Session: 9

Persistence of Entities

Objectives

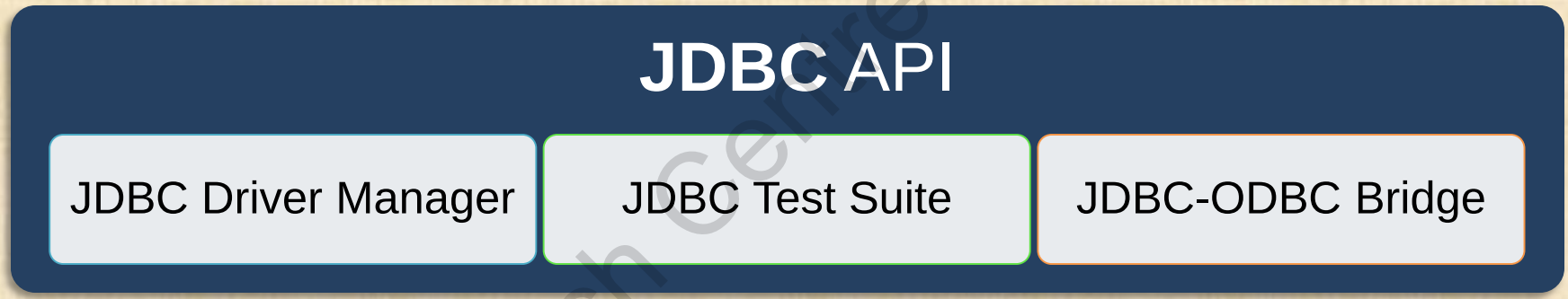
- Understand how to use JDBC API for persisting data
- Explain Object Relational Mapping (ORM)
- Describe the ORM tools used for performing data persistence
- Explain Java Persistence API (JPA)
- Describe Entities and Entity Manager in JPA
- Understand how to manage entities using JPA
- Understand how persistent objects are mapped onto the database

Introduction

- Enterprise applications persist data on relational databases.
- **Relational databases**
  - Stores data in the form of tables.
  - Each table in a relational database comprises rows and columns.
- **Java SE platform**
  - Introduced **JDBC API** for persistence of data in the relational database.
  - **JDBC API** is a low-level API used by Java developers to store and retrieve data from the relational database through Structured Query Language (SQL).
  - Allows the developers of Java applications to establish a connection with the database using a driver.
- Entity Beans were introduced by Java EE for persistence of objects.

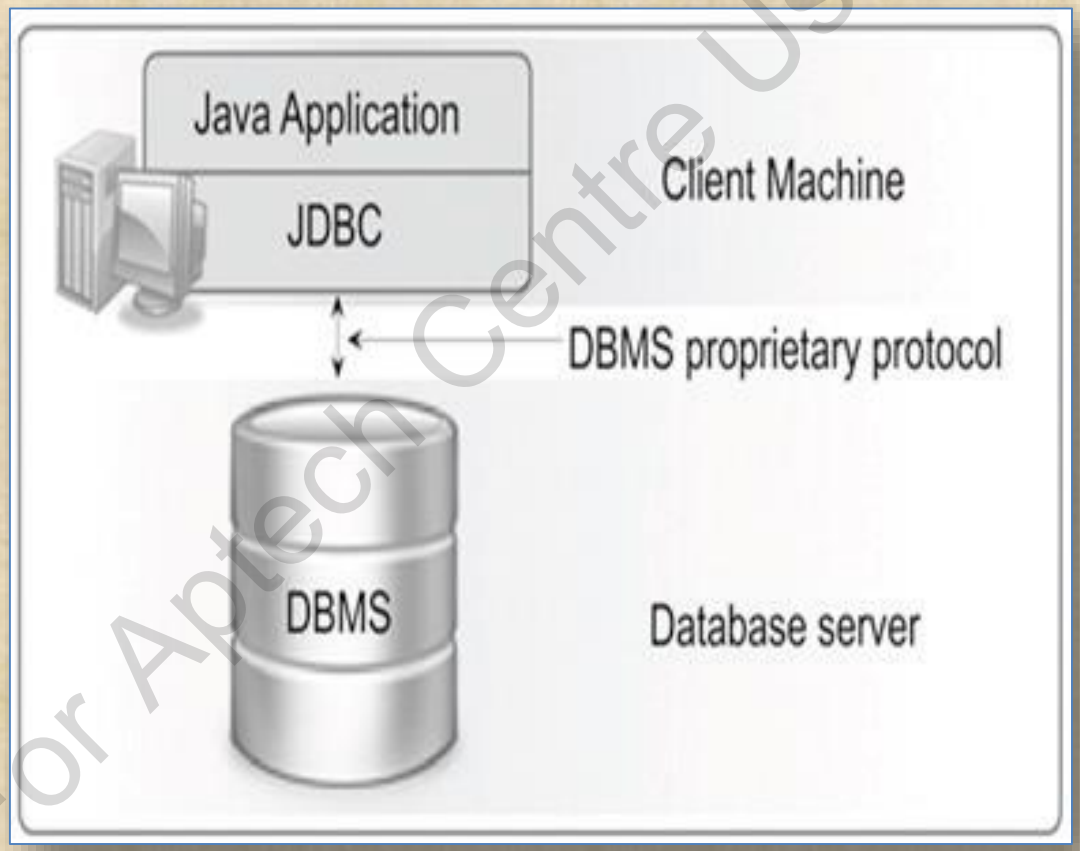
Persistence Using JDBC API 1-2

- It is a set of classes and interfaces to programmatic access the database – `java.sql` and `javax.sql`.
- Following are the components of the JDBC API:



Persistence Using JDBC API 2-2

- Following figure shows the two-tier architecture for accessing the database using JDBC API:



Implementing JDBC API 1-3

- Following are the steps to process SQL statements using JDBC API:

Loading the Drivers

- The JDBC driver provides connection to the database and helps in transferring queries and their results between Java application and relational databases.

