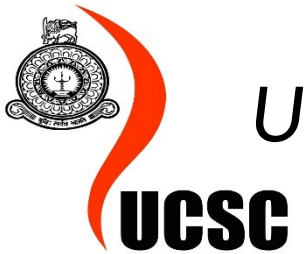
Advanced Java Application Development Using JavaEE



University of Colombo School of Computing

Entity Beans

- A entity bean is a remote object that manages persistent data.
- This performs complex business logic, potentially uses several dependent Java object, and can be uniquely identified by a primary key. Entity beans are normally coarse-grained persistent object, in that the utilized persistent data stored within several fine-grained persistent object.
- Fine-grained persistent Java objects typically manage persistent data that has a one-to-one mapping between the data and a table column.

Fine-Grained versus Coarse-Grained Object Access

- Entity beans with local interfaces provide efficient access to fine-grained objects. Model these objects as local entity beans when you can assure that their clients are co-located on the same JVM, or implement the business objects with a local view and provide a remote enterprise bean facade to them.

- As mentioned earlier, enterprise beans that are implemented as remote objects may consume significantly more system resources and network bandwidth to execute. Because of the overhead of remote access, remote entity beans should not be used for fine-grained access. Therefore, only model a business object as a remote entity bean if its clients are distributed. In such a case, keep the entity bean access coarse grained, and use a value object to pass data across the remote interface.

http://sites.google.com/site/sureshdevang/coarse-grained-objects-and-fine-grained-objects

Fine-Grained versus Coarse-Grained Object Access

- A value object is a serializable Java object that can be passed by value to the client. A value object can be used to aggregate state extracted from a remote entity bean for use by the client. By avoiding the use of fine-grained remote method calls to retrieve the persistent state of an entity bean, value objects help reduce the network overhead involved in remote access.

- The sample application models the details of an order as a value object representing the state of a particular order in the database and providing getter methods to query the state of this account. An administration client makes just one remote call to execute getOrdersByStatus on the remote object account and gets back a collection of the serialized OrderDetails objects. The client can then query the state of these orders locally via the methods provided with the OrderDetails object.

http://sites.google.com/site/sureshdevang/coarse-grained-objects-and-fine-grained-objects

Fine-Grained versus Coarse-Grained Object Access

- In reference to entity bean accessor methods: a "fine grained" entity bean is one that has properties accessed through individual properties (getXXX, setXXX). This is usually simpler (and the way most textbook examples look), but experience in distributed environments tells us to minimize remote calls; this is where the "ValueObject" pattern is used.(Access to the entity bean is then "coarse grained".(More information is on the Wiki: http://www.c2.com/cgi-bin/wiki?EjbIdioms)

- The way that typically use the Coarse/Fine grained entity bean terminology is in EJB design. In designing EJBs, my thoughts of coarse grained vs. fine grained usually depend on how many tables, or what kind of transaction, the bean is executing. For example, if we take a common object in most business senses, a "Purchase Order" has a large amount of data associated with it. (Individual line items, purchaser information, etc.)

Fine-Grained versus Coarse-Grained Object Access

- In this case, a "Purchase Order" may be the item you wish to "save" in a transaction. There are two ways that I consider this problem:

  1) Having "Purchase Order" be my entity bean. (Likely BMP). The entity bean is basically responsible for persisting all items related to the purchase order, and can span multiple database tables. (This is what I typically refer to as "coarse grained" design.)

  2) Having a "Save" function for a Purchase Order be a facade that controls many EJBs, one for purchaser information, one for line items, etc. (This may be a one-bean to one-table-or-persistance-mechanism.) This is typically what I have referred to as "fine-grained" design.

http://sites.google.com/site/sureshdevang/coarse-grained-objects-and-fine-grained-objects

UCSC

Managing Persistence Data

- An entity bean manages its data persistentcy through callback methods, which are defined in the javax.ejb.EntityBean interface. When you implement the EntityBean interface in your bean class, you develop each of the callback functions as designated by the type of persistence that you choose: bean-managed persistence or container-managed persistence. The container invokes the callback functions at designated times. That is, the contract between the container and the entity bean designates the order that the callback methods are invoked and who manages the bean's persistent data.

Uniquely Identified by a Primary Key

- Each entity bean has a persistent identity associated with it. That is, the entity bean contains a unique identity that can be retrieved if you have the primary key. Given the primary key, a client can retrieve the entity bean. If the bean is not available, the container instantiates the bean and repopulates the persistent data for you.

- The type for the unique key is defined by the bean provider.

Performing Complex Logic Involving Dependent Objects

When designing your EJB application, you need to keep in mind the aspects of each type of object.

Session beans are typically used for performing simple tasks for a remote client.

Entity beans are typically used for performing complex tasks that involve coarse-grained persistence for remote clients.

Java objects, persistent or otherwise, are used for simple tasks for local clients.

Enterprise JavaBeans are remote objects and are used for interacting with clients over a network. Remote objects have a higher overhead for verifying security and transaction information. Thus, when you design your application, you may have an entity or session bean interacting with the client, but also have the Enterprise JavaBean invoke other dependent Java objects to perform tasks or manage persistent data.

Performing Complex Logic Involving Dependent Objects

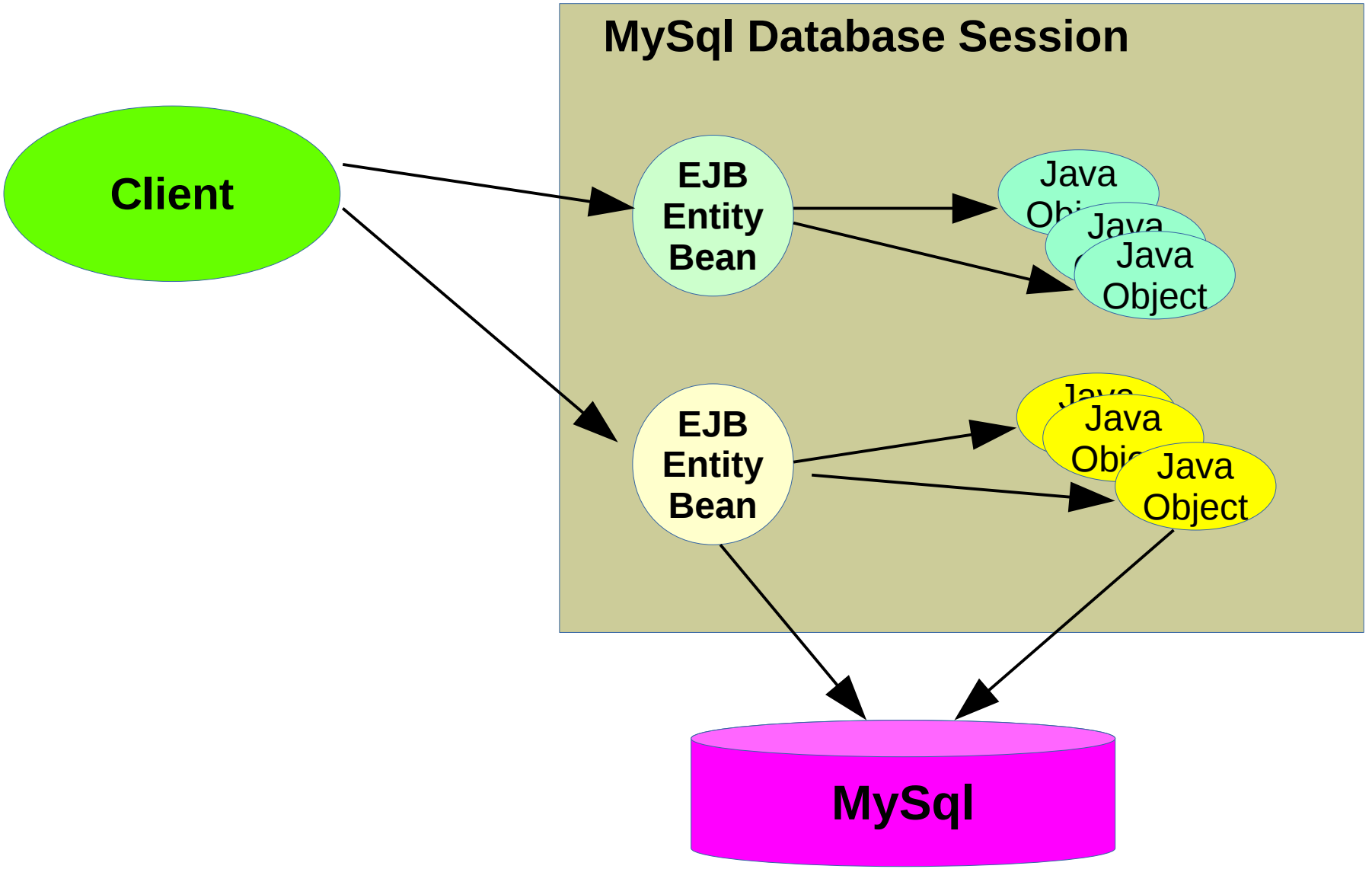
UCSC

CSC Computing Services Centre

- Entity beans are normally used to manage complex, coarse-grained persistent data for a remote client. Be careful to separate the difference between an entity bean and a persistent object. Your entity bean should be more than just a persistent object; it should manage and return complex data to justify using a remote object for managing data.

