HCL TALENTCARE

HCL TALENTCARE

Introduction to Hibernate

Prerequisite

HCL TALENTCARE

Knowledge of-

- Relational Model and SQL
- Object oriented concepts and implementation
- Java programming language

Course Objectives

- Introduction- Introduction to ORM & Hibernate
- Hibernate Architecture and Framework
- Annotations
- Hibernate Instance States ,Lifecycle Operation
- Hibernate Configurations
- Criteria Query API, HQL & native SQL

Introduction

- What is Persistence?

  It is a process of storing the data to some permanent medium and retrieving it back at any point of time even after the application that had created the data ended.

- Persistence is the major challenge for any enterprise application.

Current Persistence Options

HCL TALENTCARE

- Serialization
- JDBC
- Entity Beans (EJB 2.x)
- DAO – Supporting technology for Hibernate
- ORM Tools

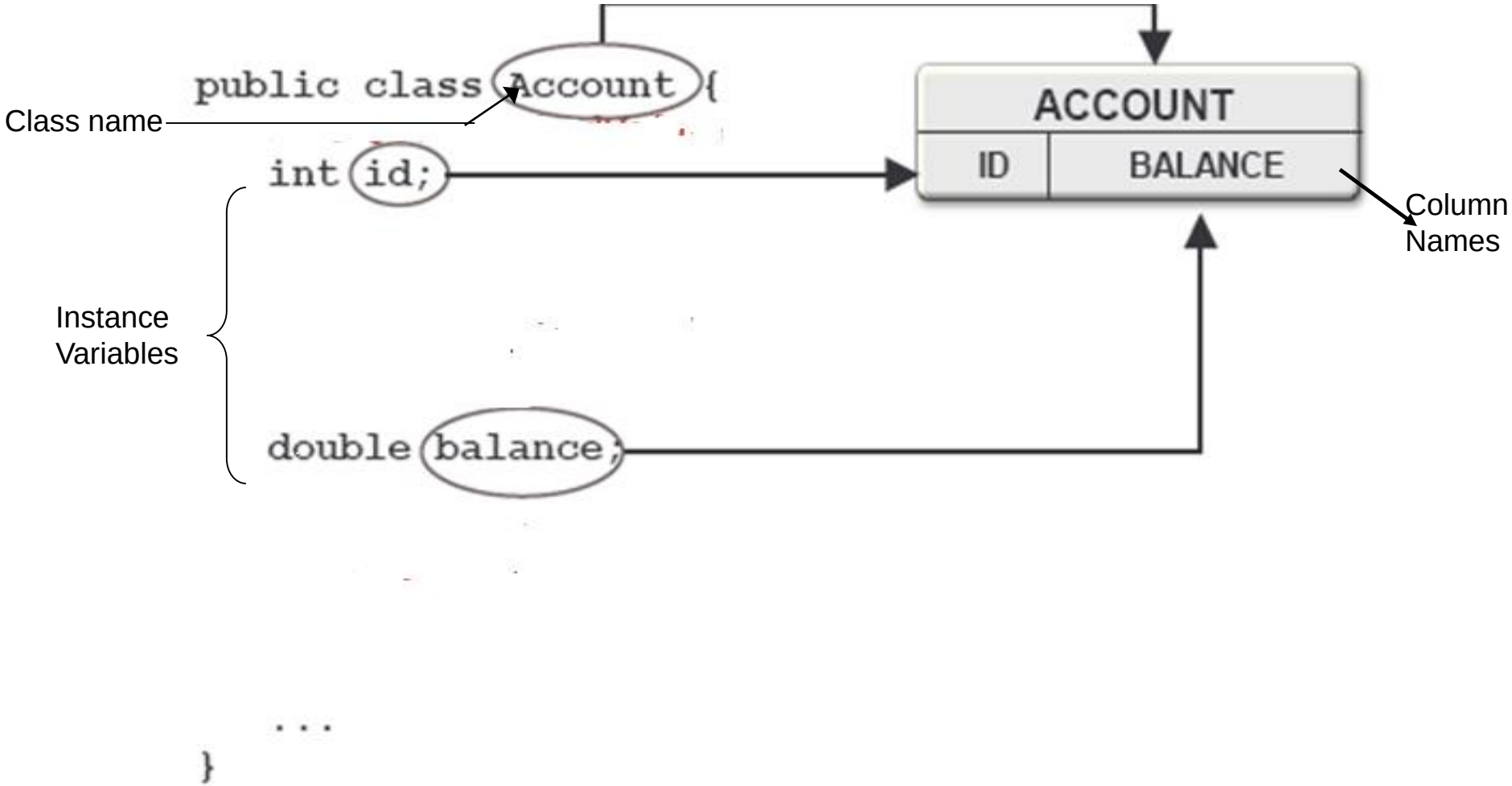
  Hibernate , Oracle Toplink etc.

Introduction to ORM

HCL TALENTCARE

- Object-relational mapping (ORM) is a programming technique for mapping the software objects to the relational model where properties of a class are mapped to a column in a table, class or an entity is mapped to table and instance of a class is a new record in a table.
- This creates, in effect, a "virtual object database" that can be used from within the programming language.
- There are both free and commercial packages available that perform object-relational mapping
- Java Entities are mapped to tables, instances are mapped to rows and attributes of instances are mapped to columns of table.

ORM Mapping.


public class Account {
int id;
double balance;
...
}
Class name
Instance Variables
ACCOUNT
ID
BALANCE
Column Names

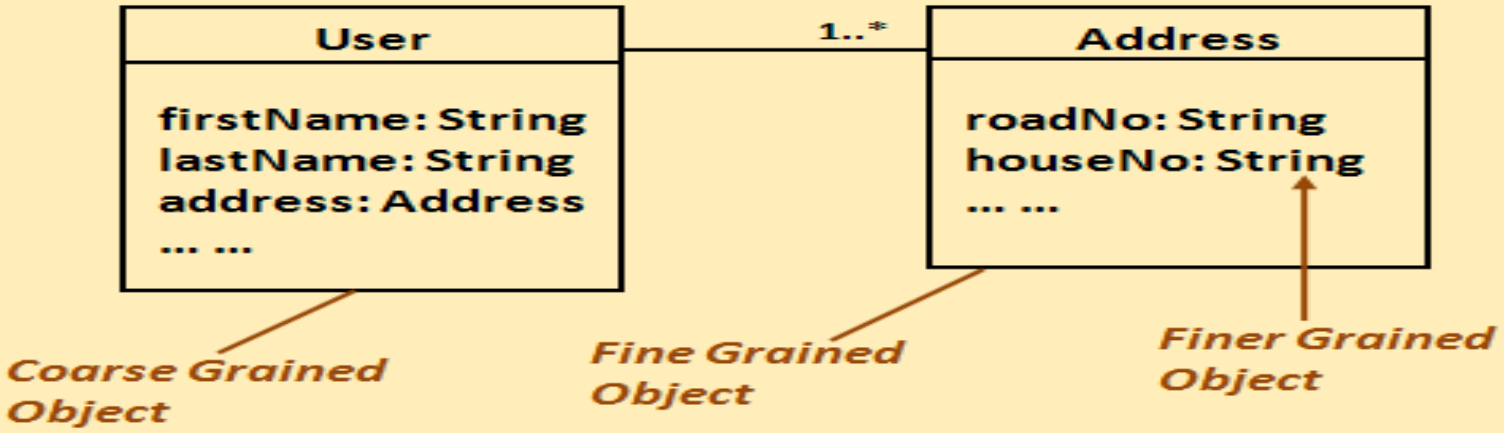
Paradigm Mismatch

HCL TALENTCARE

- There is lot of mismatch between the relational technology and Object Oriented Technology. This mismatch is also known as **Object relational mismatch**.
- ORM tries to solve this mismatch.
- Below are the list of mismatch between Object and Relational world
  - Problem of Granularity
  - Problem of Subtype
  - Problem of Identity
  - Problem of Association
  - Problem of Data Navigation

Paradigm Mismatch

- **Problem of Granularity:**
  - Java objects can have several levels of granularity. E.g. consider an association between classes like **User** and **Address**.



The granularity problem comes when the number of classes mapping to number of tables in the database do not match. For example let's say we have the User class which has an Address object

Paradigm Mismatch [contd…]

HCL TALENTCARE

- Also the table structure for User is Table USER:
  - FNAME
  - LNAME
  - Road No
  - House No

- There is one table but the data is sitting in two objects. The same problem can come the other way round also where you have two tables and one class containing all the data points. ORM frameworks has to care of this mismatch in terms of different number of tables mapped to different number of classes.