Establishing connection

- Driver should be loaded and initialized.
- `DriverManager` or `DataSource` class are used to establish connections.
- `Connection` object implies a database connection.

Implementing JDBC API 2-3

Creating SQL Statements

- Created using interfaces such as `Statement`, `PreparedStatement`, and `CallableStatement`.
- `createStatement()` method of `Connection` class is used to create an SQL statement.

Executing the Query

- Statements are executed using methods `execute()`, `executeQuery()`, and `executeUpdate()`.
- `ResultSet` object is returned when queries are executed.

Implementing JDBC API 3-3

Processing ResultSet object

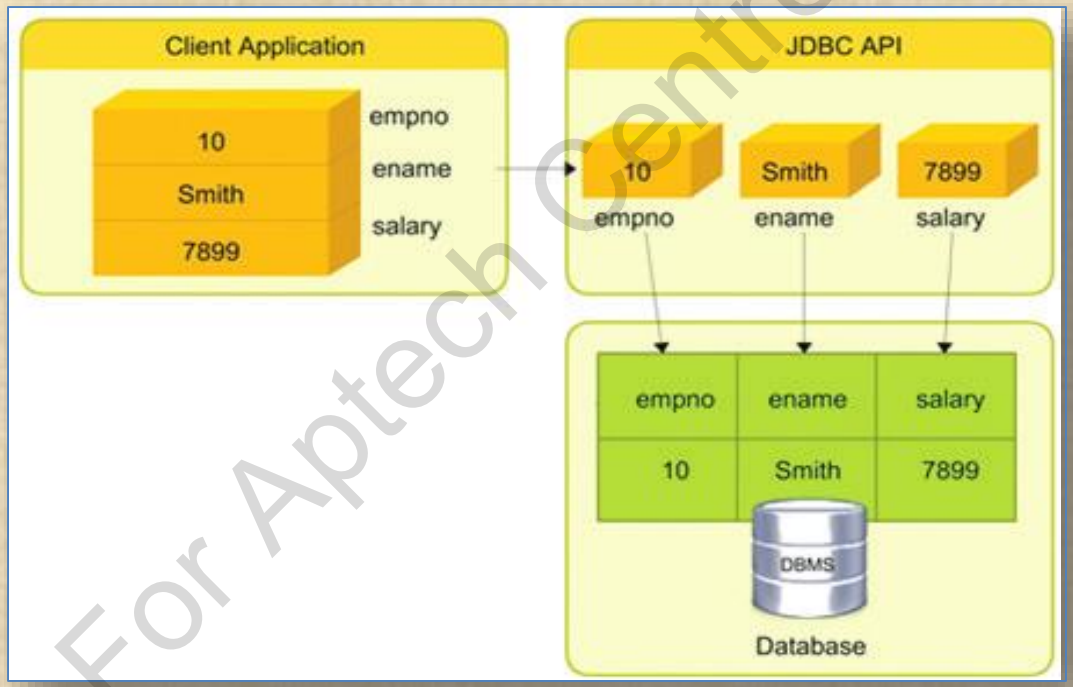
- The `ResultSet` object holds the data retrieved from the query executed in the database.
- The obtained `ResultSet` object is further processed in the application using a cursor.

Closing the Connection

- `close()` method of the `Statement` interface can be used to close the statement on completion.
- `close()` method frees the allocated resources for query execution.

Limitations of JDBC API

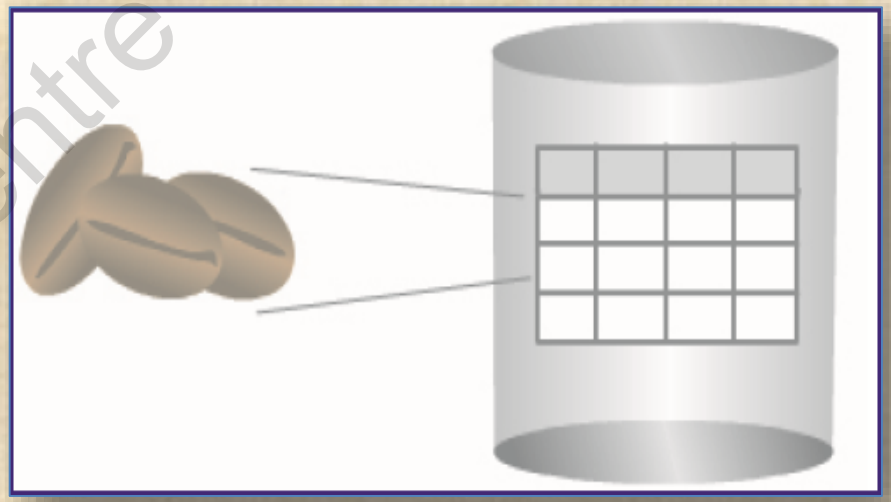
- Following are the drawbacks of the JDBC API:
  - Developers have to manually map the object representation to the relational database model in the JDBC code.
  - JDBC application contains large amount of code which are database specific.
  - Following figure demonstrates the storage of object using JDBC API:



Persistence with EJB 2.0 1-2

- EJB 2.0 introduced Entity Beans for data persistence.
- An Entity corresponds to a row in the relational database table.
- Data is accessed through Entity instances in the application.
- Whenever store or retrieve operation is performed, the data from the database is loaded in the entity instance created in the memory.