- You can have an entity bean that calls one or more dependent objects within the application. The entity bean is a remote object and thus its primary function is interacting with the client over the network. You should not have an entity bean invoking another entity bean within the same node on the network. If you need to design multiple objects within your application, design your application so that the entity bean facilitates the communication and data management between the client and other Java objects.

Performing Complex Logic Involving Dependent Objects



CMP and BMP Persistence

Entity EJBs support two persistence options: Bean Managed Persistence (BMP) and Container Managed Persistence (CMP).

With BMP, an entity bean contains its own database access code. This code includes methods that execute SQL commands against the database. For example, a bean might include methods that create new bean instances (SQL INSERT) as well as methods that retrieve existing bean instances from the database (SQL SELECT). BMP can enhance a bean's performance because it allows the database access code to be optimized for a specific database. Of course, developing the database access code can be very labor intensive and can involve complex object-relational mappings.

CMP and BMP Persistence

With CMP, an entity bean contains no database access code. Instead, the EJB container generates all of the required database access code based upon a specification provided in an XML file called the abstract schema. CMP enhances a bean's reusability by removing any code that might be database-specific from the bean. CMP also speeds development by freeing developers from having to create the database access code.

The following table provides a summary comparison of BMP and CMP:

CMP and BMP Persistence

Type	Efficiency	Development Effort	Re usability
BMP	Higher	Higher	Lower
CMP	Lower	Lower	Higher

Sample JPA

```
@Entity
public class Employee implements java.io.Serializable
{
	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)
	private long id;

	@Column (name="NAME")
	private String name;

	private int age;

	@Column (name="AGE")
	public int getAge()
	{
		return this.age
	}
	public String getName()
	{

	}
	// other getter/setters
}
```

UCSC

Ways of persisting data

Computing Services Centre

There are currently many ways to persist data. Some of them are:

- Relational Databases (RDBMS) - accessed through a combination of JDBC and SQL,
- File systems,
- Object databases (ODBMS), and more recently,
- Entity EJB (both Container Managed and Bean Managed), and
- Java Data Objects (JDO)

Both Entity Beans and JDO (like EAI) are persistence frameworks layered on top of the data store

An ideal Persistence layer should have the following characteristics

an ideal Persistence layer should have the following characteristics:

- It should be extremely simple to use
- It should have minimal intrusion into the application code.
- It should be extremely transparent.
- It should have a consistent set of APIs
- It should provide Transaction support
- It has to provide managed-environment support
- It should provide the ability to query the data-store
- It should provide for efficient cache-management to improve application performance
- It should provide object to relational mapping for relational databases.

Issues with using Entity Beans for Persistence

Entity Beans are best suited for coarse-grained business components; they should not be used to represent fine-grained objects.

Avoid mapping your object model directly to the Entity Bean model.

Avoid mapping your relational model directly to the Entity Bean model.

Entity Bean to Entity Bean relationships introduces severe performance penalties. This often results when you directly map your object model or relational model into an Entity Bean model.

Do not store an object graph of Entity Beans

The semantic differences between local (pass by reference) and remote (pass by value) invocation introduces a lot of issues even if some of the flaws have been addressed in the EJB 2.0 specification.

The lack of any meaningful support for inheritance is also a cause for concern.

Persistence and query functions have to be usually hand-coded (in SQL with JDBC) or described by hand (in Enterprise JavaBeans Query Language – EJBQL – which stems from SQL).

There are concurrency issues endemic to the EJB Threading Model.

Gross inefficiencies are possible when manipulating large datasets.

Inheritance Issues: True inheritance has never been possible with Entity Beans.

Issues with using Entity Beans for Persistence

This is because EJBs were always intended to be self-contained components as opposed to domain objects. Even though code sharing can be accomplished by inheriting from entity bean classes or by inheriting from remote interfaces, true inheritance with the overall component is absolutely not possible. By that I also mean true Java inheritance – a client cannot down-cast or up-cast and entity bean's EJBObject stub to a parent or subclass, etc.

Managing Object Identity issues: Each Entity Bean has to implement a Primary Key class. If JDO were used, the implementer can have a choice of using either Application Managed Identity like Entity Beans, or Data-store Identity. JDO can also manage Object Identity on its own using the Data-store Identity approach.

Issues with using BMP Entity Beans for Persistence

Data Store Connection Issues: The Bean implementer has to implement code in the Entity Bean that connects to the data store and loads and retrieves data (all the different ejbXXX … methods). This happens transparently if a JDO implementation were used.

Portability Issues: Entity Beans with BMP are assumed to be portable if developers embedded their SQL statements into external resources, and used java.sql.DataSource objects provided by the container to access these external SQL statements. However, this overlooks some scenarios where this approach may not be as portable as we were led to assume. One of them is the case where the implementer may want to switch the Bean implementation to use, say, scrollable Prepared Statements to speed-up their implementation. The whole persistence mechanism has to be re-coded. Another scenario is that it doesn't help very much if the implementer wanted to move from using a Relational Database to an Object-Oriented Database that may not understand JDBC. In such a case too portability is lost as the persistence part has to be rewritten once again from scratch. This is not a problem with JDO because persistence is transparent in JDO, and JDO was intended to persist Java objects in a transactional data store (be they RDBMS, or ODBMS, or flat-files) without the need to explicitly manage the storage and retrieval of individual fields.

Issues with using BMP Entity Beans for Persistence

No Lazy Data Loading: By virtue of coding an ejbLoad method, we ensure that the whole object is loaded into memory completely when the EJB container invokes this method. However, a JDO implementation would use Lazy Data Loading and would only load those persistent fields defined in the Default Fetch Group (DFG). Other fields get loaded as they are referenced.

Relationship Implementation Issues: Relationships between classes have to be manually implemented. It's not possible to access other related EJB objects directly. We have to implement code that goes through the Home Interface of the related object to invoke them. This is not a problem with JDO implementations as JDO provides automatic navigation between persistent domain objects in the model. With JDO, there is no extra code that needs to be added by the implementer.

No Caching support: Unlike Container Managed Persistence, there is no guarantee that all database calls go through a singleton layer below. This makes it difficult to implement an efficient caching scheme. This is not a problem when using a JDO implementation as JDO specifies the presence of an object cache associated with each Persistence Manager.

Issues with using CMP Entity Beans for Persistence

EJBQL definition in Server-side Descriptors: As all EJBQL queries are defined in deployment descriptors on the Application Server, it is impossible for a client to use client-side query functionality. For example, a method like findObject(Query query) does not exist.

No Aggregate function support in EJBQL: EJBQL is too simplistic. There is no support for Aggregate functions which are essential required functionality when dealing with Relational Databases. Similarly, support for updating a set of rows instead of one just does not exist in EJBQL today.

No Lazy Data Loading: With the current specification, it is not possible to configure lazy loading of large result sets from the client. There is no point in using a findAll() method which may return a large result set of beans if you just need to access a couple of them. It is also possible that one may reference a huge collection of other beans from one and access only a few beans from the referenced collection. This will have a very telling affect on performance.

Issues with using CMP Entity Beans for Persistence

No support for Fetch Groups: It is not possible to tell the EJB Container to selectively load relationships rather than loading everything when a bean is accessed. The only exception to this is BEA WebLogic's CMP container

.

No Raw ResultSets: The resulting Result Set as the result of running an EJBQL query will have to contain EJBs. This is really important as reports will need to access an intersection of bean types, and then select a property from there.

