop"
            r.bind("PizzaShop",new PizzaShopImpl());

            System.out.println("[RegisterPizzaShop::main] Registered Pizza Shop");
        } catch(Exception e) {
            System.out.println("[RegisterPizzaShop::main] Cant register Pizza Shop - " + e);
            e.printStackTrace();
        }
    }
}
```



Remote Method Calls

- Special considerations apply to calling remote methods
 - Normal rules for parameters and exceptions don't apply
- Parameters can be passed in two ways
 - If a parameter is 'Serializable' it is passed by value
 - Multiple value parameters share the same serialization record
 - This prevents shared objects being duplicated
 - If a parameter is 'Remote' it is passed by reference
 - A remote reference to the parameter is sent to the server
 - The distinction between client and server becomes arbitrary
- If these rules are not followed the call will fail
 - This is one of the most common problems in EJB development



Remote Method Calls

- Remote objects never throw local exceptions
 - If a method throws an exception it is wrapped in a 'RemoteException' and sent to the server
 - Remote exceptions may be thrown
 - As a result of an error on the network (client or server)
 - As a result of an error in your code
 - As a result of a problem with the RMI Runtime
- Remote objects are very robust
 - All the error handling is delegated to the client
 - Servers are usually more important than clients
 - Remote objects should be designed to be fault tolerant



Enterprise JavaBeans

History and Evolution



Introducing Enterprise JavaBeans

- EJB is an area to be approached cautiously
 - It is by far the hardest part of Enterprise Java
 - Generations of Java developers grew to hate it
- EJB provides a distributed component architecture
 - Objects on one VM talk to objects on another ‘seamlessly’
 - Applications can be clustered over any number of servers
- This type of architecture has proved notoriously hard to simplify for use by mainstream developers
 - Earlier attempts like CORBA and DCOM were heavily promoted but were successful only in tightly controlled environments



Introducing Enterprise JavaBeans

- RMI by itself is not enough to build distributed systems:
 - Developers must manually create all the server side objects
 - Or else publish object factories that will be called by clients
 - The issue of concurrent access is not addressed
 - How will multiple calls against server side objects be handled?
 - Transactions, security etc... should be available 'out of the box'
- The EJB specification tries to address these problems:
 - Scalability and concurrency are handled automatically
 - Developers don't need low-level API's to control transactions
 - The full lifecycle of the bean is managed by the container
 - There is backward compatibility with CORBA, forward compatibility with Web Services and provision for local access



Comparing EJB's and Servlets

- EJB's and Servlets perform very different roles
 - Servlets are lightweight and used to maintain the web interface
 - Their lifecycle is (relatively) easy to manage
 - They should not contain business logic
 - EJB's are heavyweight business components
 - They have to support complex business logic
 - Such as transactions across multiple calls to JDBC, JMS etc...
 - Their lifecycle is complex to manage
- EJB's and Servlets share one vital design principal:
 - Access to the component is always mediated and never direct
 - This applies to both clients on other VM's and peer components



Introducing Enterprise JavaBeans

- The EJB spec has had a complex history
 - New types of bean were introduced and existing ones altered
- There were four key problems with the original version:
 - It was assumed that access would always be via RMI
 - Even when components were beside one another in memory
 - The model for object persistence was too complex
 - With too much emphasis placed on the provision of tools by vendors
 - No thought was given to asynchronous messaging
 - The number of steps required to create a bean was onerous
 - Two interfaces, one class and an XML file to get to 'Hello World'
 - Plus additional (vendor specific) configuration on the server



Developing Enterprise JavaBeans

- We will be covering two approaches to writing EJB's:
 - The 'classic' approach that has always been used
 - The new approach introduced in EJB 3 / JEE 5
- Note that the underlying model remains the same
 - Although the ways beans are written has been radically simplified the same underlying concepts remain
 - An exception is the replacement of Entity EJB's with the JPA
- It is likely you will need to know both approaches
 - Especially when converting old applications to EJB 3
 - It remains to be seen if JEE5 will reawaken interest in EJB's



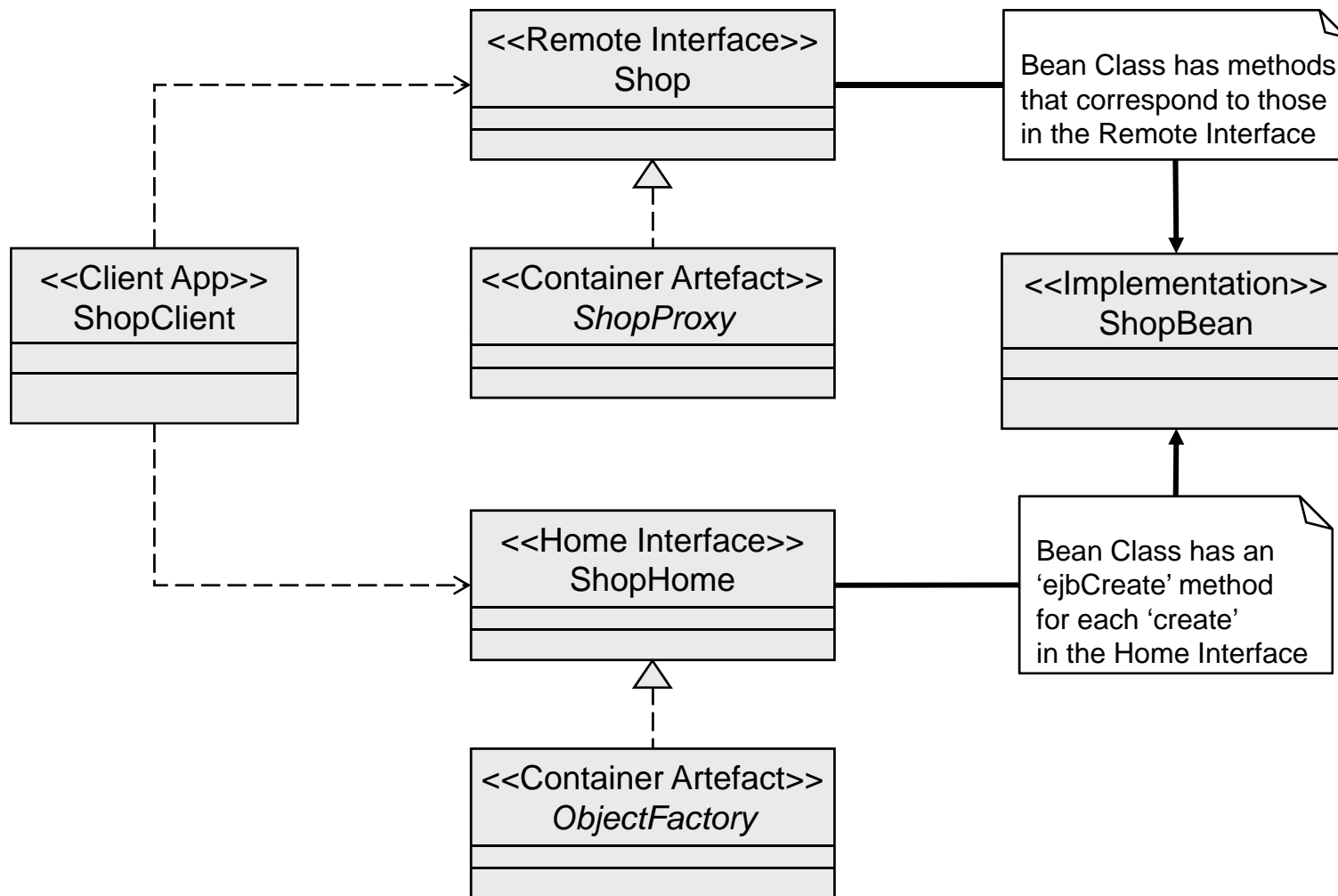
Classic EJB's Part I

An Example Bean



Components of an EJB

- The next slides show components of an example EJB:
 - The Remote Interface
 - Used to generate a proxy class
 - Confusingly called the 'EJB Object'
 - The Home Interface
 - Used to generate an object factory
 - How this works depends on the bean type
 - The Bean Implementation
 - Which doesn't implement the home or remote interfaces
 - The Deployment Descriptor
 - The WebLogic Deployment Descriptor