- Following figure shows Entity bean model:



Persistence with EJB 2.0 2-2

- Following are the features of the Entity Beans:

Persistence of data

Shared Access by application components

Primary key implementation

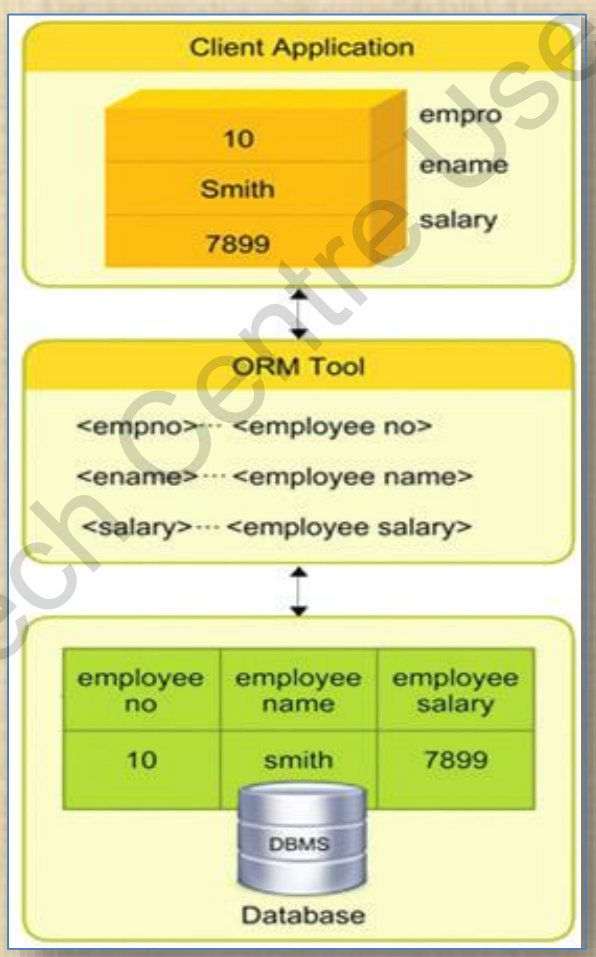
Relationships among various entities

Object Relational Paradigm 1-3

- Object Relational Mapping (ORM) is a technique of persisting objects onto the database.
- ORM automates the task of object persistence to the database.
- ORM can be done either manually or through ORM tools.
- ORM tools contain automatic mapper which generate DDL for the target platform.
- Oracle TopLink and Hibernate are examples of ORM tools.

Object Relational Paradigm 2-3

- Following figure demonstrates persistence of an object through ORM tool:



Object Relational Paradigm 3-3

- The ORM tool contains an automated mapper that generates the data definition of the target platform using DDL.
- These data definitions are generated either from the Java class or from a descriptor file.

ORM Tools 1-2

Java Persistence API

- Manages persistence of Java objects by defining Entity classes

Java Data Objects

- Defines object persistence through external XML meta files

Hibernate

- Open source ORM framework which maps Java classes into database tables

EclipseLink

- Open source ORM tool enabling interaction between application and other data services such as databases, Web services, EIS, and Object XML mapping

ORM Tools 2-2

Fast Java Object Relational Mapping

- Lightweight ORM tool
- Does not support mapping of m:n and 1:n relationships

Java Object–Oriented Querying

- Object mapping software library implementing active record pattern

Active JDBC

- An ORM tool based on active record pattern

Overview of Java Persistence API 1-2

- Was introduced in Java EE 5 as part of EJB 3.0 specification.
- Used to map relational data onto Java objects.
- Used for storage of Java objects onto relational databases.
- Defines a standard mechanism for ORM mapping.
- Manages data by converting the Plain Old Java Objects (POJOs) into entities.
- Current version of JPA in Java EE 7 is JPA 2.1.

Overview of Java Persistence API 2-2

Java Persistence API (JPA)

- Defines a standard mechanism of object-relational mapping.
- Provides specification compliant products known as ORM tools for object-relational mapping.
- Allows inheritance among different entities of the application.
- Allows association of different entities.
- Manage applications' interactions with the database.

Entities

- Persistent objects of enterprise applications are known as **entities**.
- Each entity is associated with data fields and methods.
- For example, `Bank_account` is an entity and it is associated with properties such as `account_number`, `account_holder_name`, and `account_balance`.
- An entity in the application domain represents a table in a relational database.
- Each row of data in the relation represents an entity instance in the application domain.

Requirements of Entity Class

- An Entity class should have a default constructor whose access specifier is `protected` or `public`.
- An Entity class should be annotated with `javax.persistence.Entity`.
- Neither the Entity class nor its methods should be `final`.
- An Entity class must implement `Serializable` interface, if it is passed by value to the business methods.
- Entity classes can be inherited.
- Persistence instance variables can be prefixed with access modifiers such as `public`, `protected`, or `package private`.