No Access to Statement Generation: ANSI SQL has to be used for statement generation (for INSERT, SELECT, UPDATE, or DELETE operations). The only exception is stored procedures. It is not possible to use other SQL enhancements provided by your data store provider to enhance performance.

Database mapping is too simple: The default CMP implementation mapping of One Bean type to One Table and One Bean Instance to One row is too simplistic. CMP Entity Beans provides no way to customize mapping strategies.

Entity Manager

In the Java Persistence specification, the EntityManager is the central authority for all persistence actions. Because entities are plain Java objects, they do not become persistent until application's code explicitly interacts with the EntityManager to make them persistent. The EntityManager manages the object-relational (O/R) mapping between a fixed set of entity classes and an underlying data source. It provides APIs for creating queries, finding objects, synchronizing, and inserting objects into the database. It also can provide caching and manage the interaction between an entity and transactional services in a Java EE environment such as the Java Transaction API (JTA). The EntityManager is well integrated with Java EE and EJB but is not limited to this environment; it also can be used in plain Java programs.

Entity Manager

Initialize Entity Manager

@***P***ersistenceContext
EntityManager entityManager;

For Save

entityManager.persist(object);

If primary key of the entity is auto generated use merge();

For Retrieve Data

Query query = entityManager.createQuery(“”);
List<> list = query.getResultList();

Hibernate Session

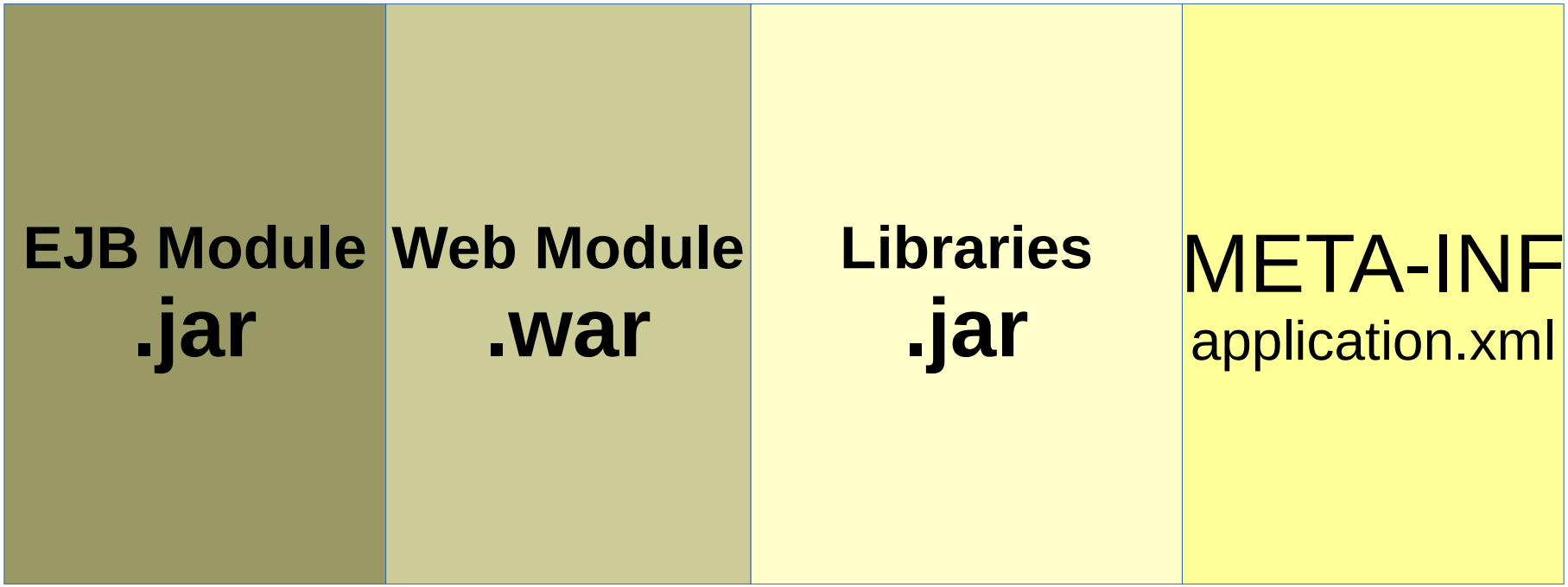
Can use Hibernate Session with EntityManager

```
@PersistenceContext
EntityManager entityManager;
```

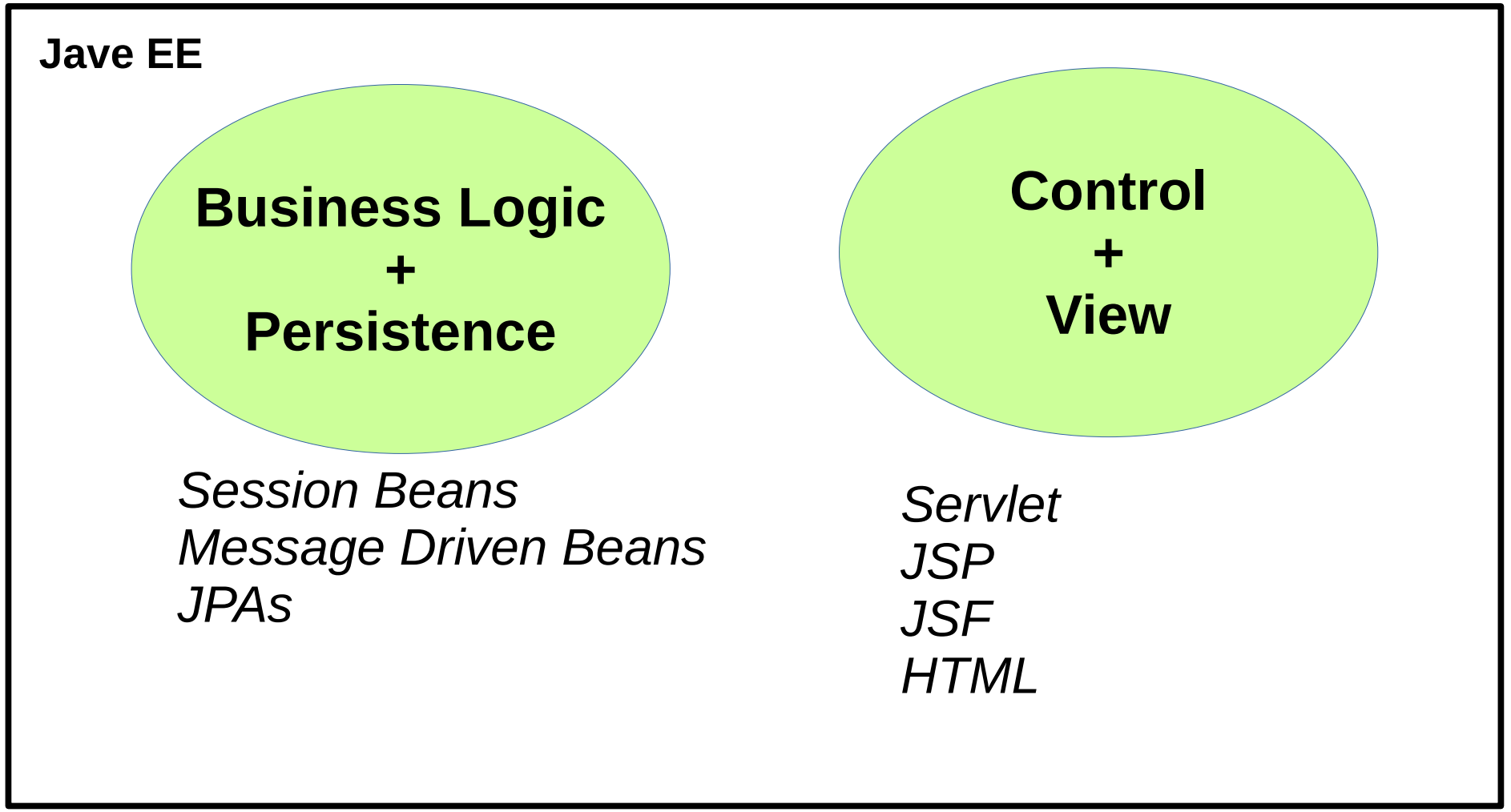
```
Session  session = (Session)entityManager.getDelegate();
```