```
public interface Shop extends EJBObject {  
    boolean isInStock(String productCode) throws RemoteException;  
    int quantityInStock(String productCode) throws RemoteException;  
    double getPrice(String productCode) throws RemoteException;  
    boolean makePurchase(String buyerID, String productCode,  
        int quantity, double price) throws RemoteException;  
}
```

Remote
Interface

```
public interface ShopHome extends EJBHome {  
    Shop create() throws RemoteException, CreateException;  
}
```

Home
Interface



Bean Implementation

```
public class ShopBean implements SessionBean {
    public void ejbCreate() {}
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void setSessionContext(SessionContext ctx) {}
    public boolean isInStock(String productCode) {
        return true;
    }
    public int quantityInStock(String productCode) {
        return 20;
    }
    public double getPrice(String productCode) {
        return 9.50;
    }
    public boolean makePurchase(String buyerID, String productCode, int quantity, double price) {
        return true;
    }
}
```



Standard Deployment Descriptor

```
<!-- XML and DTD declarations omitted -->
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>Shop</ejb-name>
      <home>demos.ejb.shop.ShopHome</home>
      <remote>demos.ejb.shop.Shop</remote>
      <ejb-class>demos.ejb.shop.ShopBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

WebLogic Deployment Descriptor

```
<!-- XML and DTD declarations omitted -->
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>Shop</ejb-name>
    <jndi-name>ShopEJB</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```



Client Application Part One

```
public class ShopClient {
    public static void main(String[] args) throws Exception {
        Shop shop = getRemoteObjectRef();

        if(shop.isInStock("TEST")) {
            System.out.println("Item in stock");
        } else {
            System.out.println("Item not in stock");
        }

        int quantity = shop.quantityInStock("TEST");
        System.out.println("There are: " + quantity + " in stock");


        double price = shop.getPrice("TEST");
        System.out.println("The product cost is: " + price);

        if(shop.makePurchase("TEST","TEST",20,9.5)) {
            System.out.println("Order succeeded");
        } else {
            System.out.println("Order failed");
        }
    }
}
```



Client Application Part Two

```
private static Shop getRemoteObjectRef()  
    throws RemoteException, NamingException, CreateException {  
    Properties p = new Properties();  
    p.put(Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.WLInitialContextFactory");  
    p.put(Context.PROVIDER_URL, "t3://127.0.0.1:7001");  
  
    InitialContext ic = new InitialContext(p);  
    Object obj = ic.lookup("ShopEJB");  
    ShopHome home = (ShopHome)PortableRemoteObject.narrow(obj,ShopHome.class);  
    return home.create();  
}  
}
```



JUnit Test