Entity Class 1-4

- Following code snippet shows an Entity class:

```
package Manage;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Message implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long message_id;
    private String text;
```


Entity Class 2-4

```
    public Long getmessage_Id() {
        return message_id;
    }

    public void setmessage_Id(Long id) {
        this.message_id = id;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (message_id != null ?
message_id.hashCode() : 0);
        return hash;
    }
```


Entity Class 3-4

```
 @Override
    public boolean equals(Object object) {

        if (!(object instanceof Message)) {
            return false;
        }
        Message other = (Message) object;
        if ((this.message_id == null && other.message_id
!= null) || (this.message_id != null &&
!this.message_id.equals(other.message_id))) {
            return false;
        }
        return true;
    }
```


Entity Class 4-4

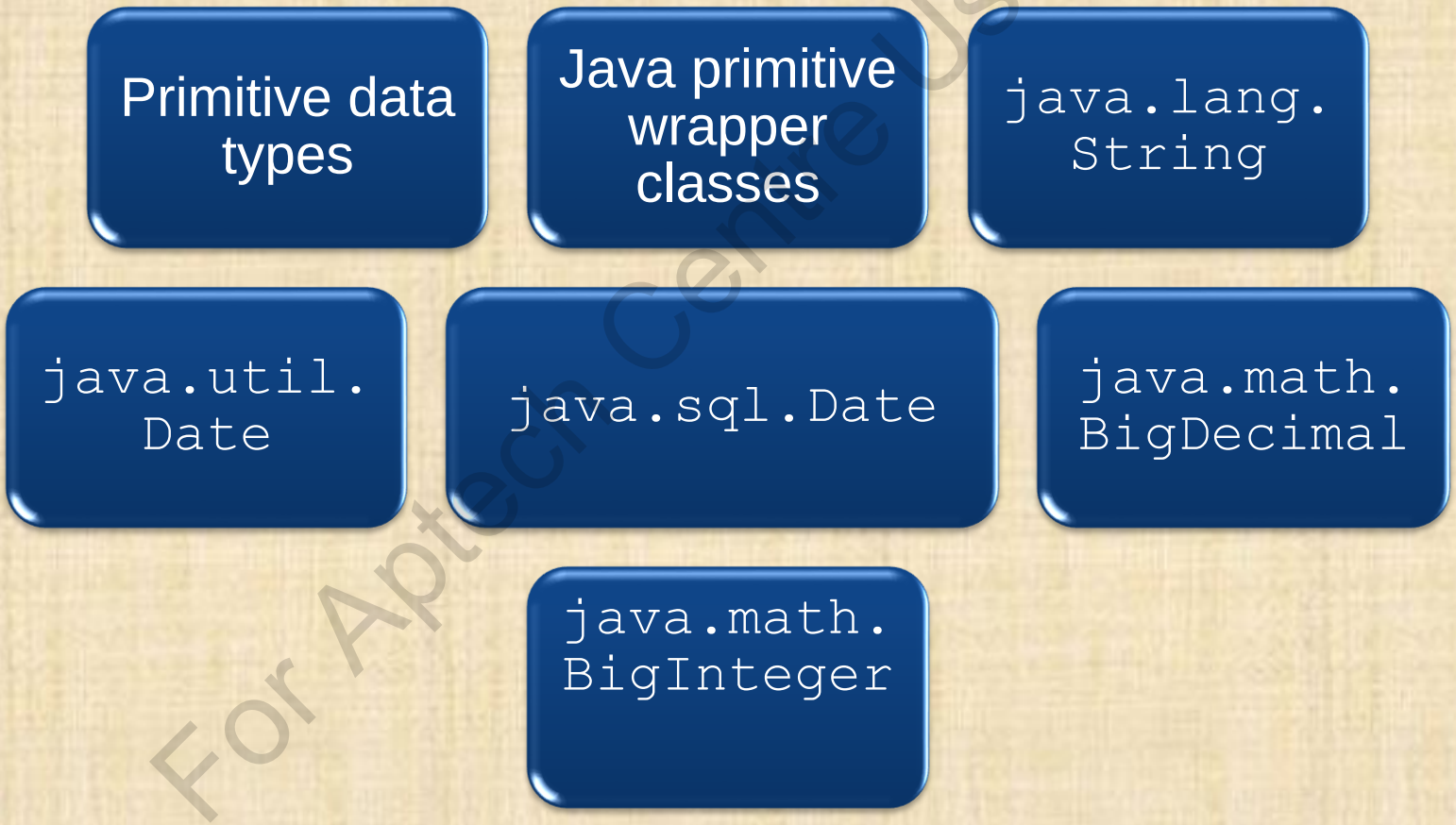
```
@Override
    public String toString() {
        return "Manage.Message[ id=" + message_id +
" ]";
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }
  }
```


Primary Keys in Entities 1-3

- Every entity object is identified through a unique identifier known as primary key.
- Primary key can be a simple primary key or composite primary key.
- Simple primary key is annotated with `javax.persistence.Id`.
- Composite primary keys are defined in the primary key class and are annotated with `javax.persistence.EmbeddedId` and `javax.persistence.Id` class.