Anatomy of EJB 3.1 Packaging

Structure of .ear Package

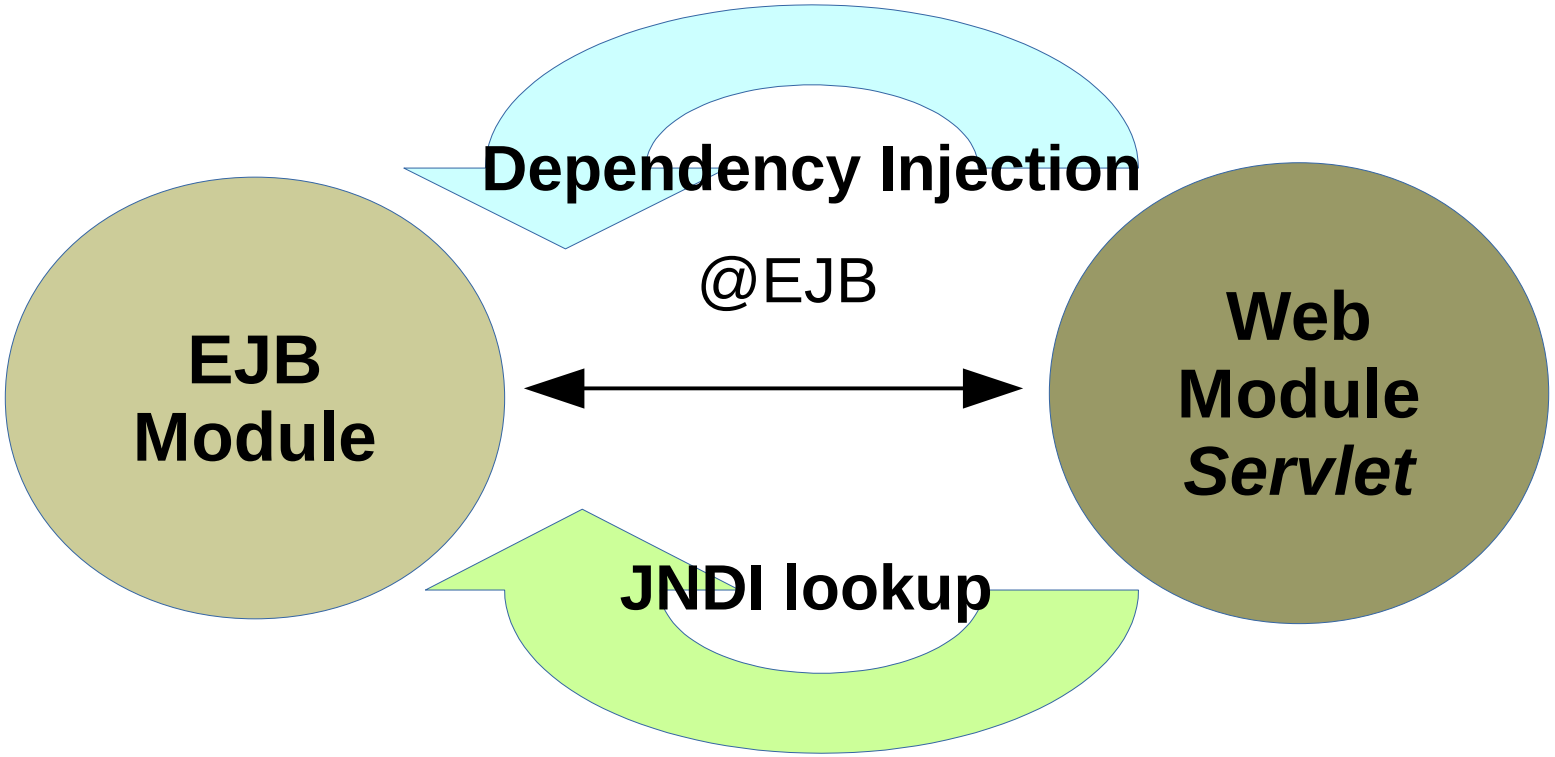


Analysis of Java EE



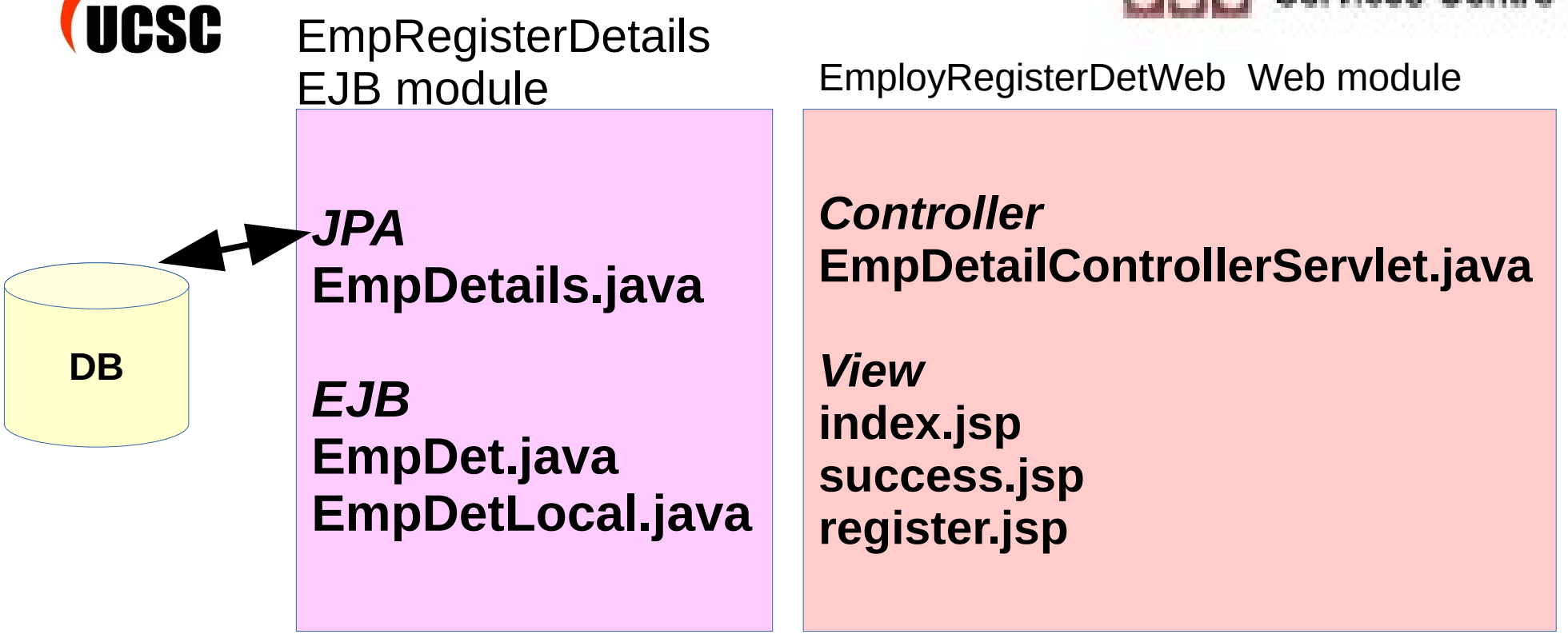
Accessing EJBs

Accessing of EJBs are two ways



java:module/EmpDet!com.ejb.EmpDet

Employee Registration Application



Dependency Injection

```
@EJB
Private EmpDetLocal   localBean;
```

Accessing Session Beans (JNDI)

```
Context  context = null;

Try{
        context = new InitialContext();
        LocalBean = (EmpDetLocal)context.lookup
                ("java:global/EmpRegisterDetWebEAR
                /EmpRegisterDetWeb/EmpDat!com.ejb.
                        EmpDet");
            }catch(){

}
```

Context Interface

- In order to access bean, it needed to initialize and assign an instance of the bean to the localBean variable.
- For this, the bean needed to be looked up using an object implementing the Context interface.
- The Context interface provided necessary information to locate the server and to create a reference to a EmpDet object. Therefore, needed to create an InitialContext object. This class implemented the Context interface.
- Exceptions resulting from the creation of the InitialContext object and the subsequent lookup method were caught.
- Once the Context has been established the Context object's lookup method was invoked and a reference to the EmpDet EJB was provided. The lookup method and a portable JNDI name to identify the EJB. In this case global name was used.