Paradigm Mismatch [contd…..]

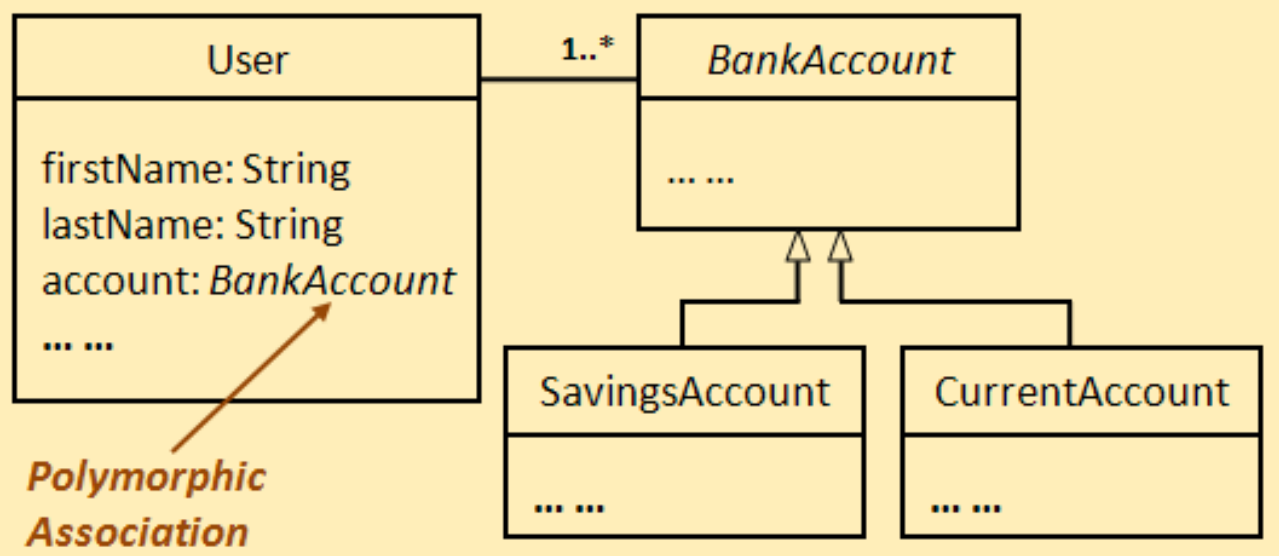
HCL TALENTCARE

- **Problem of Subtype:**
  - Java objects implement inheritance for defining super-type/ sub-type relationship.
  - Each sub or super class define different data and different functionality.
  - There is no support of inheritance in SQL.
- **Problem of Identity:**
  - Java provides two methods for checking equality of objects.
    - The == operator checks Object identity.
    - The equals() method checks object state equality (equality of value).
  - In SQL, equality of two records are determined by the primary key values.
  - Two or more objects can represent same row of data.

Paradigm Mismatch.. [contd…..]

HCL TALENTCARE

- **Problem of Association:**
  - Associations define relation between entities.
  - In Java we can have polymorphic associations represented by references where in relational world it is represented by foreign keys.
  - Table association is only one to many or one to one



Paradigm Mismatch.. [contd…..]

HCL TALENTCARE

- **Problem of Data Navigation:**
  - Difference in the way objects are accessed in Java and SQL.
  - In Java, we can have something like **currentUser.getAddress().getZipCode()** to access the zip code of the user.
  - In SQL, it can be done through SQL joins which are less efficient and cumbersome.

Why ORM ?

HCL TALENTCARE

- Shields developers from ‘Messy’ SQL.
- The business logic and domain are normally represented as an object model.
- The developer should concentrate and work with the object model.
- Tells how to cleanly connect Java objects to RDBMS tables

Introduction to Hibernate

HCL TALENTCARE

- Hibernate is a persistence framework
- Object relational mapping tool for java environment
- Part of JBoss Enterprise Middleware System (JEMS) suite of products.(JBoss, a division of Red Hat)
- Mapping of data representation from object model to a relational model with a SQL based schema
- Useful for object oriented domain model

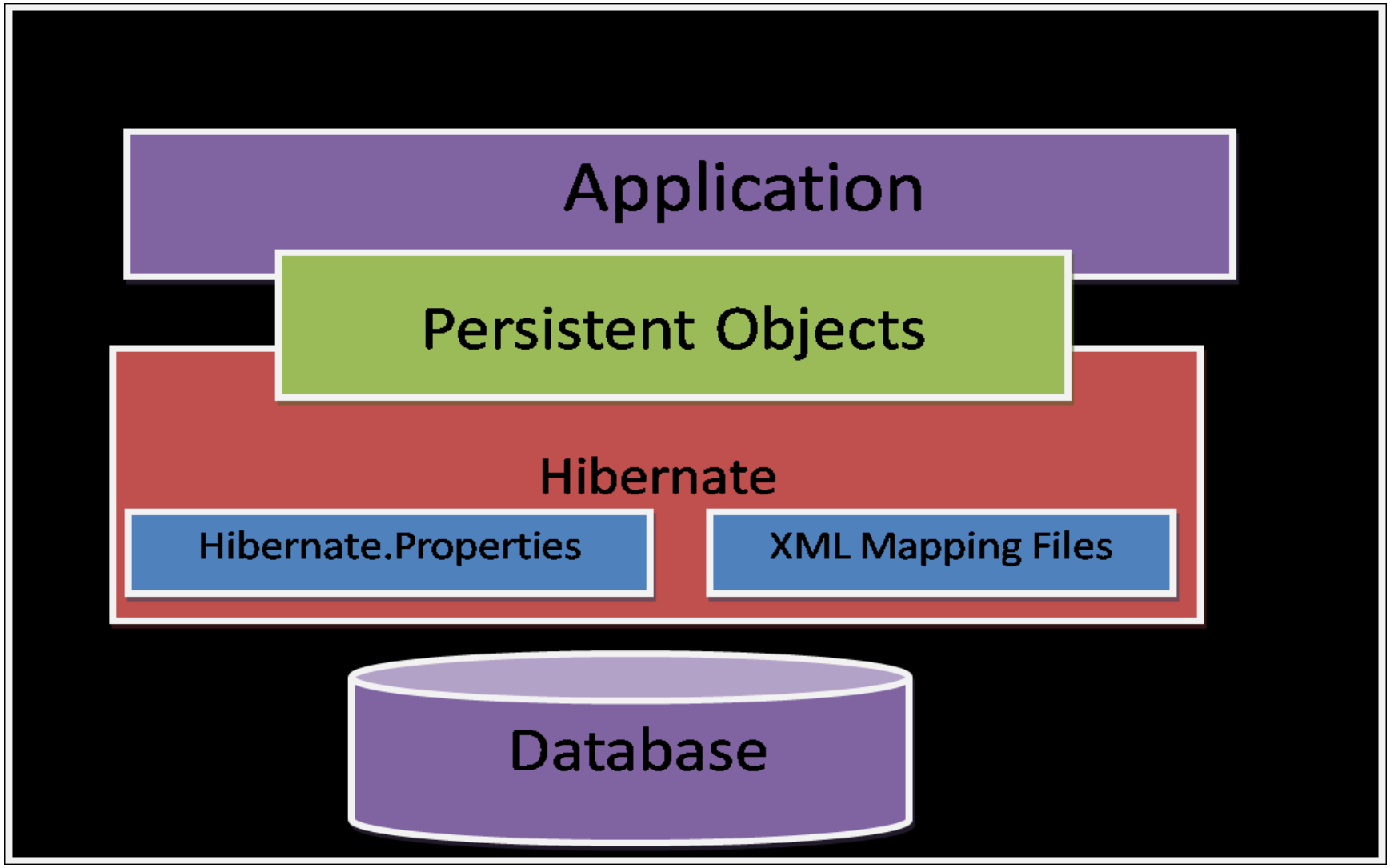
Hibernate

- Allows developers to focus on domain object modeling not the persistence plumbing
- Automated persistence of objects to relational table
- Metadata for describing mapping between objects and database
- Sophisticated query facilities like criteria API, Query by example(QBE) , Hibernate query language(HQL)
- High performance Object caching
- Vendor independence – Support for all type of relational database