```
public class ShopBeanTest extends TestCase {
    protected void setUp() throws Exception {
        // Home reference is obtained as on previous slide
        shop = home.create();
    }
    protected void tearDown() throws Exception {
        shop.remove();
    }
    public void testIsInStock() throws Exception {
        assertTrue("Item should be in stock!",shop.isInStock("DUMMY"));
    }
    public void testQuantityInStock() throws Exception {
        assertEquals("Quantity wrong!",20,shop.quantityInStock("DUMMY"));
    }
    public void testGetPrice() throws Exception {
        assertEquals("Price wrong",9.5,shop.getPrice("DUMMY"),0.001);
    }
    public void testMakePurchase() throws Exception {
        assertTrue("Cannot make purchase",shop.makePurchase("DUMMY","DUMMY",1,2.3));
    }
    private Shop shop;
}
```

Classic EJB's Part 2

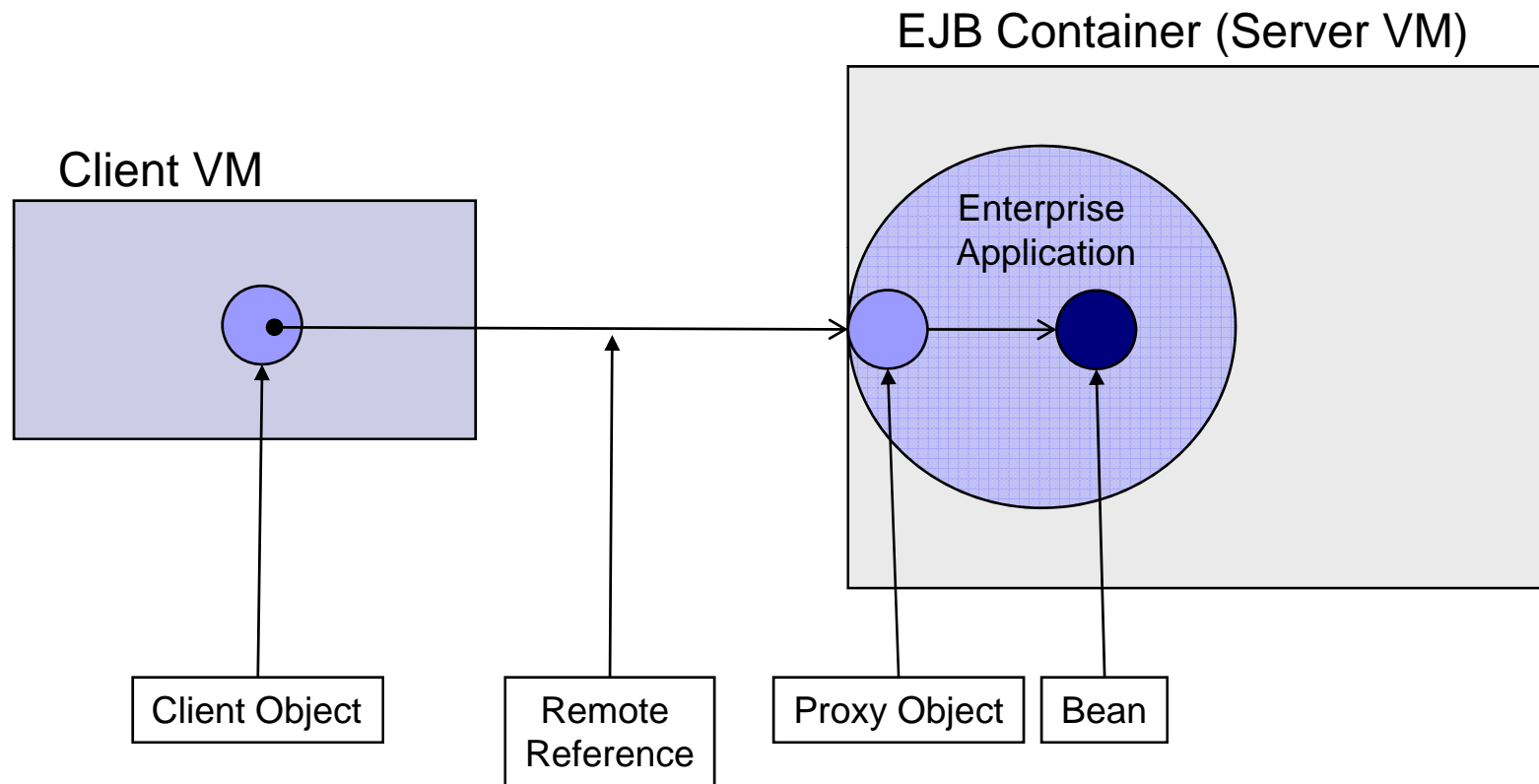
Key Concepts and Syntax



Indirect Access

- It is easy to see why Servlet access is indirect
 - The container turns an HTTP request into a method call
- The indirection in EJB's is more complex
 - The client makes a method call on what looks like the bean
 - In fact the method call is made against a proxy object
 - The proxy object passes the request to an instance of the bean
- This indirection is the key to understanding EJB's
 - Remember that a client call is always received by a proxy object

Indirect Access

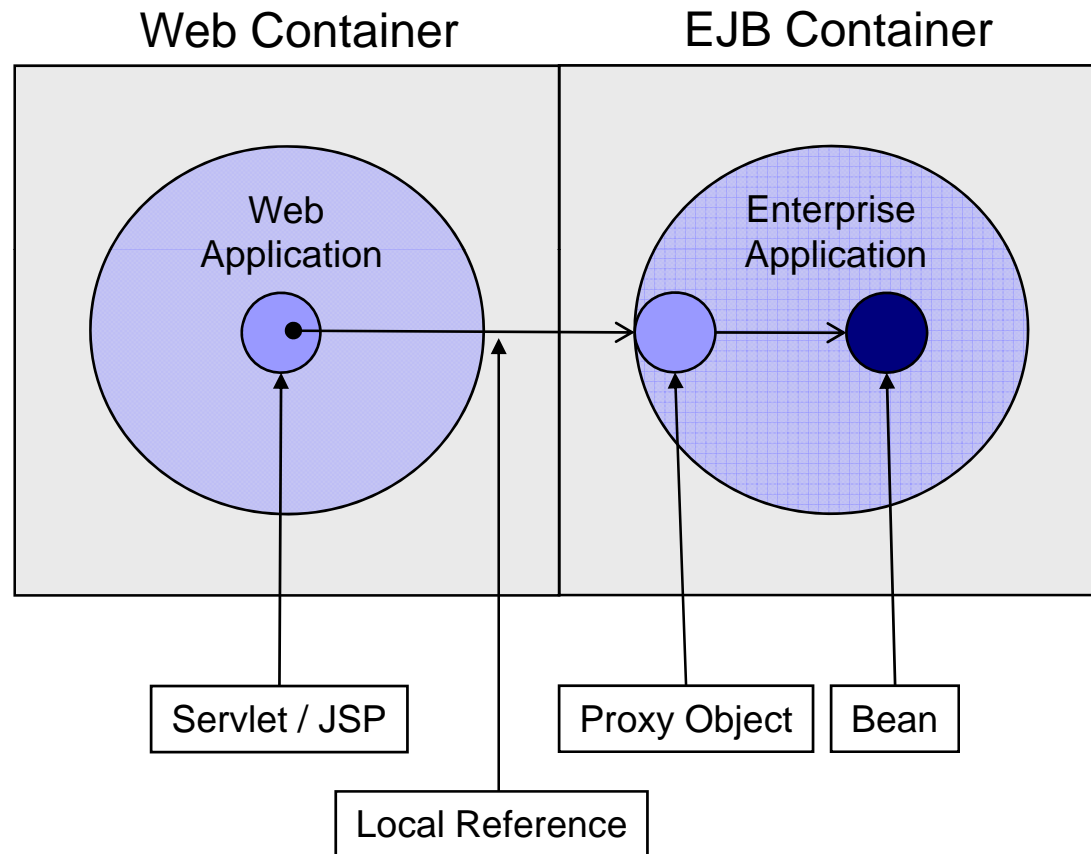




Indirect Access

- Indirection applies inside and outside the server
 - Remote clients call your EJB using Java RMI
 - Parameters must be serializable or remote objects
 - For CORBA compatibility IIOP is the network protocol
 - Local clients call your EJB using normal method calls
 - Parameter passing is the same as conventional Java code
 - Both client and bean will refer to objects passed as parameters
 - But in both cases the method call is indirect
 - It is made against the proxy rather than the bean instance