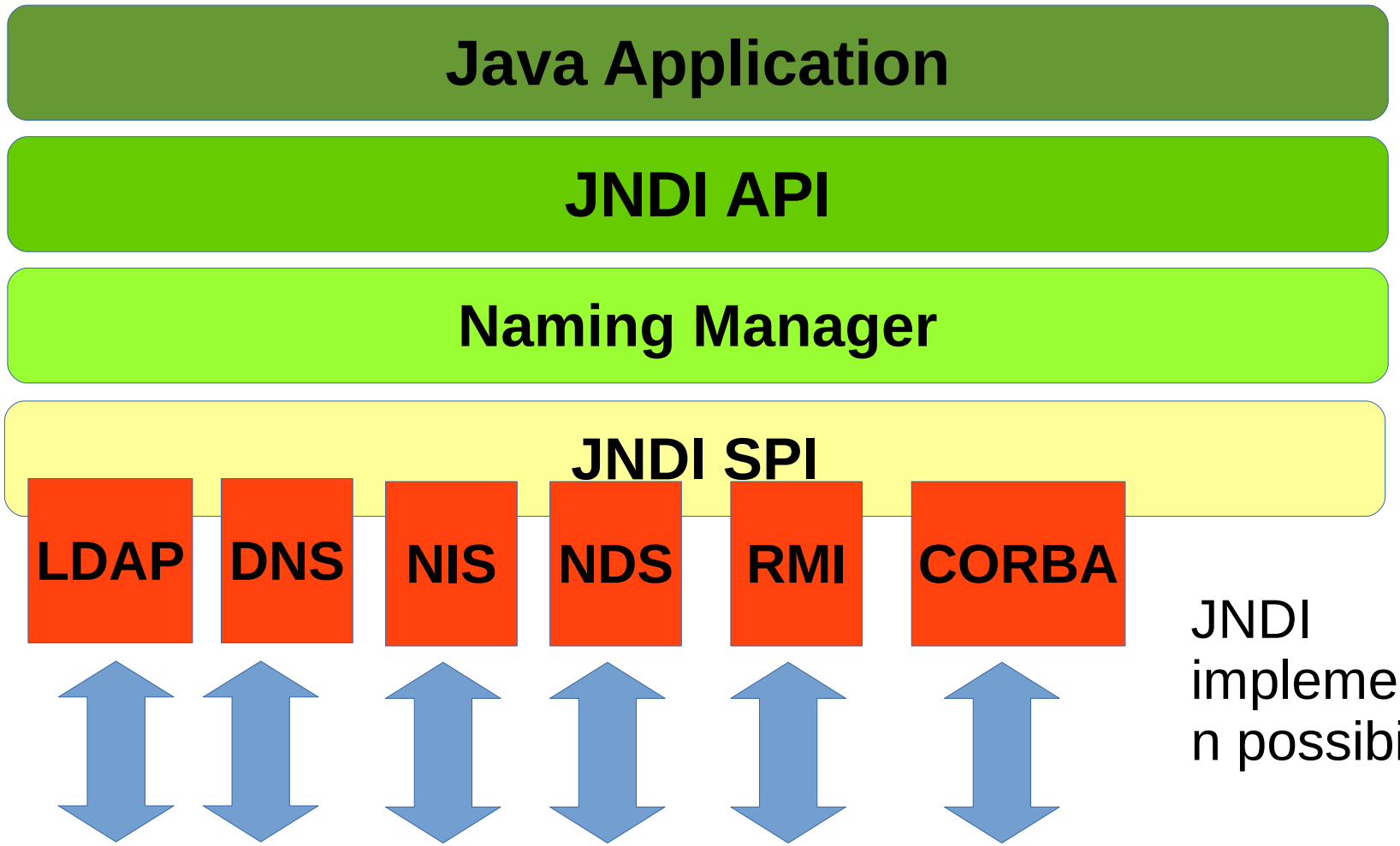
JNDI Overview

- JNDI stands for “Java Naming and Directory Interface” is and application programming interface (API)
- This provides naming and directory functionality to applications written using the Java™ programming language.
- This is defined to be independent of any specific directory service implementation.
- JNDI provide a common way to access variety of directories. (new, emerging, and already deployed directories).

JNDI Architecture

- JNDI architecture encompasses with and API and Service Provider Interface (SPI).
- To access variety of naming and directory services by Java applications that need to use JNDI API.
- The SPI enables a variety of naming and directory services to be plugged in transparently.
- This allows the Java applications using the JNDI API to access their services.

JNDI Architecture



JNDI implementation possibilities

JNDI Packaging

- JNDI is included in the Java SE platform.
- To use JNDI application must have the JNDI classes and one or more service providers. The JDK includes service providers for the following naming/directory services.
- Lightweight Directory Access Protocol (LDAP).
- Common Object Request Broker Architecture (CORBA).
- Java Remote Method Invocation (RMI) Registry.
- Domain Name Service (DNS).

Service providers can be downloaded from JNDI page

URL : www.oracle.com/technetwork/java/jndi/

JNDI Packaging

- JNDI is divided into five packages

    1. javax.naming
    2. javax.naming.directory
    3. javax.naming.ldap
    4. javax.naming.event
    5. javax.naming.spi

- JNDI is an application like JDBC.
- It is implemented by other software vendors.
- JNDI provides the way by which one can the objects binds in context. It uses directory kind of structure to bind the objects which is accessible by passing simple string showing logical directory structure. eg. “jdbc/DBI”. Implementation of JNDI first parse the string passed then fetch the object stored in the context from the location requested by string.

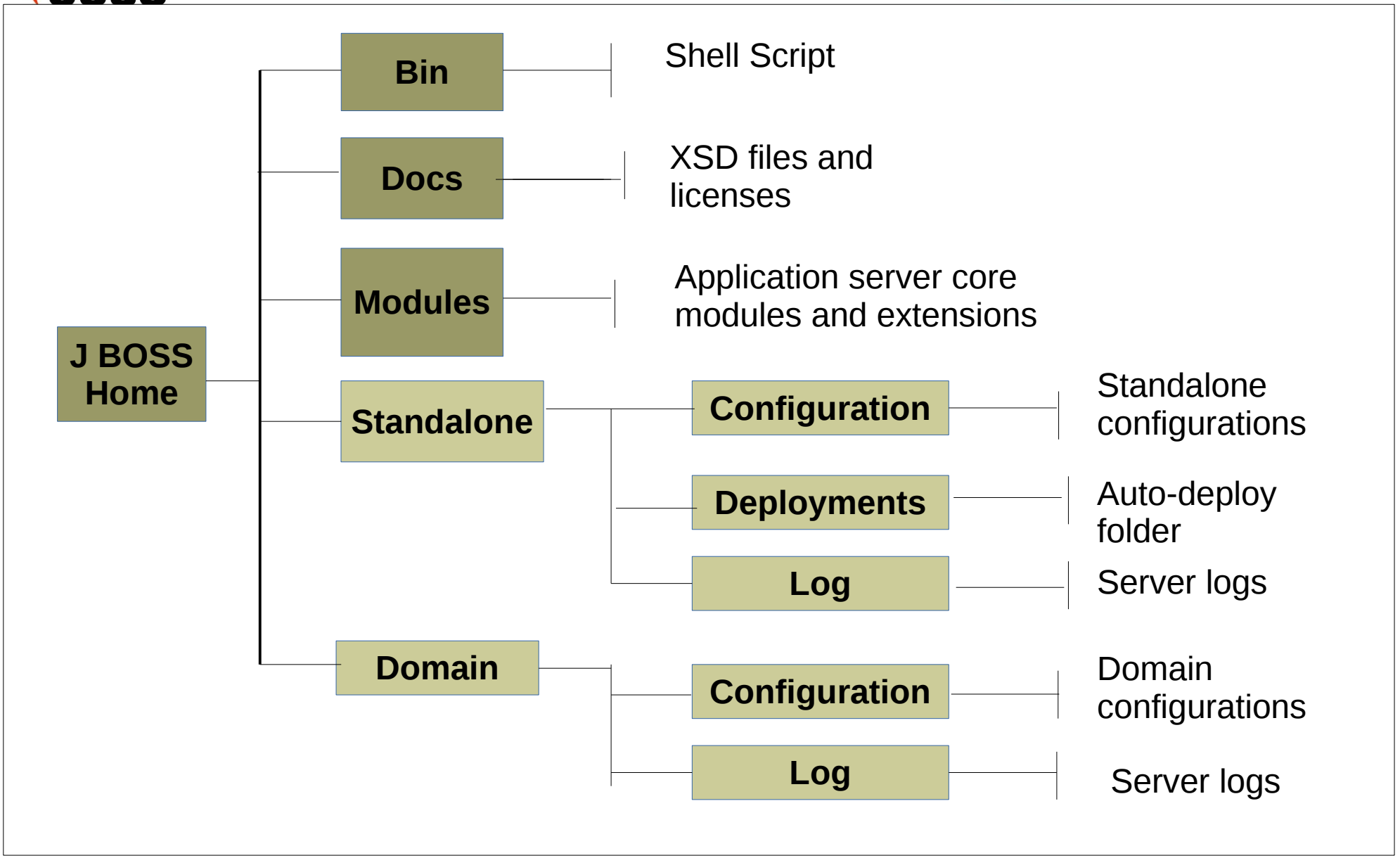
JNDI Naming Syntax

- JNDI is used to lookup resources such as session beans within an application and across the network.
- A JNDI server allows resources to be registered and then client can lookup and use these resources.
- Each EJB is automatically assigned a unique name by the server though it is possible to assign a specific name if desired. This name is then combined with one of three JNDI namespace prefix to form a complete portable JNDI name.