Hibernate Distribution

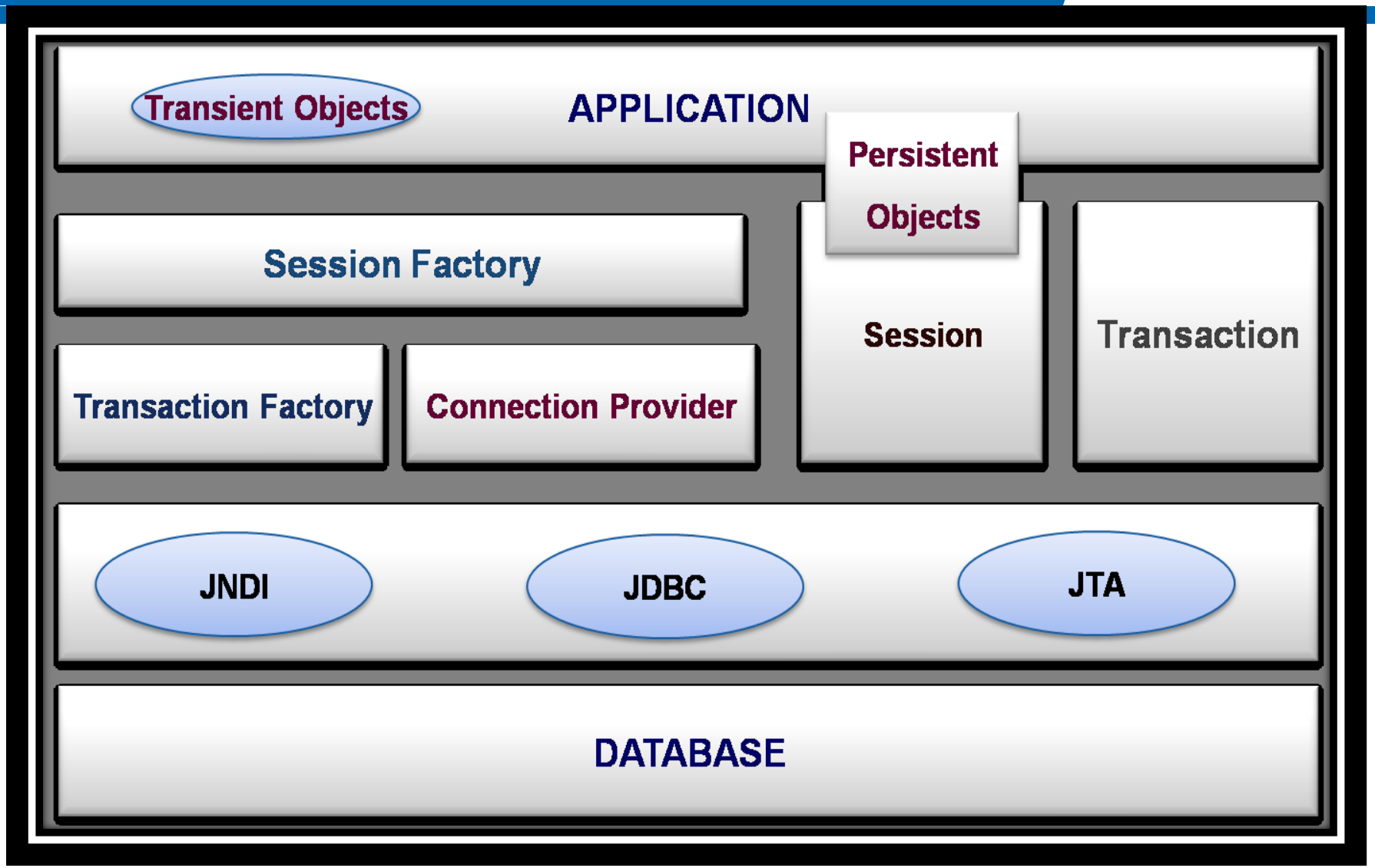
- Hibernate Core
  - Hibernate 3.2.x base service
  - Criteria Query and HQL support
  - Mapping metadata via XML mapping files
- Hibernate Annotations
  - Use of jdk 5.0 metadata
  - Reduced Code
- Hibernate EntityManager
  - Provides JPA compatability

Hibernate Architecture



Hibernate Architecture – A Detail look

HCL TALENTCARE



Hibernate Framework Objects -1

HCL TALENTCARE

- **SessionFactory**
  - Found in org.hibernate package
  - A n Object from which session objects are created
  - A client of Connection Provider
  - one for each database
  - May hold second level cache(optional) which can be reused between transactions at a process or

    cluster-level

Hibernate Framework Objects-2

HCL TALENTCARE

- **Session**
  - Found in org.hibernate package
  - A single threaded, short lived object representing a conversation between the application and persistence store
  - Wraps JDBC connection
  - Factory of Transaction

Hibernate Framework Objects-2 [contd…..]

HCL TALENTCARE

- Holds a mandatory first level cache of persistent objects, used when navigating the object graph or looking up objects by identifier
- The life of a Session is bounded by the beginning

  and end of a logical transaction.
- represents a persistence context
- Handles life-cycle operations of persistent Objects - insert, read ,delete ,update

Hibernate Framework Objects-3 [contd…..]

HCL TALENTCARE

- **Transaction**
  - Found in org.hibernate package
  - Its a single threaded, short lived object used by the application to specify an atomic unit of work
  - It abstracts application from the underlying JDBC , JTA or CORBA transaction
  - A session might span several transactions in some cases
  - Transaction demarcation is never optional while using the underlying API or Transaction

Hibernate Framework Objects-3 [contd…..]

HCL TALENTCARE

- **ConnectionProvider –**
  - Found in org.hibernate.connection package
  - A factory for(and pool of) JDBC connections
  - Abstracts the application from the underlying DataSource or DriverManager
  - Not exposed to application but can be extended or implemented by the developer

Hibernate Framework Objects-3 [contd…..]

HCL TALENTCARE

- **TransactionFactory**
  - Found in org.hibernate package
  - A factory for Tranasaction instances
  - Not exposed to application but can be extended/implemented by developer

Employee Bean

```
public class Employee {
        private int empId;
        private String name;

        public int getEmpId() {
            return empId; }
        public void setEmpId(int empId) {
                this. empId = empId; }
        public String getName() {
                return name; }
        public void setName(String name) {
                this.name = name; }
}
```

Employee Table

empId(PK)	name
101	Sara
102	Peter
103	Joe

employee.hbm.xml

HCL TALENTCARE

```
<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.demo.sample">
<class name="Employee" table="EMPLOYEE_TABLE">
<id name="empId" type="int" column="EMP_ID" >
<generator class="native"/>
</id>
<property name="empName" type="java.lang.String"
column="EMP_NAME"/>

</class>
</hibernate-mapping>
```

Employee Entity

HCL TALENTCARE

```
@Entity    //  To make Employee as an Entity
public class Employee {
        @Id  // To make empId as primary key
        private int empId;
        private String name;
        public int getEmpId() {
            return empId; }
        public void setEmpId(int empId) {
                this. empId = empId; }
        public String getName() {
                return name; }
        public void setName(String name) {
                this.name = name; }
}
```

Employee Table

empId(PK)	name
101	Sara
102	Peter
103	Joe

Mapping to a Table

HCL TALENTCARE

- @Entity and @Id annotations need to be specified to map an entity to a database table.
- By default, table name will be the name of the entity class without the package name.
- @Table annotation will be used to specify the name of the table if the names are different.
- *Overriding the default table name*

```
@Entity
@Table(name="EMP")
public class Employee { ... }
```

EMP Table

empId(PK)	name
101	Sara
102	Peter
103	Sam

Column Mappings

HCL TALENTCARE

- @Column annotation will be used to specify the name of the column if entity property name is different from column name in the table

```
@Entity
public class Employee {
   @Id
   @Column(name="EMP_ID")
   private int id;

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

      // ...getter and setter methods

}
```

Employee Table

EMP_ID	EMP_NAME
101	Sara
102	Peter
103	Sam

Temporal Types

- Temporal types are the set of time-based types
- To persist Temporal data types of java, @Temporal annotation is used
- The list of supported temporal types used in Entity
  - java.util.Date and java.util.Calendar

Mapping Temporal Types

HCL TALENTCARE

```
@Entity
public class Employee {
    @Id
    private int id;

    @Temporal(TemporalType.DATE)
    private Calendar dob;

   @Temporal(TemporalType.TIME)
    private Date startDate;
          // ...
}
```

Different Temporal Types from JPA

In Entity	Mapped To Database
TemporalType.DATE	Mapped to java.sql.Date
TemporalType.TIME	Mapped to java.sql.Time
TemporalType.TIMESTAMP	Mapped to java.sql.Timestamp

Transient State

Attributes that are part of an entity but not required to be persisted should be prefixed with the @Transient annotation

```
@Entity
public class Employee {
    @Id
    private int empid;
    private String name;
    private long salary;
   @Transient
   private String translatedName;
   //  getter and setter method
}
```

Employee Table

empid	name	salary
100	Sara	1000
101	Joe	2000

translatedName will not be persisted

Hibernate Configurations

Hibernate Configuration class has two key component operations: database connection and class mapping setup

Hibernate provides following types of configuration

- hibernate.properties – A standard java .properties file
- hibernate.cfg.xml – An xml file
- Programatic configuration – in .java file

Configuring using hibernate.cfg.xml

HCL TALENTCARE

- To configure through xml based configuration, create an xml file hibernate.cfg.xml. This file needs to be placed in root of application's classpath.