Indirect Access

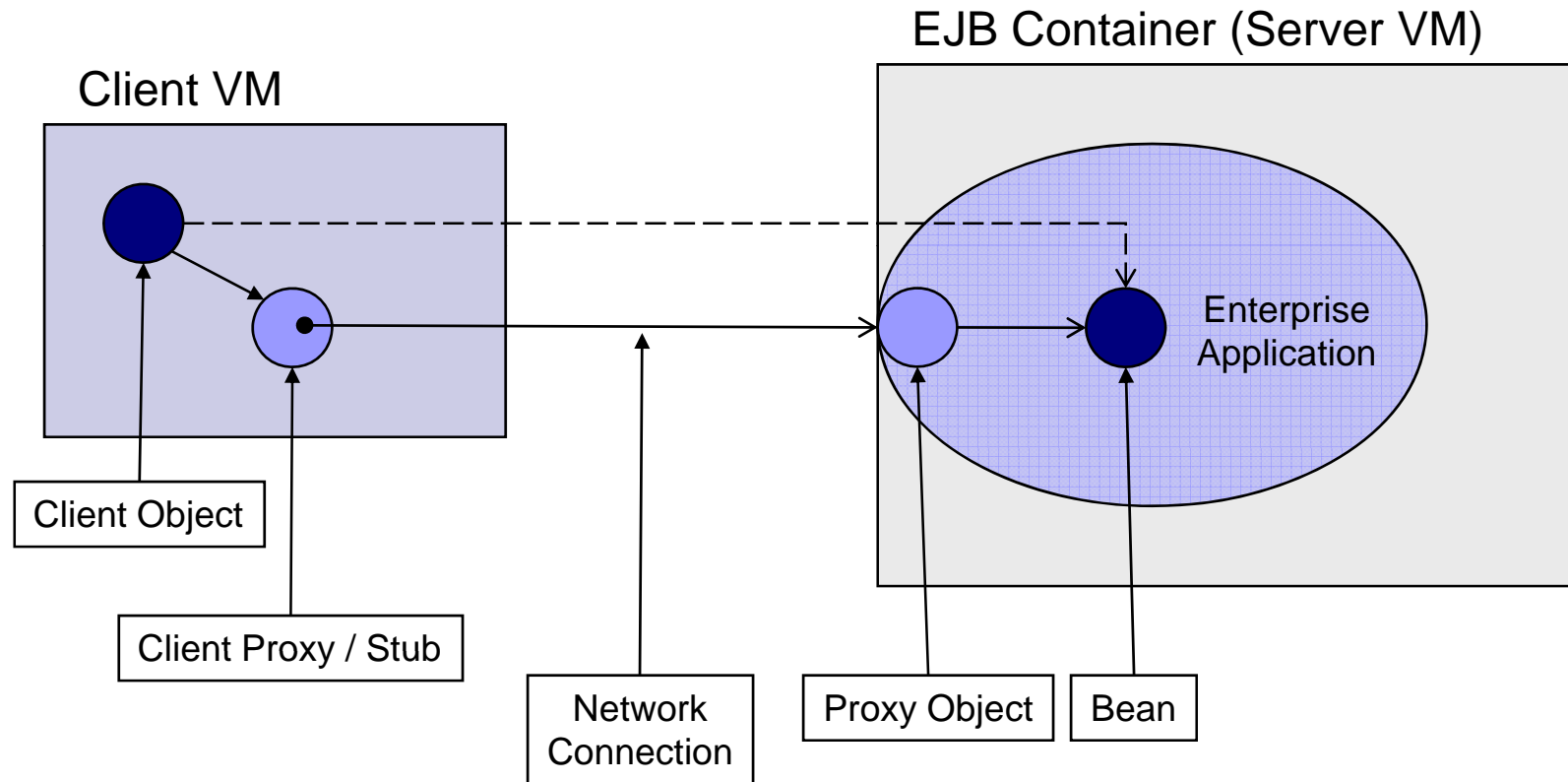




Structure of an EJB

- An EJB is made up of four parts
 1. The bean class itself
 2. Two or four interfaces
 3. The deployment descriptor
 4. Container specific helper classes
- The developer provides the first three items
 - The container generates the fourth at deployment
 - Generated classes are often called 'Container Artefacts'
- In addition remote clients need a 'stub' or 'proxy'
 - To provide marshalling for remote method calls
 - In WebLogic the stubs also provide clustering
- Container artefacts are an integral part of the infrastructure
 - How their responsibilities overlap depends on the container

Structure Of An EJB





Method Calls on EJB's

- When a remote client calls a method on an EJB:
 1. The method call is made on the client side proxy
 2. The client side proxy
 - Marshals the methods parameters
 - Transmits the method call over IIOP
 3. The method call is received by the server side proxy
 4. The server side proxy
 - Selects an available instance of the bean
 - Makes a local call to one of the beans methods
 5. The process is reversed to carry back the return value
- How an instance is chosen depends on the beans type
 - There are five types of EJB, each with a different lifecycle



The Remote Interface

- As in standard RMI remote interfaces are used
 - Remote clients have references of this type
- The remote interface is implemented by the server side proxy
 - Your bean class does NOT implement this interface
- EJB remote interfaces indirectly extend 'Remote'
 - All EJB remote interfaces extend 'javax.ejb.EJBObject'
 - Which in turn extends 'java.rmi.Remote'
 - For this reason the server side proxy is usually referred to as either the 'remote object' or the 'EJB object'
- The 'EJBObject' interface declares utility methods
 - These are implemented for you by the server side proxy
 - Not all of the methods can be used by every type of bean



The Remote Interface