String prefix	Visibility
“java:global/”	A globally accessible name
“java;app/”	Can be seen in the same application
“java:module/”	Can be seen in the same module

Java:global [/<app-name>]/<module-name>/<bean-name>

Directory Structure of Application Server



Building Tools

- Using building tools facilitate integrated developments.
- It facilitate dependency management.
- It support and automate complete build, build, jar, apply static code analysis, run the unit tests, generate the documentation copy to some directory, tune some properties, depending on the environment,etc.
- Once automated it can use a continuous integration system which builds the application at each change or every hour to make sure everithing still builds and the tests still pass….
- Eclipse, NetBeans, Idea are development environments not build tools.

Java Build Tools

- First build tool is Make.
- Later it was improved with GNU Make
- As requirement several build tools are sprung up later to cater demand
- JVM ecosystem is dominated with three build tools Apache Ant with Ivy, Maven and Gradle.

Ant with Ivy

- _Ant is the first build tool among modern build tools. This is similar to Make.
- Ant released on 2000 and within short period of time it became the most popular build tool for Java projects.
- Major drawback was XML as the format t write build script.

Java Build Tools

- XML being hierarchical in nature, is not a good fit for procedural programming approach that Ant uses.
- Hammering, problem with Ant is that XML tends to became unmanageably big when used with all but very small project.
- Later, Ant adopted Apache Ivy due to dependency management over the network became essential part of developments.
- Main benefit of Ant is, its control of the build process.

Maven

- Maven was released in 2004.
- It improved upon some problems that developers were facing when using Ant.
- Maven continuous using XML as the format to write build specifications. However, structure is dramatically different.

Java Build Tools

UCSC

CSC Computing Services Centre

- Ant requires to write all commands.
- Maven uses available targets (goals) that can be invoked.
- Maven introduced the ability to download dependencies over the network (later adopted Ant through Ivy).
- However, Maven itself has its own problems.
- Dependencies management does not handle well conflicts between different versions of the some library (Something Ivy is much better at).
- XML as the build configuration format is strictly structured and highly standardized. Customization of targets (goals) is hard.
- Therefore, Maven is focused mostly on dependency management, customized build scripts are actually harder to write in Maven than in Ant

Java Build Tools

- Maven configuration written in XML continuous being big and cumbersome. On bigger projects it can have hundreds of lines of codes without actually doing anything “extraordinary”.
- Main benefit from Maven is its Life-cycle.
- As long as the project is based on certain standards with Maven one can pass through the whole life-cycle with relative ease. This comes at a cost of flexibility.

Gradle

- Gradle combines good parts of both tools and builds on top of them with DSL and other improvement.
- It has Ant's power and flexibility with Maven's life-cycle and ease of use.
- This release in 2012 and gained a lot of attention in a short period of times. For ex. Google adopted Gradle as the default build tool for the Android OS.

Hibernate

Hibernate framework – An Overview

Hibernate association and collection mapping

Hibernate object lifecycle

Hibernate transaction management

Hibernate querying

Hibernate HQL

Java Persistence API

Hibernate

Application is made up of layers or tiers

Each layer encapsulates specific responsibilities

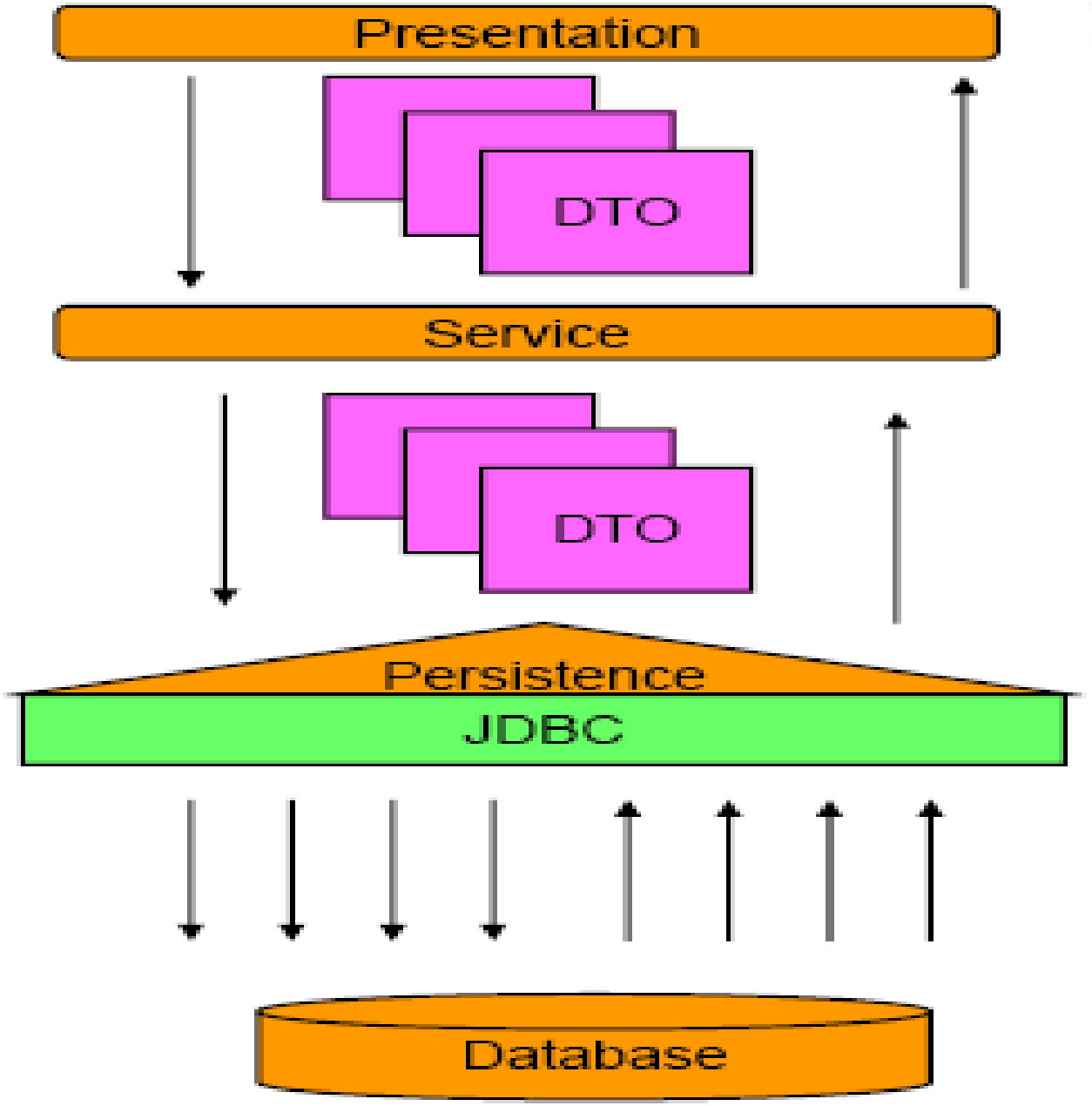
Enables changes in one area with minimal impact to other areas of the application

Hibernate

Common tiers

- Presentation
  - ‘View’ in model-view-controller
  - Responsible for displaying data only. No business logic
- Service
  - Responsible for business logic
- Persistence
  - Responsible for storing/retrieving data