- This file must contain Hibernate 3 configuration DTD, available at "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

JDBC Properties

- hibernate.connection.driver_class → jdbc driver class
- hibernate.connection.url → jdbc URL
- hibernate.connection.username → database user name
- hibernate.connection.password → database user password
- hibernate.connection.pool_size → maximum number of pooled connections

DataSource properties

- hibernate.connection.datasource →datasource JNDI name
- hibernate.jndi.url → Provides the URL for JNDI provider (optional)
- hibernate.jndi.class →Provides the initial context for JNDI (optional)
- hibernate.connection.username → database user name (optional)
- hibernate.connection.password → database user password (optional)

Optional Configuration Properties

HCL TALENTCARE

- hibernate.dialect →The classname of a Hibernate Dialect to be used. The SQL dialect allows Hibernate to generate SQL optimized statements for a particular relational database.
- hibernate.show_sql → Writes all SQL statements to console.
- hibernate.connection.provider_class → The classname that implements Hibernate's ConnectionProvider interface
- hibernate.cache.provider_class → Specifies the classname that provides CacheProvider interface.

Optional Configuration Properties

- hibernate.hbm2ddl.auto→Automatically creates, updates or drops the database schema on startup or shutdown(update | create | create-drop)

- hibernate.cache.use_second_level_cache: Determines whether to use second level of cache. It accepts values true or false.

A sample XML based configuration file

HCL TALENTCARE

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">

<hibernate-configuration>

<session-factory>

<!-- Database connection settings -->

<property
name="connection.driver_class">com.mysql.jdbc.Driver</property>
<property
name="connection.url">jdbc:mysql://localhost:3306/hibernatedb</prop
erty>
<property name="connection.username">root</property>
<property name="connection.password">password</property>
```

A sample XML based configuration file

HCL TALENTCARE

```
<!-- JDBC connection pool (use the built-in) -->
<property name="connection.pool_size">1</property>

<!-- SQL dialect -->
<property name="dialect">
org.hibernate.dialect.MySQLDialect
</property>

<!-- Enable Hibernate's automatic session context management -->
<property name="current_session_context_class">thread</property>

<!-- Disable the second-level cache -->
<property
name="cache.provider_class">org.hibernate.cache.NoCacheProvider</pro
perty>

<!-- Echo all executed SQL to stdout -->
<property name="show_sql">true</property>
```

A sample XML based configuration file

HCL TALENTCARE

```
<!-- Drop and re-create the database schema on startup -->
<property name="hbm2ddl.auto">create</property>
<mapping class="com.hcl.entity.Employee" />

</session-factory>

</hibernate-configuration>
```

Programmatic Configuration

HCL TALENTCARE

- In programmatic configuration, the details such as JDBC connection and resource mapping are supplied using Configuration API
- For programmatic configuration an instance of org.hibernate.cfg.configuration is created
- org.hibernate.cfg.configuration is used to build an immutable org.hibernate.SessionFactory
- **Creating an Instance of Configuration**

```
Configuration cfg=new Configuration()
```

Programmatic Configuration (contd….)

HCL TALENTCARE

- **Adding mapping documents**

  cfg.addResource(“Person.hbm.xml”);

  cfg.addResource(“Event.hbm.xml”);

  OR

  cfg.addClass(“Person.class”);

  cfg.addClass(“Event.class”);

- **Setting Properties**

  cfg.setProperty(“hibernate.connection.driver_class”, “com.mysql.jdbc.Driver”);

  cfg.setProperty(“hibernate.connection.url”,” jdbc:mysql://localhost:3306/MyDB”);

Programmatic Configuration (contd….)

HCL TALENTCARE

- **Creating a SessionFactory**

```
SessionFactory sf=cfg.buildSessionFactory();
```

Configuring using hibernate.properties file

HCL TALENTCARE

- Hibernate configuration can also be configured hibernate.properties file
- This file has to be placed in root directory of application's classpath
- A sample hibernate.properties file

```
hibernate.connection.driver_class = com.mysql.jdbc.Driver
hibernate.connection.url = jdbc:mysql://localhost:3306/MyDB
hibernate.connection.username = root
hibernate.connection.password = password
```

Mapping Document

- Hibernate needs mapping document for all objects that needs to be persisted to the database
- The mapping document must be saved with .hbm.xml extension.

Primary Key Generation Scheme

HCL TALENTCARE

- Increment

  <generator class="increment"/>

  Generates identifier of type long, short or int that are unique when no other process is inserting data to the same table.

- Native

  The native class selects either identity, sequence or hilo depending upon the capabilities of the underlying database

- Others – hilo, seqhilo, sequence etc.

Primary Key Generation Scheme

HCL TALENTCARE

- To generate a primary key optional **<generator>** element is used . <generator> has an attribute class that has many options to generate a primary key in different situations.
- **<generator class=“ ”/>**
- The different options available for a class are
- Assigned
- **<generator class=“assigned”/>**

  An application assigns an identifier to object before save() is called. Default strategy when no generator specified

To Configure Primary Key Generation

HCL TALENTCARE

- **@GeneratedValue**
  - Annotation used to indicate JPA to generate values.
  - To specify generation strategy
  - Used along with **@Id** annotation

@GeneratedValue(strategy, generator)

strategy – used to specify different strategy type. AUTO is the default strategy.

generator - name of the generator(for SEQUENCE strategy type)

Using a Default Generation Strategy

```
@Entity
public class Account {
  @Id
  @GeneratedValue(
    strategy=GenerationType.AUTO)
 /*Indicates that the persistence should pick an
appropriate strategy for the particular
database.*/
private int accountNo;
private int customerId;
private String accountType;
//setter and getter methods
}
```

Example – Specifying User defined Sequence

HCL TALENTCARE

```
@Entity
 public class Account {
@Id
//Create Sequence
@SequenceGenerator(name="seqAccNo",sequenceName="DB_seqAccNo",
                    initialValue=1000,allocationSize=10)

@GeneratedValue(strategy=GenerationType.SEQUENCE
,generator="seqAccNo")

private int accountNo;
private int customerId;
private String accountType;
// setter and getter methods
}
```

Composite Key In Entity

A composite primary key can be declared using the annotations
@EmbeddedId

Example -

One customer can have more than one account.
One account can be shared by two customers. (Joint Account)

For the above described scenario, the combination of the accoundNo and the customerId forms the composite primary key.

Example Using @ EmbeddedId

HCL TALENTCARE

Entity Class

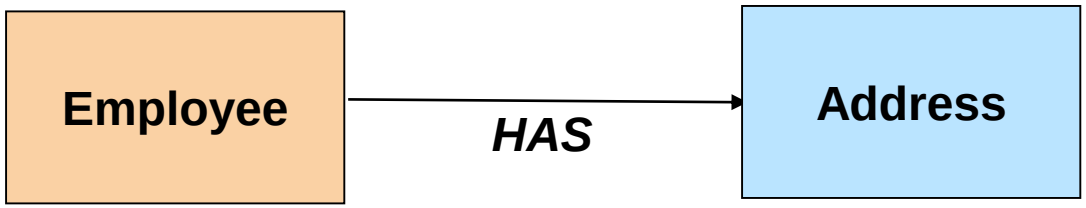
```
@Entity
public class Account {

/*designate two persistent fields
or properties as the entity's
primary key*/

@EmbeddedId
private AccountPK AccountID;

private String accountType;
private long currentAmount;
private String status;

//setter and getter methods
}
```

AccountPK.java

```
@Embeddable
public class AccountPK
{

        private int accountNo;
        private int customerId;