```
package javax.ejb;

import java.rmi.RemoteException;

interface EJBObject extends java.rmi.Remote {
    // Obtain the enterprise Bean's remote home interface
    EJBHome getEJBHome() throws RemoteException;

    // Obtain a handle for the EJB object
    Handle getHandle() throws RemoteException;

    // Obtain the primary key of the EJB object
    Object getPrimaryKey() throws RemoteException;

    // Test if an EJB object is identical to the invoked EJB object
    boolean isIdentical(EJBObject obj) throws RemoteException;

    // Remove the EJB object
    void remove() throws RemoteException;
}
```



Creating an EJB (Step 1)

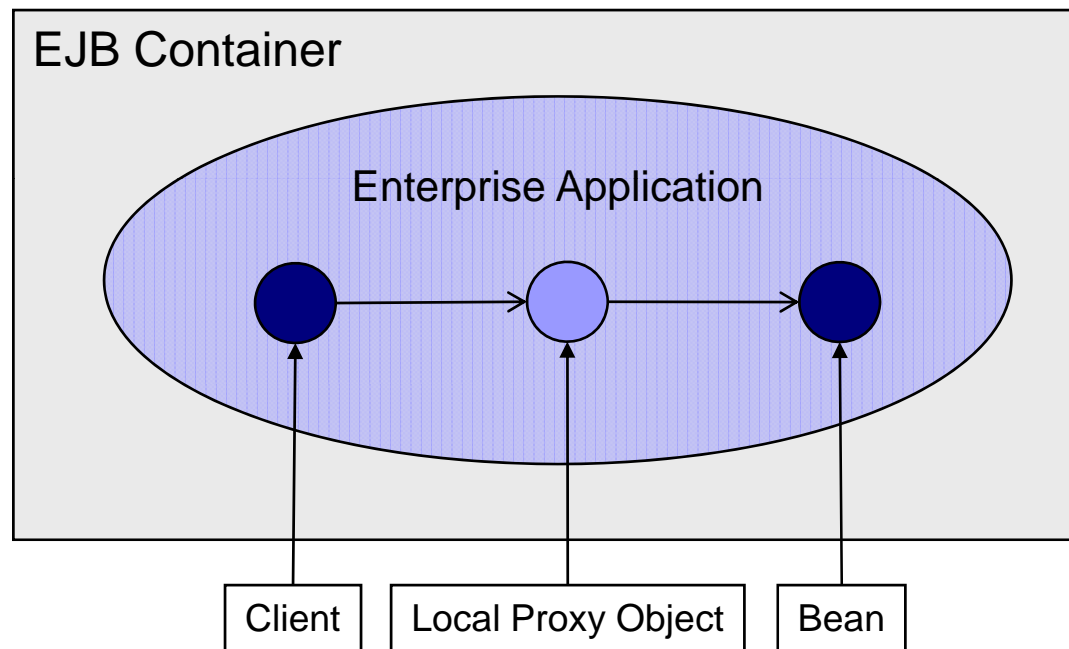
- The first step is to write the remote interface
 - This will contain your business methods
 - The interface will be used by the container to generate the class for the EJB object
 - Remote clients will have a reference of this type
- The remote interface must:
 - Extend the interface 'javax.ejb.EJBObject'
 - Contain methods which
 - Are declared as throwing 'RemoteException'
 - Pass parameters which are serializable



The Local Interface

- Remote interfaces are used for distributed access
 - Originally this was the only way of calling a bean
 - However calls against beans will often be made from the same VM
 - By Servlets or other Enterprise JavaBeans
 - Using RMI inside the same VM is inefficient
 - Although we always use a proxy we want standard method calls
- Access inside the VM is faster using a local interface
 - Local interfaces extend 'javax.ejb.EJBLocalObject'
 - They contain the same business methods as the remote interface
 - EJB's can offer either local or remote access or both
 - Local access is faster but restricts loose coupling
 - Local business methods should never modify parameters

Indirect Access Via Local Interface





The Local Interface