Primary Keys in Entities 2-3

- Primary key field can be any one of the following data types:



Primary Keys in Entities 3-3

- A primary key class has the following requirements:
  - Primary key class should have public access modifier.
  - Primary key class properties should be public or protected.
  - It must have a default constructor.
  - It must implement `hashCode()` and `equals(Object other)` methods.
  - It must be serializable.
  - Composite primary key class must be created as either embedded class or be mapped onto the properties of Entity class.

Comparison of Entity with Session bean

Session Beans	Entities
• Instantiated into the container based on application requirement. • Can be accessed through local and remote interface. • Session beans cannot be compared. • Cannot survive application failures. • Does not have an unique identity.	• Number of instances are defined by the application domain. • Cannot be accessed remotely. • Entity instances can be compared. • Can survive application failures. • Has an unique identity.

Packaging and Deploying Entity Classes 1-2

Entity classes are packaged as persistence units

A persistence unit is a set of logically connected entity classes

Persistence units are defined in a configuration file `persistence.xml`

Deployment descriptors are added to `META-INF` directory of the application

Packaging and Deploying Entity Classes 2-2

- Following code snippet shows the definition of a persistence unit in `persistence.xml` file:

```
<persistence>
<persistence-unit name="AccountOperation">
<description>This unit manages accounts of customers
when the customer has multiple accounts this unit links
these accounts </description>
<jta-data-source>jdbc/MyAccounts</jta-data-source>
<jar-file>MyAccount.jar</jar-file>
  <class>FixedDeposit</class>
  <class>SavingsAccount</class>
  <class>CurrentAccount</class>
  </persistence-unit>
</persistence>
```


Persistence Provider

- Persistence provider refers to a third party technology which can be used to manage application data objects on the disk.
- MySQL and Oracle are some of the persistence providers.

Managing Entities

- Entities in a persistence unit are managed through an `EntityManager`.
- `EntityManager` is an instance of `javax.persistence.EntityManager`.
- **EntityManager**
  - Is responsible for managing activities associated with a set of entities such as:
    - persisting the entity
    - Removing an entity
    - Querying the entity
  - Each `EntityManager` instance is associated with a `PersistenceContext`.

PersistenceContext

`PersistenceContext` keeps track of the state changes associated with an entity.

`PersistenceContext` refers to a set of entities existing in an application environment.

There are two variants of persistence context:

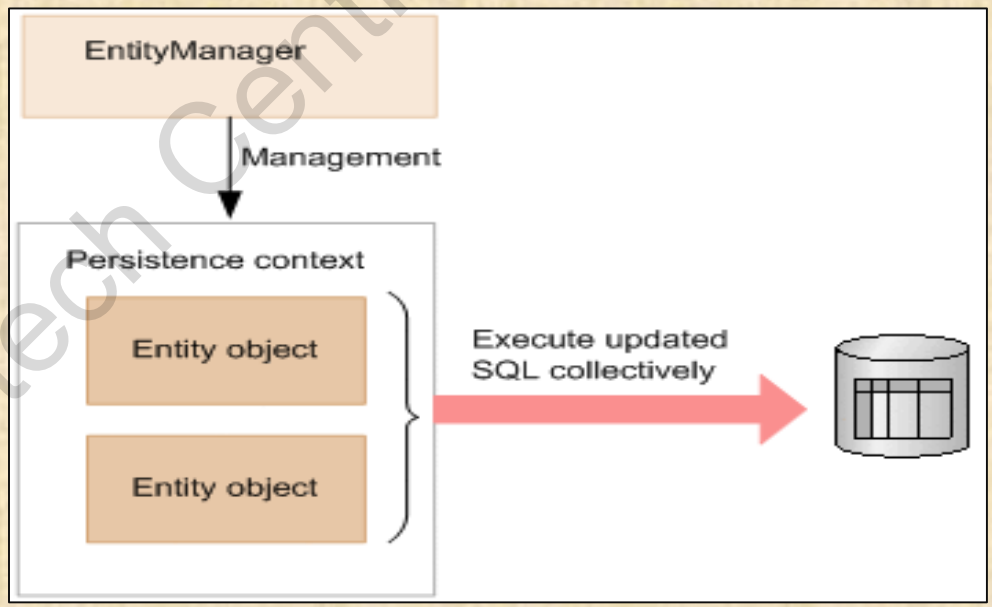
- Transaction scoped persistence context – A set of entities associated with a transaction.
- Extended persistence context – A set of entities independent of any transaction.

EntityManager Interface

- `EntityManager` API is responsible for the following operations:
  - Creating and removing persistent entity instances.
  - Finding entities based on the primary key.
  - Allowing queries to run on entities.
- `EntityManager` instances can be created either through container or through application code.

Container Managed EntityManagers

- `EntityManager` is instantiated by the container.
- It defines `PersistenceContext` and shares it with all the application components.
- The lifecycle of the `EntityManager` is also managed by the container.



Application Managed Entity Managers 1-3

- Application managed entity managers are used when the persistence context is not to be shared among all the application components.
- It creates an isolated `PersistenceContext`.
- Applications create `EntityManager` instance from an `EntityManagerFactory` class.
- `createEntityManager()` method of `EntityManagerFactory` class creates a thread-safe `EntityManager` instance.