      //setter and getter methods
}
```

Entity Relationship

- Consider the Employee and Address entities

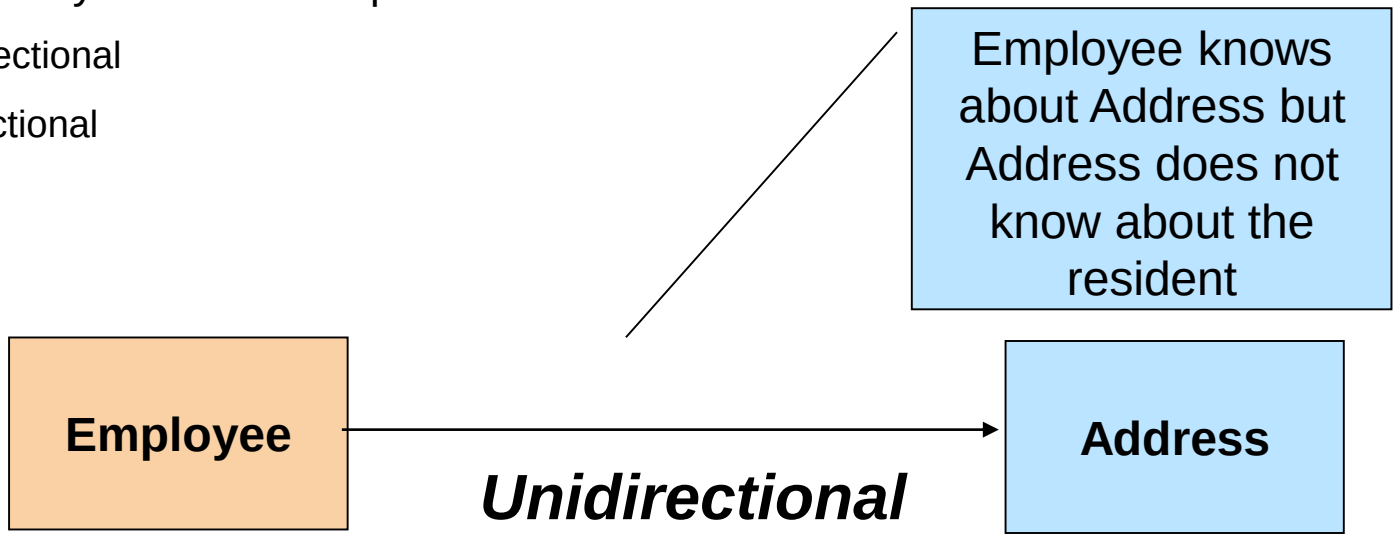


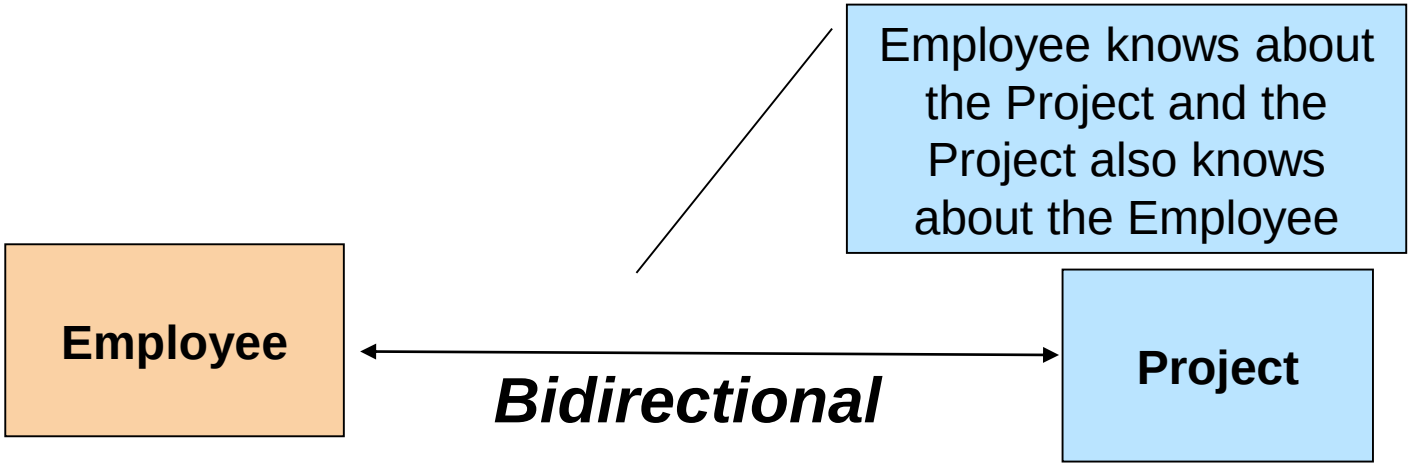