```
package javax.ejb;

interface EJBLocalObject {
    // Obtain the enterprise Bean's local home interface
    EJBLocalHome getEJBLocalHome() throws EJBException;

    // Obtain the primary key of the EJB local object
    Object getPrimaryKey() throws EJBException;

    // Test if an EJB local object is identical to the invoked EJB local object
    boolean isIdentical(EJBLocalObject obj) throws EJBException;

    // Remove the EJB local object
    void remove() throws EJBException;
}
```



Creating An EJB (Optional Step 2)

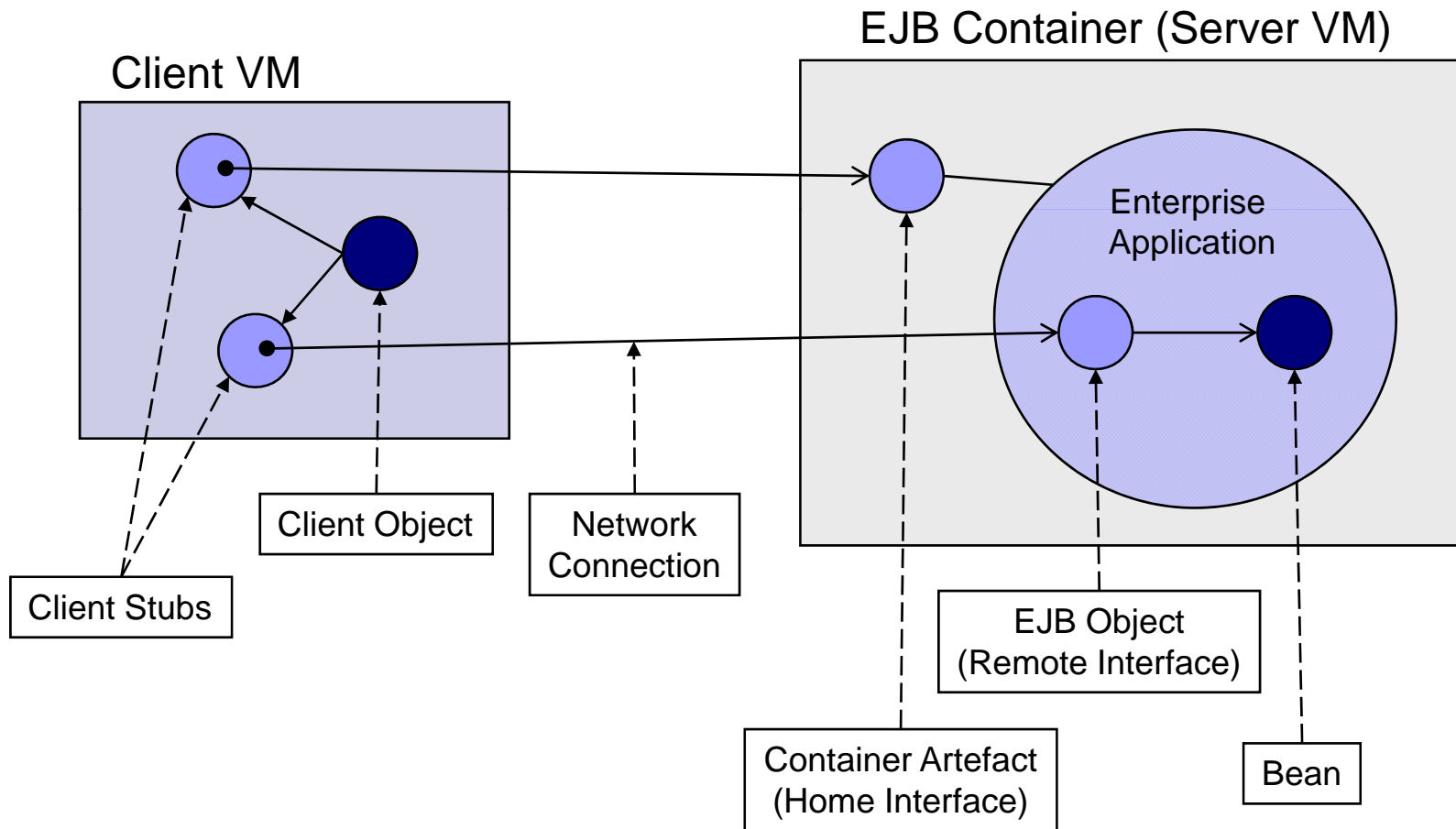
- To speed up local access write a local interface
 - This contains the same business methods as the remote
 - It will be used to generate a proxy for local clients
 - Often called the 'EJB Local Object'
 - The container generated proxy will implement the utility methods
- The local interface must:
 - Extend the interface 'javax.ejb.EJBLocalObject'
 - Declare methods which should throw 'javax.ejb.EJBException'
 - 'EJBException' inherits from 'RuntimeException' so declaring it is optional but helpful



The Home Interface

- So far we have only discussed how beans are used
 - We have not covered how beans are found
- Clients ask the container for a reference to a bean
 - What they get is a reference to a proxy object
- To ask the container for this reference clients:
 - Look up the home interface of the bean via JNDI
 - Call a 'create' or 'find' methods declared in this interface
- A home interface is implemented by a container artefact
 - This is indistinguishable from the container itself

Artefacts Created by the Server





Creating An EJB (Step 3)

- For clients to discover beans write a home interface
 - This contains methods to create or find instances of your bean
 - The signature of the methods depends on the type of the bean
 - All creation methods must be called 'create'
 - All finder methods must begin with 'find'
- The remote home interface must:
 - Extend the interface 'javax.ejb.EJBHome'
 - Declare one or more 'create' or 'find' methods
 - All methods must be declared as throwing 'RemoteException'



The Remote Home Interface