Hibernate



Hibernate

Data Access Object

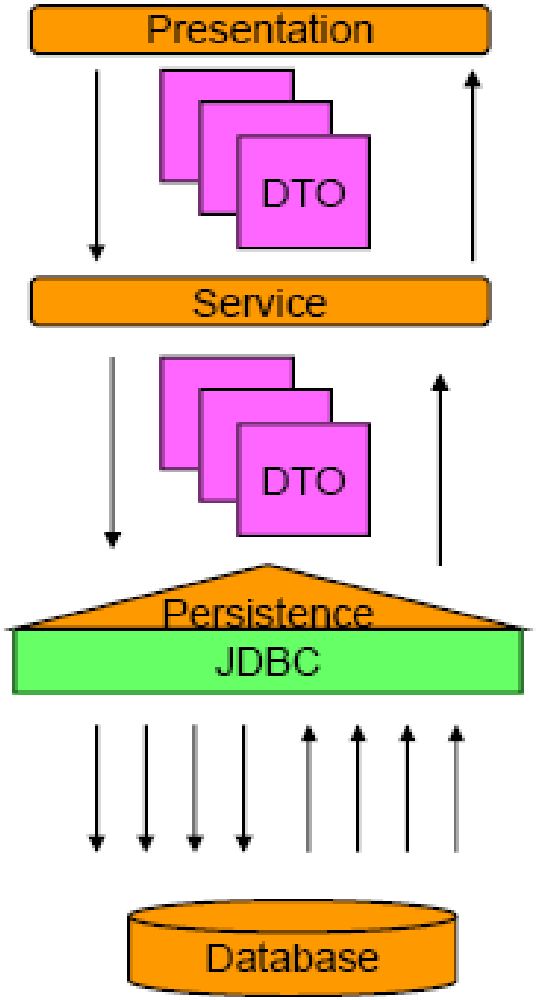
- Abstracts CRUD (Create, Retrieve, Update, Delete) operations

Benefits

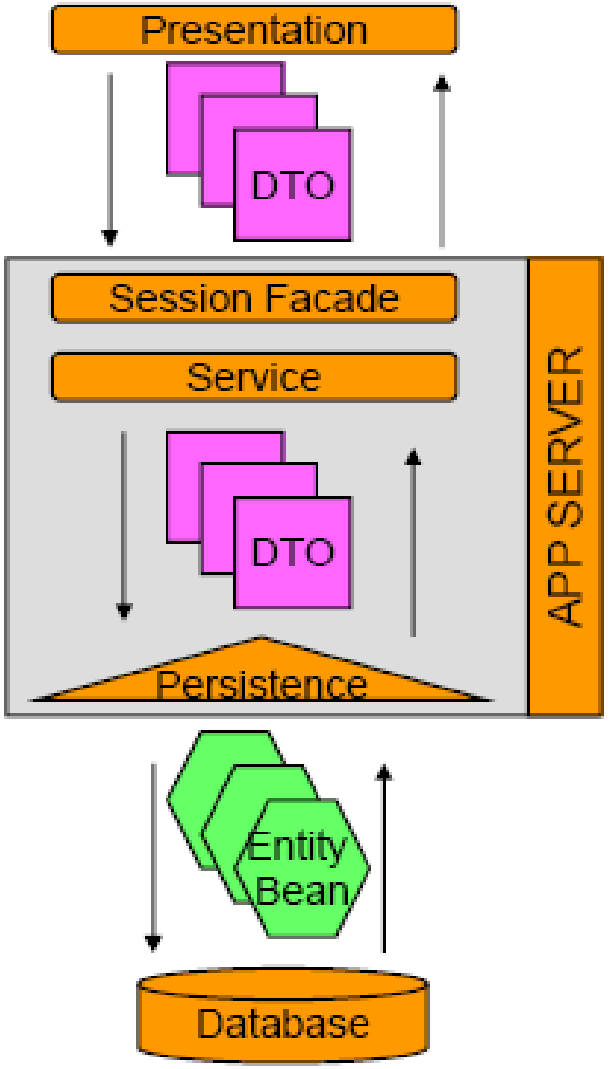
- Allows different storage implementations to be ‘plugged in’ with minimal impact to the rest of the system
- Decouples persistence layer
- Encourages and supports code reuse

Hibernate

Persistence with JDBC



Persistence with EJB 2.x



Hibernate

JDBC API provides ability to

- Establish connection to a database
- Execute SQL statements
- Create parameterized queries
- Iterate through results
- Manage database transactions

Basic Steps to JDBC Operations

Load driver or obtain datasource
Establish connection using a JDBC URL
Create statement
Execute statement
Optionally, process results in result set
Close database resources
Optionally, commit/rollback transaction

Hibernate

```
public Account createAccount(Account account) {
Connection connection = null;
PreparedStatement getAccountIdStatement = null;
PreparedStatement createAccountStatement = null;
ResultSet resultSet = null;
long accountId=0;
// Load driver
try {
Class.forName("oracle.jdbc.driver.OracleDriver");
catch (Exception e) {
throw new RuntimeException(e);
}
try {
//Get connection and set auto commit to false
Connection connection =
DriverManager.getConnection("jdbc:oracle:
thin:lecture1/lecture1@localhost:1521:XE");
connection.setAutoCommit(false);
...
```

Hibernate

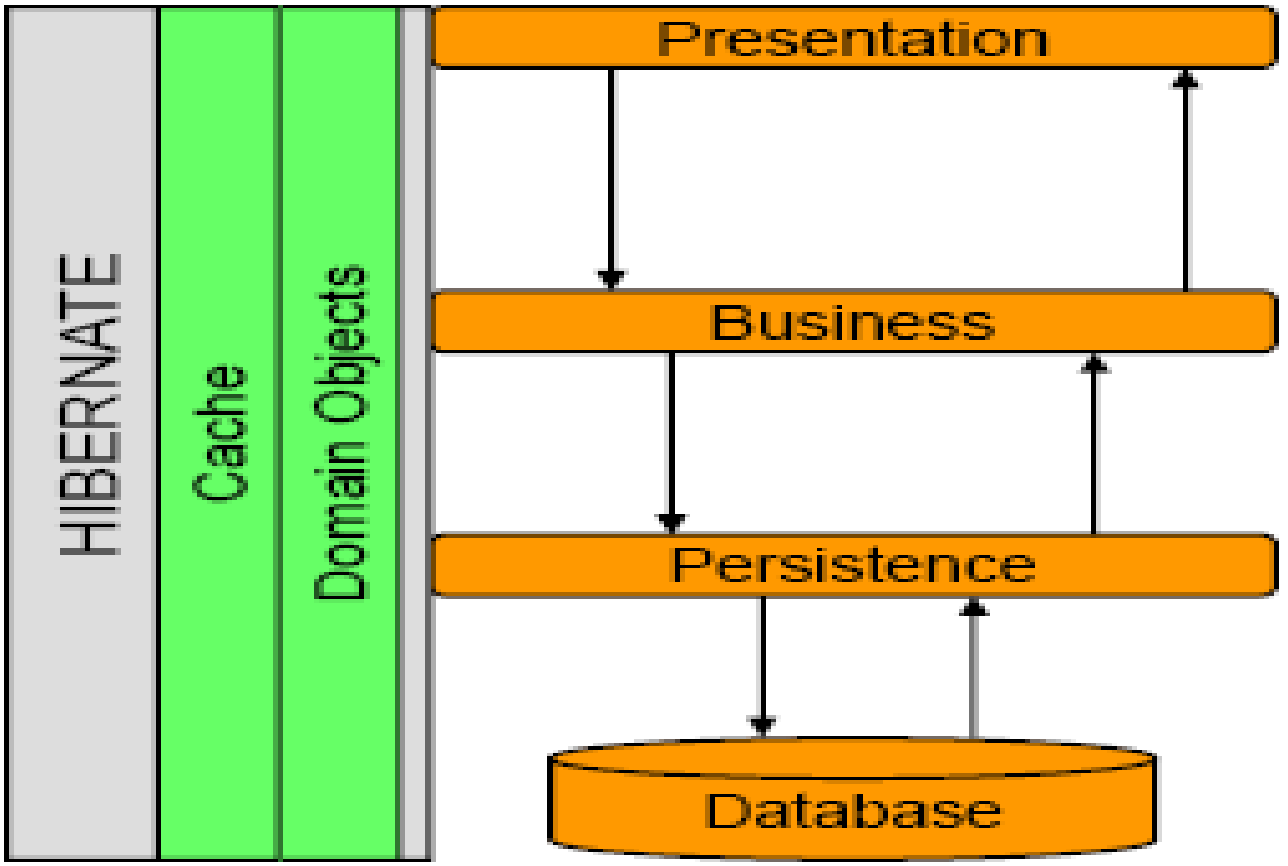
```
//Get account id from sequence
getAccountIdStatement = connection
.prepareStatement("SELECT
ACCOUNT_ID_SEQ.NEXTVAL
FROM DUAL");
resultSet = getAccountIdStatement.executeQuery();
resultSet.next();
accountId = resultSet.getLong(1);
//Create the account
createAccountStatement = connection
.
prepareStatement(AccountDAOConstants.CREATE
_ACCOUNT);
createAccountStatement.setLong(1, accountId);
createAccountStatement.setString(2,
account.getAccountType());
createAccountStatement.setDouble(3,
account.getBalance());
createAccountStatement.executeUpdate();
//Commit transaction
connection.commit();
}
```