```
public class Employee{
 int empId;
 Address addres;
}
```

```
public class Address{
  int doorNo;
 String street;
}
```

Entity Relationship

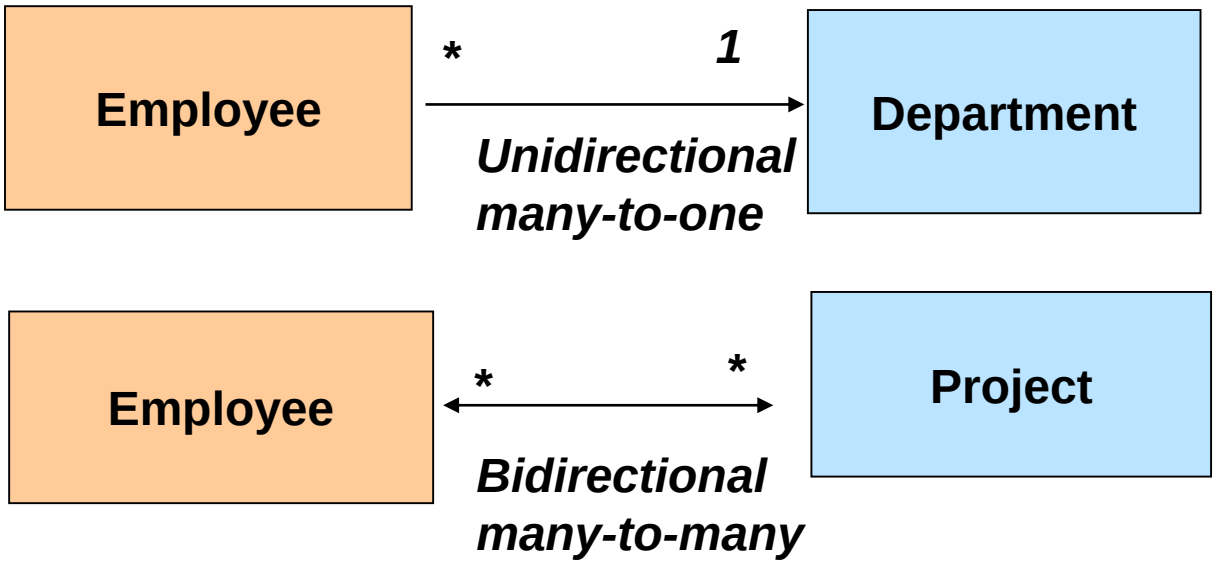
- Directionality of Relationships
  - Unidirectional
  - Bidirectional





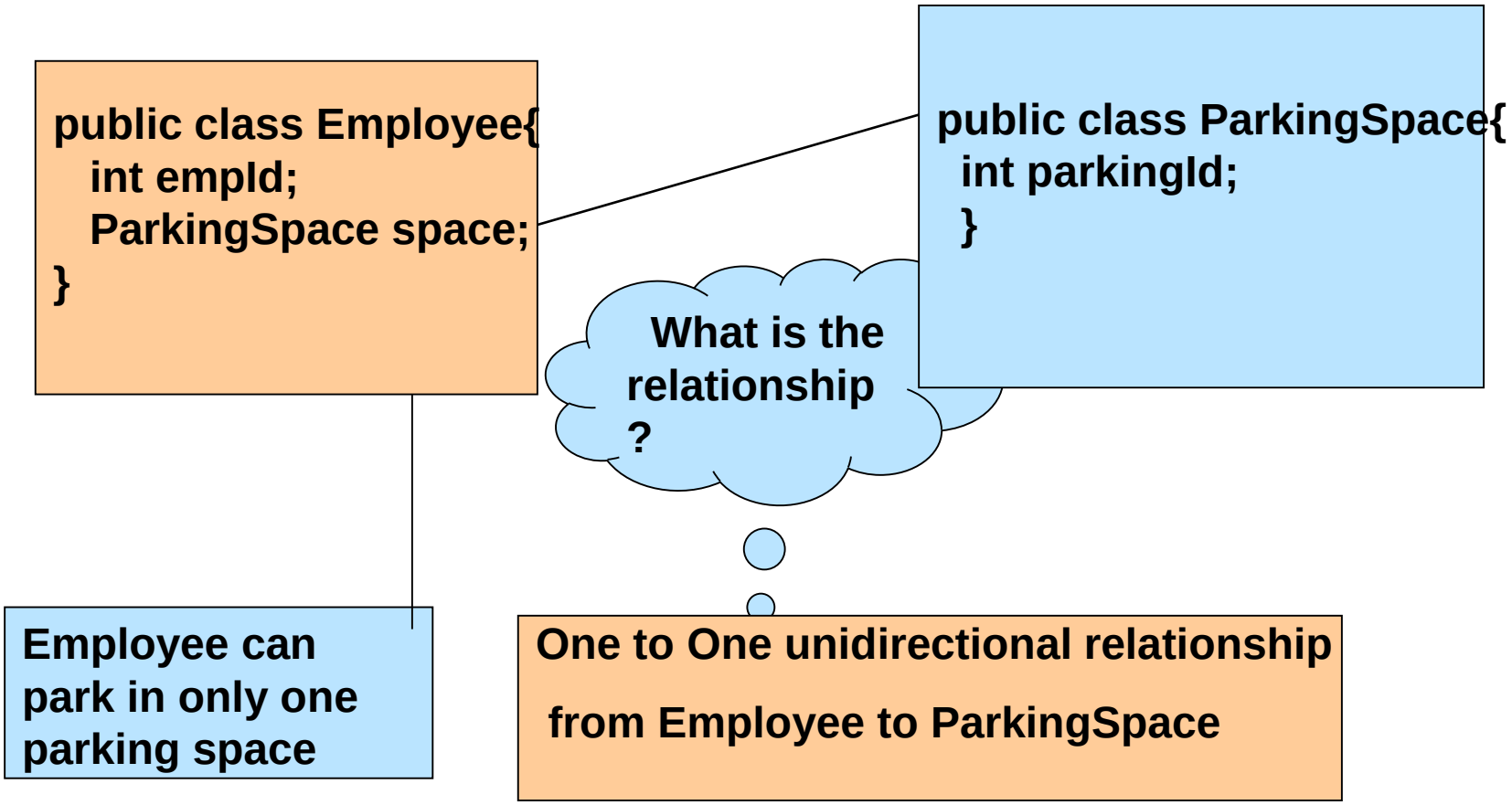
Entity Relationship

- Cardinality of Relationships
  - One to One
  - Many to One
  - One to Many
  - Many to Many

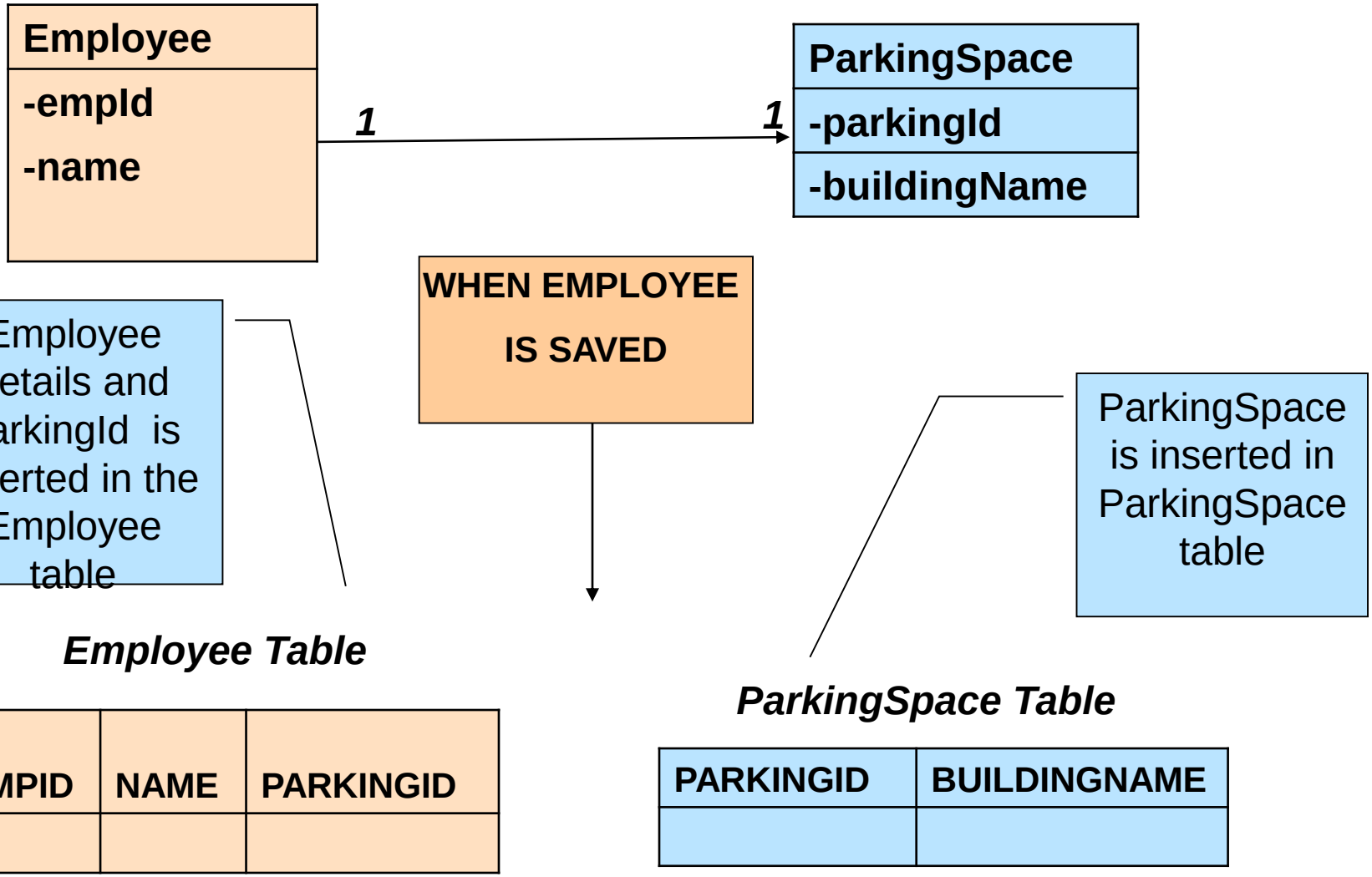


One to One Unidirectional

- Consider the Employee and ParkingSpace class



One to One Unidirectional



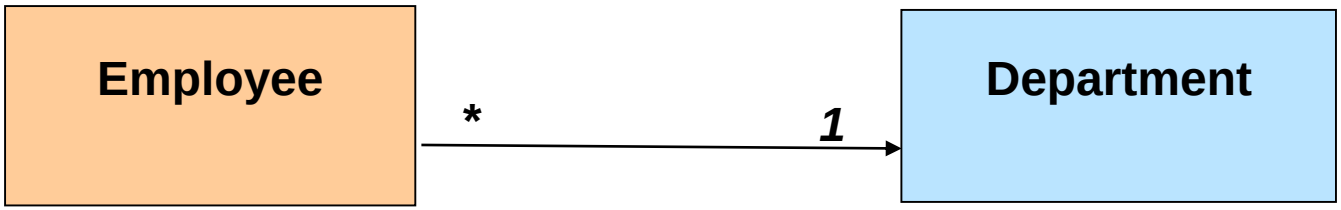
One to One Unidirectional

```
@Entity
public class Employee{
   @Id
   int empId;
   String name;

 @OneToOne(cascade=CascadeType.ALL,
 targetEntity=ParkingSpace.class)

@JoinColumn(name="parkingId",
                              unique=true)

     ParkingSpace space;
          //getter and setter
}
```

```
@Entity
public class ParkingSpace{

    @Id
    int parkingId;

    String buildName;