Application Managed Entity Managers 2-3

- Following code snippet shows the creation of `EntityManager` instance:

```
package Access;

import Entity.Message;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.PersistenceContext;
@Stateless
public class MessageFacade extends
AbstractFacade<Message> implements
```


Application Managed Entity Managers 3-3

```
MessageFacadeLocal {
    @PersistenceContext(unitName = "MessageStore-ejbPU")
    private EntityManager em;
@Override
    protected EntityManager getEntityManager() {
        return em;
    }
     public MessageFacade() {
        super(Message.class);
    }
    public static void main(){
    EntityManagerFactory emf =
Persistence.createEntityManagerFactory("jdbc:derby://loca
lhost:1527/sample[app on APP]");
     EntityManager e = emf.createEntityManager();
     } }
```


Obtaining a Persistence Context 1-2

- The `PersistenceContext` is defined through the annotation `@PersistenceContext`.
- A `PersistenceContext` can be associated with a transaction.
- Following is the usage of `PersistenceContext` annotation:

```
@PersistenceContext(name = "PersistenceUnitName")
```

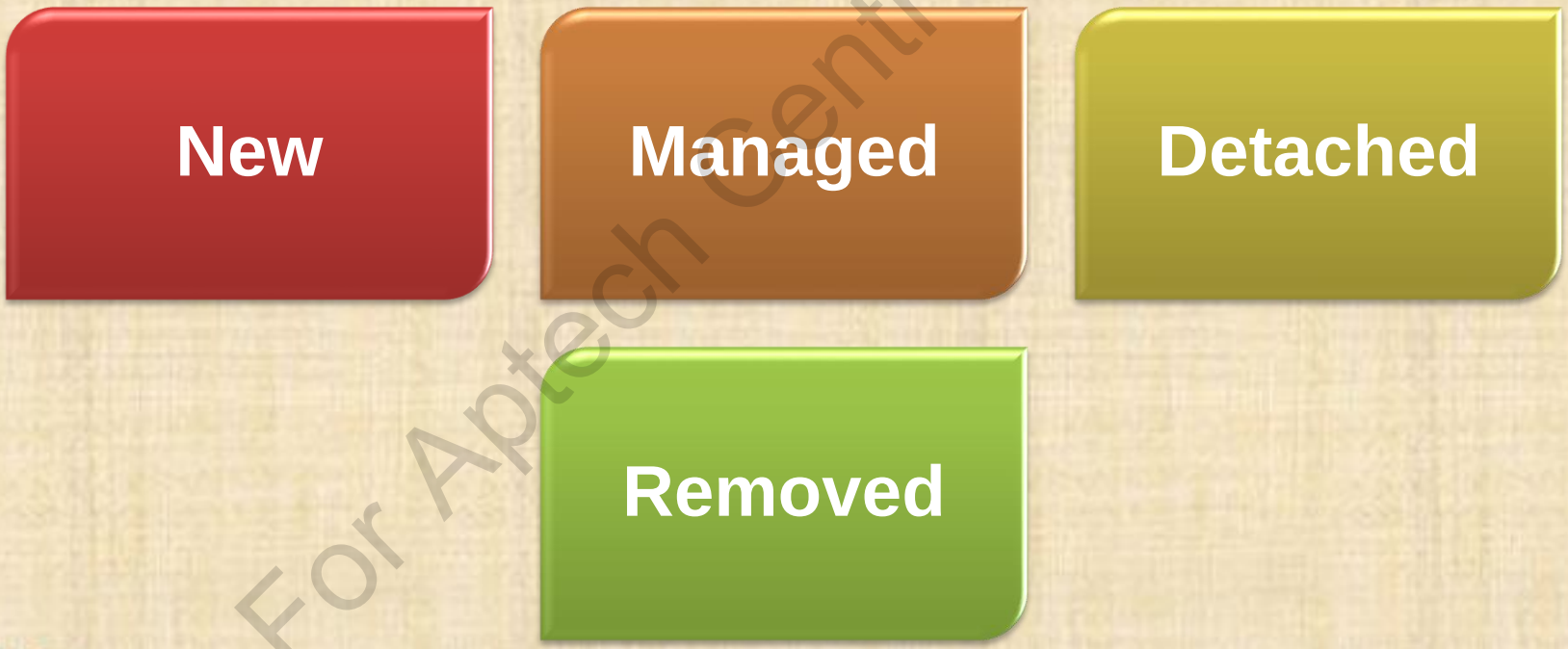

Obtaining a Persistence Context 2-2

- `PersistenceContext` can also be associated with a Stateful Session bean.
- `PersistenceContext` is injected into Stateful Session bean through `@PersistenceContext` annotation.
- Following is the usage of `@PersistenceContext` annotation with a Stateful Session bean:

```
@PersistenceContext(unitName = "MessageStore-ejbPU")
```