Hibernate


```
catch (SQLException e) {
//In case of exception, rollback
try{
connection.rollback();
}catch(SQLException e1){// log error}
throw new RuntimeException(e);
}
finally {
//close database resources
try {
if (resultSet != null)
resultSet.close();
if (getAccountIdStatement!= null)
getAccountIdStatement.close();
if (createAccountStatement!= null)
createAccountStatement.close();
if (connection != null)
connection.close();
} catch (SQLException e) {// log error}
}
}
```

Hibernate

Persistence with Hibernate



Hibernate

- Grass roots development (2001)
  - Christian Bauer
  - Gavin King
- JBoss later hired lead Hibernate developers (2003)
  - Brought Hibernate under the Java EE specification
  - Later officially adopted as the official EJB3.0 persistence
- implementation for the JBoss application server.
- EJB 3.0 Expert Group (2004)
  - Key member which helped shape EJB3.0 and JPA
- NHibernate
  - .NET version release in 2005

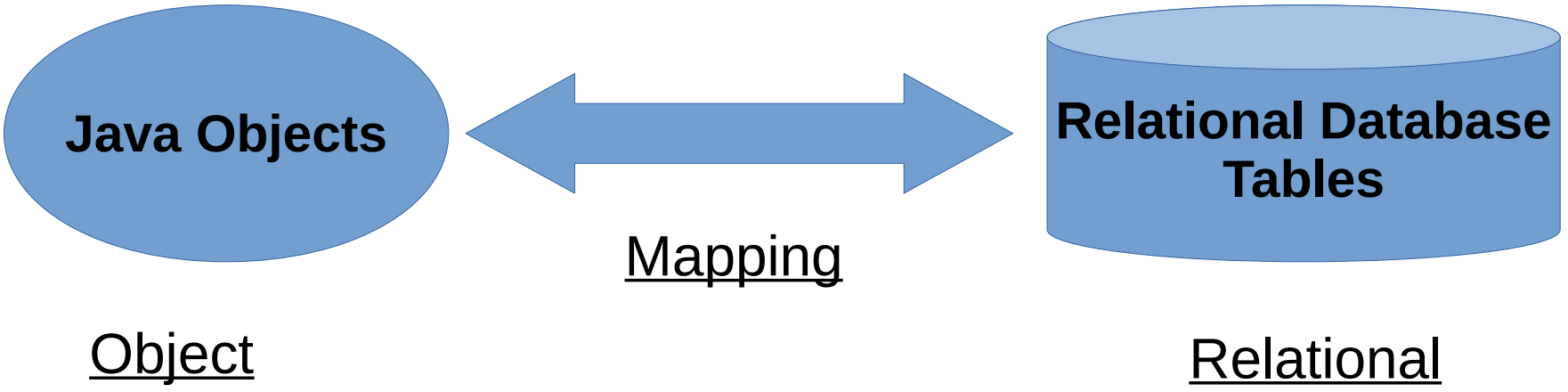
Hibernate

- Impedance mismatch
  – Object-oriented vs. relational
- Failure of EJB 2.x
  – Entity Beans were extremely slow, complex
- Reduce application code by 30%
  - Less code is better maintainable

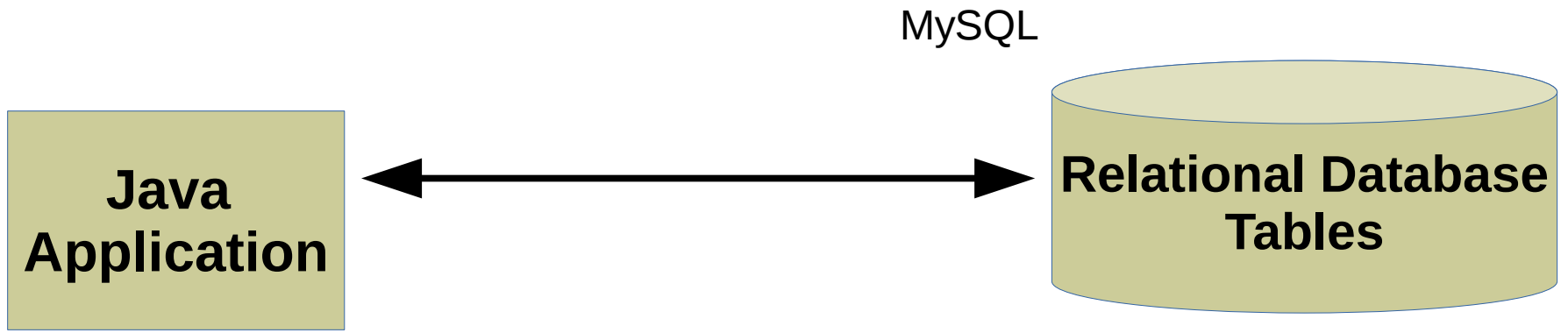
Hibernate

- Java developers are not database developers
  - Reduce the need for developers to know and fully understand database design, SQL, performance tuning
  - Increase portability across database vendors
- Increase performance by deferring to experts
  - Potential decrease in database calls
  - More efficient SQL statements
  - Hibernate cache usage

Hibernate


Java Objects
Relational Database Tables
Mapping
Object
Relational

Hibernate



Student class
Name
Roll_Num
Address
Mobile

Name	Roll_num	Address	Mobile

student_info_table

Student student = new Student(“Tom”,”#223”,”Australia”,”3345678”);

Hibernate

Student student =
new Student(“Tom”,”#223”,”Australia”,”3345678”);

```
Try{
//Register JDBC driver
Class.forName(“com.mysql.jdbc.Driver”);
//Open a Connection
Conn = DriverManager.getConnection(“jdbc:mysql://localhost:3306/<db-name>”);
//write code to map Java Object Student to the STUDENT_INFO table
stmt=conn.createStatement();
String sql = “INSERT INTO STUDENT_INFO(Name,Roll_Num,Address,Mobile)”
+ “VALUES(student.getName(),student.getRollnum(),……..,student.getMobile)”;
Stmt.executeUpdate(sql);
}catch(SQLException  sql){


}catch(Exception  exe){

}finally{
//Close connection
}
```

Hibernate

Student student =
new Student(“Tom”,”#223”,”Australia”,”3345678”);

```
Session  session = factory.openSession();
Transaction tx = null;

Try{
  Tx = session.beginTransaction();
  Session.save(student);
  tx.commit();

}catch(Hibernate h){
   if(tx != null)
   tx.rollBack();
  h.printStackTrace();
}
```

Hibernate

```
ResultSet  rs = stmt.executeQuery("SELECT
Name,Roll_Num,Address,Mobile FROM Student_info");

        //Fetch each row from the resultset
        //and create a list object containing student_info

List studentinfolist = new LinkedList();

while(rs.next){

         String name = rs.getString("name");
}
```

In Hibernate

List studentlist = session.createQuery("FROM student_info").list();

Hibernate

- ✔A real good caching mechanism for faster retrieval of data.
- ✔Opening and closing of databases connection would no longer be a problem.
- ✔Your application will support almost all the relational databases.
- ✔A little modification is needed in your Java application for any change in databases.

Persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
  <persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="BookStore"
     transaction-type="JTA">
     <jta-data-source>java:/BookStore</jta-data-source>
     <properties>
      <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQLDialect" />
      <property name="hibernate.hbm2ddl.auto" value="create" />
     </properties>
    </persistence-unit>
  </persistence>
```

THANK YOU