      //getter and setter methods
}
```

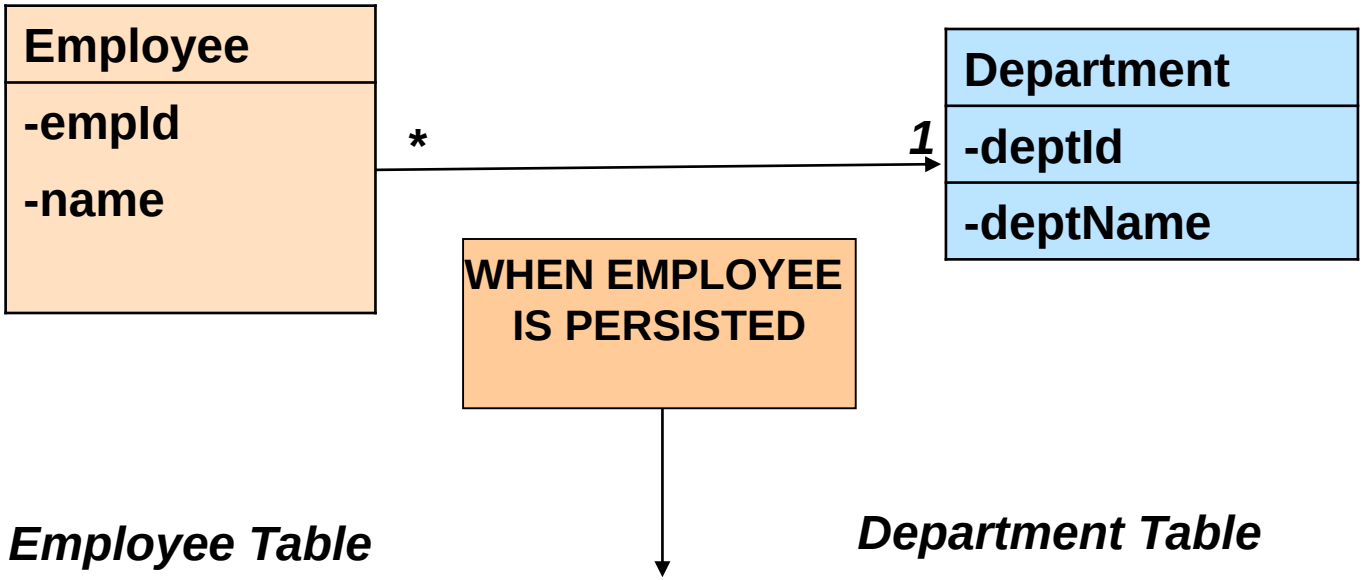
ManyToOne Unidirectional

- Consider the Employee and Department entities



Many Employees belongs to one Department

ManyToOne Unidirectional



Employee Table

EMPID	NAME	DEPTID
1001	John	101
1002	Rahul	101

Department Table

DEPTID	DEPTNAME
101	E&R

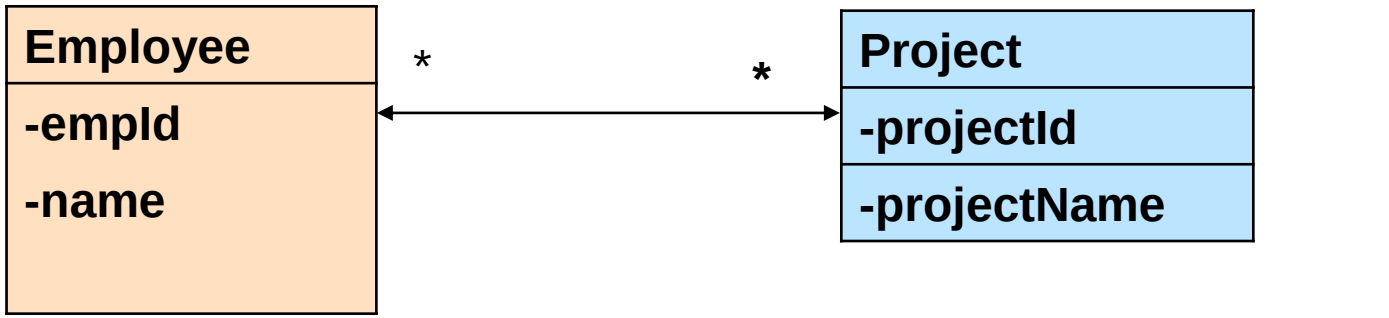
ManyToOne Unidirectional

```
@Entity
public class Employee{

  @Id
  int empId;
 String name;

   @ManyToOne(targetEntity=Department.class,
          cascade=CascadeType.ALL)
    @JoinColumn(name="deptId")

    Department dept;

      //getter and setter methods
}
```

```
@Entity
public class Department{

    @Id
    int deptId;

    String deptName;

       //getter and setter methods
}
```

Many To Many Unidirectional

- Consider the Employee and Project entities




```
public class Employee{
  int empId;
 List  project;
}
```

```
public class Project{
  int  projectId;
String projectName;

}
```

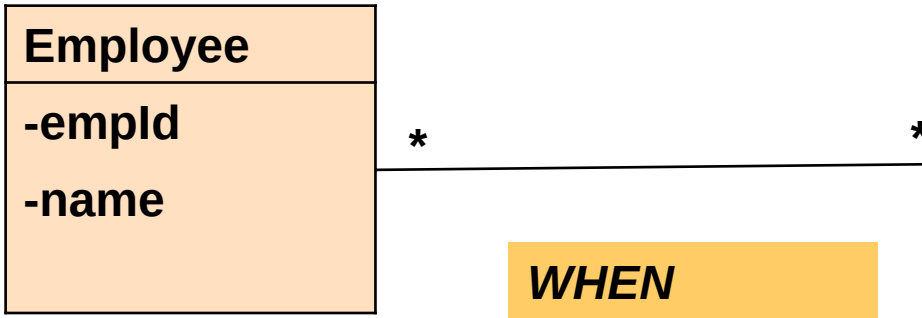
The relationship is Many To Many

Many To Many Unidirectional

```
@Entity
public class Employee{

 @Id
 int empId;
 String empName;

@ManyToMany(targetEntity=Project.class,
cascade=CascadeType.ALL)
@JoinTable(name="project_emp")

 Set project;

        //getter and setter
}
```

```
@Entity
public class Project

    @Id
    int projectId;

   String projectName;