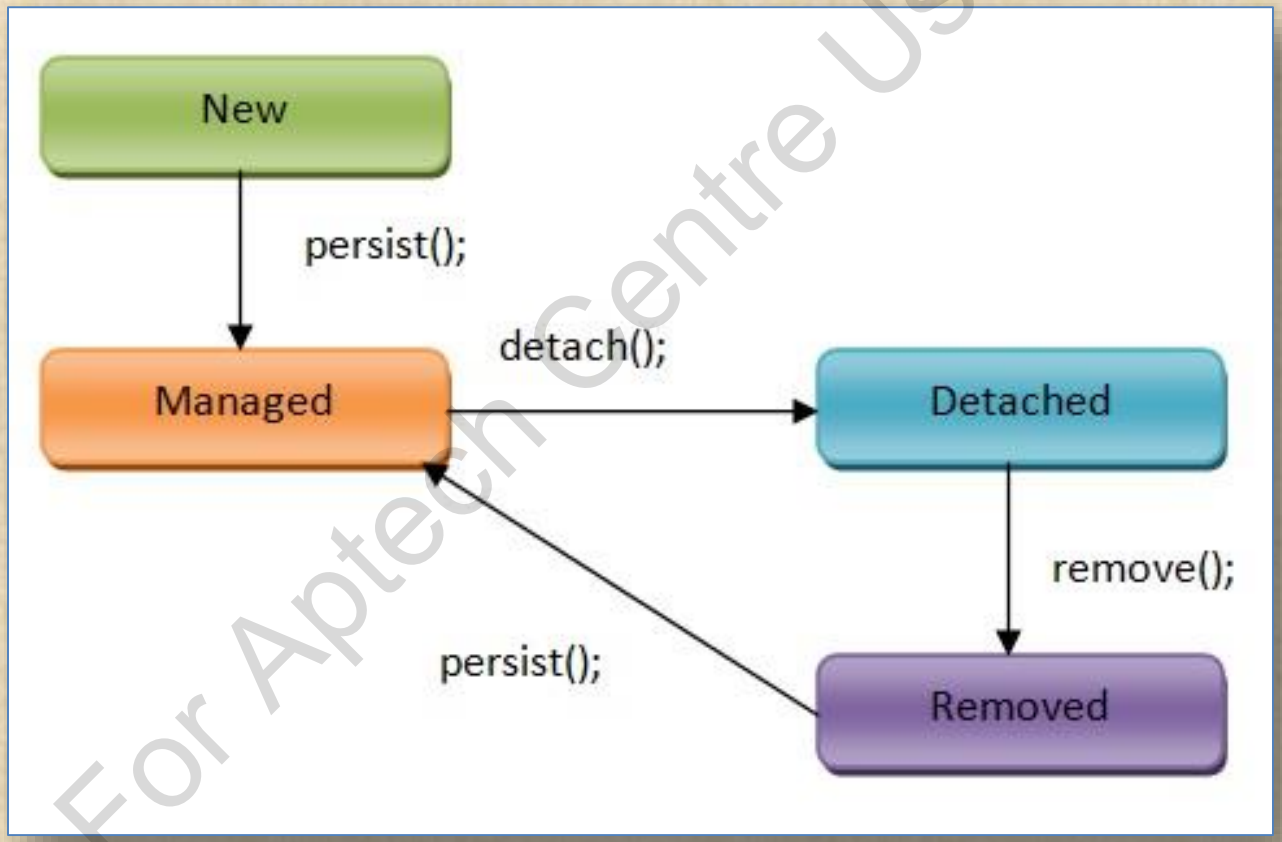

Managing the Lifecycle of an Entity Instance

- The lifecycle of an entity is managed through an `EntityManager`.
- An entity instance can exist in any one of the following states:



Life Cycle Callback

- Following figure shows the transition from one state to another state in the life cycle of an entity:



persist() Method

`persist()` method stores the entity onto the database based on the type of persistence context in use.

Following are the uses of `persist()` method:

- In '**new**' state – assigns an identity to the entity instance.
- In '**managed**' state – the method invocation is ignored.
- In '**removed**' state – the entity is transformed to 'managed' state.
- In '**detached**' state – results in `EntityExistsException`.

If an entity references other entities then `persist()` method invocation is cascaded.

remove() and detach() Method

`remove()` method

- In '**new**' state it is ignored
- In '**managed**' state `remove()` method transforms the entity into '**removed**' state
- In '**removed**' state it is ignored
- In '**detached**' state it throws an `IllegalArgumentException`

`detach()` method

- In all states the `detach()` method removes the entity from the `PersistenceContext`
- If the entity is already in '**detached**' state then it results in `IllegalArgumentException`

Mapping the Persistent Objects

- Objects are persisted onto the relational database by the `EntityManager`
- The code snippet shows the `BankAccount` entity class:

```
. . .
@Entity
public class BankAccount implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    public Long getId() {
        return id;
    }
     public void setId(Long id) {
        this.id = id;
    }}
```


Mapping of Entities onto Database Through XML Files 1-3

- XML files can be used for object-relational mapping.
- `orm.xml` is the deployment descriptor used for Object Relational Mapping.
- It is located in the `META-INF` directory.
- Other `.xml` files can also be used for entity mapping, which are defined through `<mapping-file>` element in `persistence.xml`.

Mapping of Entities onto Database Through XML Files 2-3

- Following code snippet shows the mapping of the `BankAccount` entity:

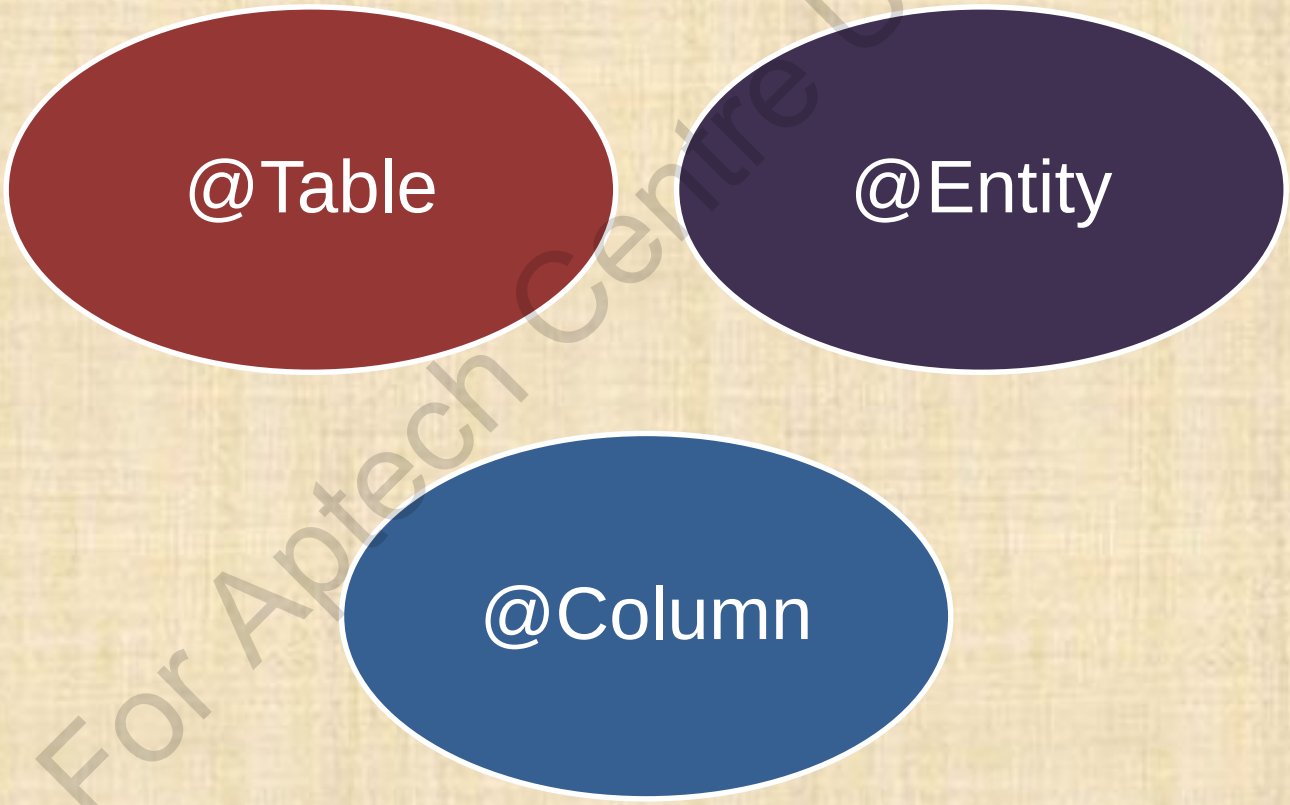
```
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings
. . .
. . .
   <entity class="BankAccount" name="BankAccount">
        <table name="Account"/>
        <attributes>
            <id name="Account_number">
              <generated-value strategy="TABLE"/>
            </id>
            . . .
```


Mapping of Entities onto Database Through XML Files 3-3

```
……
<basic name="account_holder_name">
                <column name="acc_hld_name" length="100"/>
            </basic>
           <basic name="acc_balance">
            </basic>
        </attributes>
   </entity>
</entity-mappings>
```


Mapping of Entities to Databases through Annotations

- Following are the annotations used to define the mapping of the entity classes onto the database:



@Table Annotation

- Used by the `EntityManager` to identify the table of the database.
- Following code snippet shows the usage of `@Table` annotation:

```
@Entity
@Table(name = "BankAccount")
```

- The `@Table` annotation can have four attributes – `name`, `catalog`, `relational schema`, and `uniqueconstraints`.

@Column Annotation

- Used to define how certain property of the entity class can be mapped
- The column annotation has the following attributes:
  - `name`
  - `unique`
  - `nullable`
  - `Insertable` and `updatable`
  - `columnDefinition`
  - `length`
  - `scale` and `precision`

Mapping Primary Keys onto the Database 1-2

- Primary keys are used to uniquely identify the entities of the class.
- Primary keys can be implemented either as a single property or set of properties.
- When implemented as a single property, the property is annotated with `@Id` annotation.
- When the entity class requires generated values for the primary key, it is annotated with `@GeneratedValue`.
- Primary keys can be defined through primary key classes or embedded classes.
- Primary key classes can be annotated with `@IdClass`.
- Embedded primary key classes are annotated with `@EmbeddedId`.

Mapping Primary Keys onto the Database 2-2

- Following code snippet demonstrates the usage of primary key classes:

```
public class BankAccount_PK{

        private long Account_number;
        private String acc_hld_name;
 }
@Entity
@IdClass(BankAccount_PK.class)
public class BankAccount{
@Id
private long acc_num;
@Id
private String account_holder;
. . .
}
```


Summary

- Java applications use Java Database Connectivity (JDBC) and Java Persistence API (JPA) to connect to the database.
- JPA defines a standard mechanism for object-relational mapping.
- JPA interprets the database objects in the application as entity classes.
- JPA defines an EntityManager to manage the entities in the application.
- The primary key for the entity objects can be implemented as an attribute in the entity class, as a primary key class or as an embedded class.
- orm.xml and persistence.xml files have the entity mapping information in the application.
- Persistent objects can be mapped using the declarative XML files.
- The persistent objects can be mapped onto the persistence unit by adding appropriate annotations to the Java classes.