```
package javax.ejb;

import java.rmi.RemoteException;

interface EJBHome extends java.rmi.Remote {
    // Obtain the EJBMetaData interface for the enterprise Bean
    EJBMetaData getEJBMetaData() throws RemoteException;

    // Obtain a handle for the remote home object
    HomeHandle getHomeHandle() throws RemoteException;

    // Remove an EJB object identified by its handle
    void remove(Handle handle) throws RemoteException;

    // Remove an EJB object identified by its primary key
    void remove(Object primaryKey) throws RemoteException;
}
```



Creating An EJB (Step 4)

- We have just described the remote home interface
 - This is only used by remote clients
- Local clients also need to discover instances of the bean
 - They also use JNDI to find a container artefact
 - Discovery is accomplished via a local home interface
- To write a local home interface
 - Extend the interface 'javax.ejb.EJBLocalHome'
 - Declare one or more 'create' or 'find' methods
 - All the methods should be declared as throwing 'EJBException'



The Local Home Interface

```
package javax.ejb;  
  
interface EJBLocalHome {  
    // Remove an EJB object identified by its primary key  
    void remove(Object primaryKey)  
}
```



Creating an EJB (Step 5)

- Having written the appropriate interfaces the next stage is to write the bean class itself
 - Paradoxically the bean class does not implement ANY of the interfaces you have created
 - The class does implement an interface specific to the bean type
 - This contains callback methods used for lifecycle management
- To write an EJB implementation class
 1. Create a class which implements the appropriate bean interface
 - Either 'EntityBean', 'SessionBean' or 'MessageBean'
 2. Provide an implementation for the callbacks in this interface
 3. For each business method declared in the home and/or remote interface declare a method with the same signature
 4. For each 'create' method in the home interface(s) declare a method called 'ejbCreate' taking the same parameters



The Bean Class

- Your bean does not implement home interfaces
 - The methods declared there are implemented by the container
 - You declare 'ejbCreate' methods so the container can pass along initialization parameters from the client
- Your bean does implement local or remote interfaces
 - Because the methods declared there will be called on the EJB object or EJB local object
 - When the proxy receives a business method call it will trigger the method with the same signature in the bean instance
- The compiler does not check for these correspondences
 - But the container tool will when your bean is deployed
 - If possible use the containers EJB compiler in isolation



The Deployment Descriptor

- The deployment descriptor is the final part of the EJB
 - It fulfils much the same role as 'web.xml' in a web application
- The deployment descriptor tells the container:
 - What class files contain the interfaces and bean class
 - What the specific type of the bean is
 - The transaction handling and security policies required
 - What external resources are used by the bean
 - Initialization parameters can also be supplied
- Containers are free to specify their own additional files
 - The standard deployment descriptor is 'ejb-jar.xml'
 - WebLogic requires an extra file called 'weblogic-ejb-jar.xml'



Creating an EJB (Step 6)

- Write the standard and container specific deployment descriptors for your bean

```
<?xml version="1.0"?>
<!DOCTYPE weblogic-ejb-jar PUBLIC "-//BEA Systems, Inc.//DTD WebLogic 6.0.0 EJB//EN"
"http://www.bea.com/servers/wls600/dtd/weblogic-ejb-jar.dtd">
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>Test</ejb-name>
    <jndi-name>EJBTest</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```



Creating an EJB (Step 6)