        //getter and setter methods
}
```

Many To Many Unidirectional



WHEN EMPLOYEE IS PERSISTED

PROJECT TABLE

PROJECTTID	PROJECTNAME
101	*HandsOn Tool*
102	AttendenceTool

EMPLOYEE TABLE

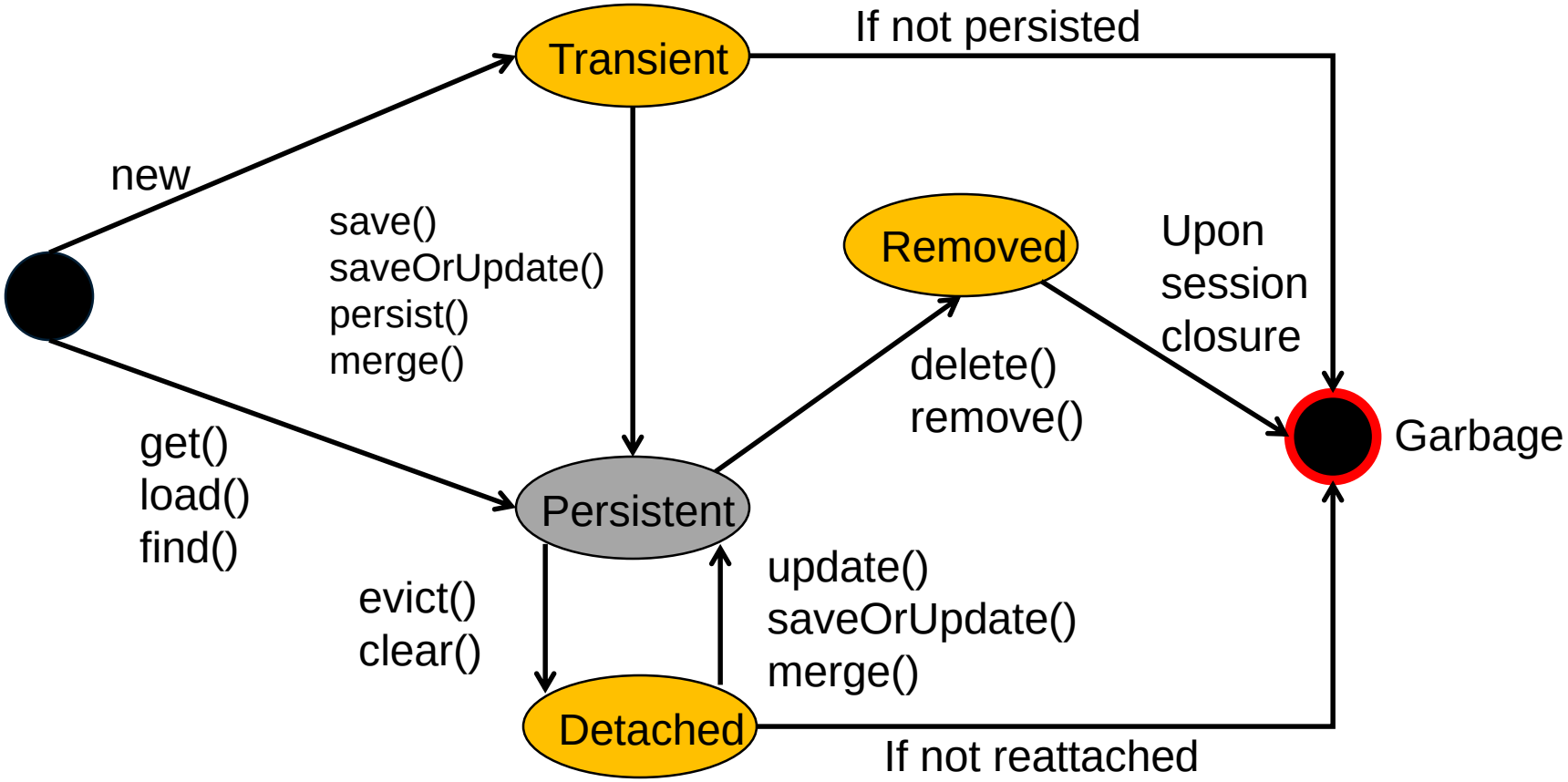
EMPID	NAME
1001	John
1002	Rahul

MYJOIN TABLE

PROJECT_PROJECTID	EMP_EMPID
101	1001
101	1002

New table created

Life Cycle Of Hibernate



Transient State

All objects start off in the transient state

– Account account = new Account();

• account is a transient object

• **Hibernate is not aware of the object instance**

• **Not related to database row**

– No value for accountId

• **Garbage collected when no longer referenced by any other objects**

Persistent State

HCL TALENTCARE

- **Hibernate is aware of, and managing, the object**
- **Has a database id**

– Already existing object retrieved from the database

– Formerly transient object about to be saved

- This is the only state where objects are saved to the database

– Modifications made in other states are NOT saved to the database while the object remains in that state

– Changes to objects in a persistent state are automatically saved to the database without invoking session persistence methods

- **Objects are made persistent through calls against the Hibernate session**

– session.save(account); – session.lock(account);

– session.update(account); – session.merge(account);

Persistent State

HCL TALENTCARE

```
Session session =
   SessionFactory.getCurrentSession();

// 'transient' state – Hibernate is NOT aware that it exists
Account account = new Account();

// transition to the 'persistent' state. Hibernate is NOW
// aware of the object and will save it to the database
session.saveOrUpdate(account);

// modification of the   object will automatically be
// saved because the object is in the 'persistent' state
account.setBalance(500);

// commit the transaction
session.getTransaction().commit();
```

Removed State

HCL TALENTCARE

- **A previously persistent object that is deleted from the database**

– session.delete(account);

- **Java instance may still exist, but it is ignored by Hibernate**

– Any changes made to the object are not saved to the database

– Picked up for garbage collection once it falls out of scope

- Hibernate does *not* null-out the in-memory object

Removed

HCL TALENTCARE

```
Session session = SessionFactory.getCurrentSession();

// retrieve account with id 1. account is returned in a 'persistent' state
Account account = session.get(Account.class, 1);

// transition to the 'removed' state. Hibernate deletes the
// database record, and no longer manages the object
session.delete(account);

// modification is ignored by Hibernate since it is in the 'removed' state
account.setBalance(500);

// commit the transaction
session.getTransaction().commit();

// notice the Java object is still alive, though deleted from the database.
// stays alive until developer sets to null, or goes out of scope
account.setBalance(1000);
```

Detached State

HCL TALENTCARE

- **A persistent object that is still referenced after closure of the active session**

– session.close() changes object's state from persisted to detached

- **Still represents a valid row in the database**
- **No longer managed by Hibernate**

– Changes made to detached objects are not saved to the database while object remains in the detached state

– Can be reattached, returning it to the persistent state and causing it to save its state to the database

- update();
- merge();
- lock(); // reattaches, but does not save state

Detached State

```
Session session1 = SessionFactory.getCurrentSession();

// retrieve account with id 1. account is returned in a 'persistent' state
Account account = session1.get(Account.class, 1);

// transition to the 'detached' state. Hibernate no longer manages the object
session1.close();

// modification is ignored by Hibernate since it is in the 'detached'
// state, but the account still represents a row in the database
account.setBalance(500);

// re-attach the object to an open session, returning it to the 'persistent'
// state and allowing its changes to be saved to the database
Session session2 = SessionFactory.getCurrentSession();
session2.update(account);

// commit the transaction
session2.getTransaction().commit();
```

Session API

Session methods do NOT save changes to

the database

– save();

– update();

– delete();

- **These methods actually SCHEDULE changes to be made to the database**
- **Hibernate collects SQL statements to be issued**
- **Statements are later *flushed* to the database**

– Once submitted, modifications to the database are not

permanent until a commit is issued

- session.getTransaction().commit();

Session API – Object Persistence

HCL TALENTCARE

- **session.save(Object o)**

– Schedules insert statements to create the new object in the database

- **session.update(Object o)**

– Schedules update statements to modify the existing object in the database

Session API Object Persistence

HCL TALENTCARE

- **session.saveOrUpdate(Object o)**

– Convenience method to determine if a ‘save’ or

‘update’ is required

- **session.merge(Object o)**

– Retrieves a fresh version of the object from the database and based on that, as well as modifications made to the object being passed in, schedules update statements to modify the existing object in the database.

Session API Object Retrieval

HCL TALENTCARE

- **session.get(Object.class, Identifier)**

– Retrieves an object instance, or null if not found

- **session.load(Object.class, Identifier)**

– Retrieves an object instance but does NOT result in a database call

» If ‘detached’, throws ObjectNotFoundException

Session API Object Retrieval

HCL TALENTCARE

- **session.lock(Object, LockMode)**

– Reattaches a detached object to a session without scheduling an update

– Also used to ‘lock’ records in the database

- **session.refresh(Object)**

– Gets the latest version from the database

Session API other Methods

HCL TALENTCARE

- **session.delete(Object)**

– Schedule an object for removal from the database

- **session.evict(Object)**

– Removes individual instances from persistence context, changing their state from persistent to detached

- **session.clear()**

– Removes all objects from persistence context, changing all their states from persistent to detached

Obtaining the Session

SessionFactory.getCurrentSession()

– Reuses an existing session

• If none exists, creates one and stores it for future use

– Automatically flushed and closed when

transaction.commit() is executed

• FlushMode.AUTO

• **SessionFactory.openSession()**

– Always obtains a brand new session

– Provides/requires more management

• FlushMode.MANUAL

– Developer must manually flush and close the session

Two ways of Reattaching the Objects

HCL TALENTCARE

- Reattach using update()
- Reattach using merge()

Reattaching Object using Update

```
// Get a handle to the current Session
Session session =
HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();

// Modify objects
Account account = new Account();
session.saveOrUpdateAccount(account);

// Close the Session without modifying the persistent object
// Changes the state of the objects to detached
session.getTransaction().commit();
HibernateUtil.getSessionFactory().close();

// Modify detached object outside of session
account.setBalance(2000);
```

Reattaching Object using Update

```
// Get handle to a subsequent Session
Session session2 =
HibernateUtil.getSessionFactory().getCurrentSession();
session2.beginTransaction();

// Reattach the object to be persisted.
// An update statement will be scheduled regardless
// of whether or not object was actually updated
session2.update(account);

// Commits/closes the session
// Saves the changes made in the detached state
session2.getTransaction().commit();
```

Reattaching Object using Merge

HCL TALENTCARE

```
// Get a handle a subsequent Session
Session session2 =
HibernateUtil.getSessionFactory().getCurrentSession();
session2.beginTransaction();

// Reattach the object using merge.
// The data is persisted, but the passed in
// object is STILL in a detached state
session2.merge(account);

// Since this account object is NOT
// persistent, change is not saved
account.setBalance(100);

// Commits/closes session
// Saves the changes made in the detached state
session2.getTransaction().commit();
```

HCL TALENTCARE

HCL TALENTCARE

Thank you