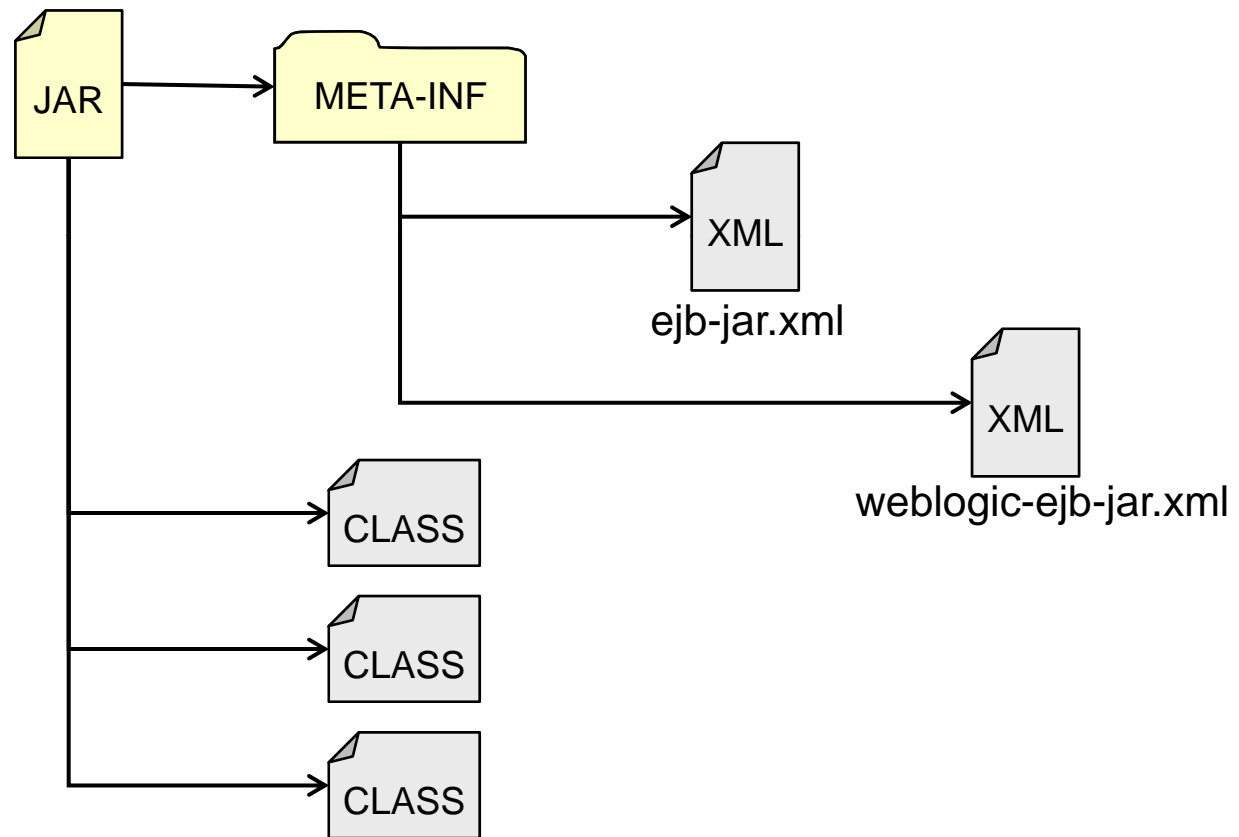
```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>Test</ejb-name>
      <home>com.demos.ejb.TestHome</home>
      <remote>com.demos.ejb.Test</remote>
      <ejb-class>com.demos.ejb.TestBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```



Deploying The Bean

- Enterprise Beans are deployed in a JAR file
 - Unlike Web Archives there is no special file extension
- Multiple beans may be deployed in the same JAR
 - Beans which use each other should be packaged together
- The structure of the archive is very simple
 - Class files are packaged directly into the JAR file
 - In their appropriate package hierarchy
 - Deployment descriptors are placed inside META-INF
 - The 'Class-Path' attribute in the manifest file can be used to add external JAR files to the beans classpath
 - This is very useful when Servlets and EJB's share common libraries

EJB Archives

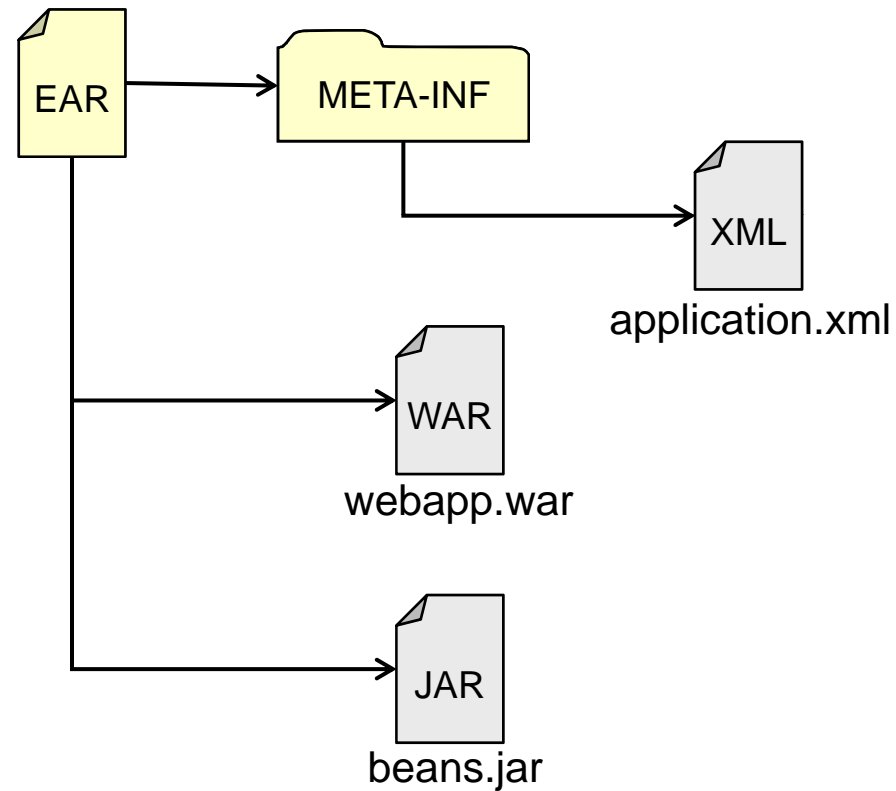




Deploying An Application

- A JAR or WAR archive is a JEE Module
 - Containers support the deployment of modules in isolation or in combination (as J2EE applications)
 - The same module can be deployed several times
 - As long as it is placed in different contexts
- A J2EE application is made up of several modules
 - A WAR archive, one or more JAR archives containing Enterprise JavaBeans and any additional libraries
- J2EE applications are deployed as an EAR archive
 - The JAR file must have the enterprise archive extension
 - The file contains a deployment descriptor called 'application.xml'

EAR Archives





EAR Archives

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
"http://java.sun.com/dtd/application_1_3.dtd">
<application>
  <display-name>Test Application</display-name>
  <module>
    <web>
      <web-uri>webapp.war</web-uri>
      <context-root>webapp</context-root>
    </web>
  </module>
  <module><ejb>beans.jar</ejb></module>
</application>
```