JAVA PERSISTANCE API / SHORT NOTES

1. The JAVA platform is well supported for MANAGING PERSISTANCE to RELATIONAL DATABASES
2. JDBS is an abstraction over proprietary client Interface to the proprietary RELATIONAL DATABASE
3. Enterprise JAVA Beans - Entity Beans were so complex
4. JDO - JAVA DATA OBJECTS , is another PERSISTANCE SPECIFICATION effort in JAVA
5. JDO was not so popular and only handful of people started supporting it
6. The science of BRIDGING the gap between the object model and the relational model is known as OBJECT-RELATIONAL MAPPING. Often referred to as O-R mapping
7. The difference between the Object Model and the RELATIONAL model is always knows as IMPEDENCE MISMATCH
8. The JAVA PERSISTANCE API is a LIGHT WEIGHT , POJO-BASED framework for JAVA persistence
9. The JPA persistence is , NON-INTRUSIVE (persistence objects are no aware of the persistence mechanism) , Provides OBJECT QUERIES [EJB-QL], having MOBILE ENTITIES [ persistent POJOS can be moved from one layer to another , one JVM to another easily using the DETACHMENT model provided], SIMPLE IN CONFIGURATION , TESTABLE
10. Characteristics of objects which have been transformed in to ENTITIES ,
   a. PERSISTABILITY - entities are persistable , means that they can be made PERSISTANT which means their STATE can be REPRESENTED in a DATA STORE and can be ACCESSED at a later time
b. IDENTITY - Entities have an Identity, both in the OBJECT state and while in the DATA BASE as well

c. TRANSACTIONALITY - Entities created, updated, deleted generally in a TRANSACTION

d. GRANULARITY - Entities are fine-grained objects that have a set of AGGREGATED state that is normally stored in one place such as a ROW in a TABLE

11. Entity META DATA
   a. With entities there is some META-DATA at some point
   b. Entity META-DATA can be either ANNOTATION or XML

12. CONFIGURATION BY EXCEPTION - means that the persistence engine defined defaults that apply to the majority of the applications

13. Simple Entity
package com.scea.jpa;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.IdGenerator;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.IdGenerator;
import javax.persistence.Table;

@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;

    public Employee(){}
    public Employee(int i){
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public long getSalary() {
        return salary;
    }

    public void setSalary(long salary) {
        this.salary = salary;
    }
}

14. When ANNOTATIONS are placed either those can be placed at FIELD or PROPERTY level. That means user can select to ANNOTATE declared FIELD or GETTER METHOD
15. Users can select any of the above, but should adhere to the once that is used in one ENTITY
16. EntityManager interface is used for almost all the PERSISTANT operations
17. The set of MANAGED entity instances within an ENTITY MANAGER at any given time is called its PERSISTANCE CONTEXT.
18. Only one JAVA INSTANCE with the same PERSISTANCE INDENTITY may exist in a PERSISTANCE CONTEXT as any time.
19. It is the PROVIDER that supplies the BACKING IMPLEMENTATION engine for the ENTIRE JAVA PERSISTANCE API.
20. All the ENTITY MANAGES come from FACTORIES of ENTITYMANAGERFACTORY. The configuration for an entity manager is bound to the EntityMangerFactory that CREATED it. But it is defined SEPARATELY as a PERSISTANCE UNIT.
21. PERSITECE UNIT – Dictates either implicit or explicit the SETTINGS and ENTITY CALSSES used by all ENTITY MANGERS obtained from the UNIQUE ENTITY MANGER FACTORY instance BOUND to that PERSISTANCE UNIT.
22. An ENTITY MANAGER is always obtained from an ENTITY MANAGER FACTORY.

23. Creating an ENTITY MANAGER FACTORY in J2SE environment
a. EntityMangerFactroy emf = Persistance.createEntityMangerFactory("EmployeeService");

25. Obtaining an Entity Manager
   a. EntityManger em = emf.createEntityManger();

26. Persisting an Entity
   a. Employee emp = new Employee(158);
   b. Em.persist(emp);

27. Finding an Entity
   a. Employee emp = em.find(Employee.class,158)

28. Removing an Entity

29. In order to REMOVE and entity, it must be MANAGED
   a. Employee emp = em.find(Employee.class, 158)
   b. em.remove(emp)

30. Updating an Entity
   a. Employee emp = em.find(Employee.class,158);
   b. emp.setSalary(emp.getSalary() + 1000)

31. While updating no ENTITY MANAGER is invoked, For this the ENTITY must be a MANAGED one at this time

32. Except FIND, all other MUST BE INVOKED in a TRANSACTION

33. In case of TRANSACTIONS, the environment in which operations are performed is important, it could be an APPLICATION SERVER or STANDALONG APPLICATION

34. When RUNNING inside a JAVA EE CONTATINER, standard JTA would be in use

35. Transaction Service that should be used in JAVA SE environment is EntityTransaction service

36. In JAVA SE environment
   a. em.getTransaction().begin()
b. `createEmployee(158,"John Doe",45000);`

c. `em.getTransaction().commit();`

37.

38. Creating Query
   a. `Query query = em.createQuery("SELECT e FROM Employee e");`
   b. `Collection emps = query.getResultList();`

39. PERSISTANCE UNIT - The configuration that describes the persistence unit is in XML file named “PERSISTANCE.XML”

40. A single persistence.xml file may contain many PERSISTENCE UNIT configurations

41.

```xml
<persistence>
  <persistence-unit name="EmployeeService" transaction-type="RESOURCE_L
  <class>examples.model.Employee</class>
  <properties>
    <property name="toplink.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
    <property name="toplink.jdbc.url" value="jdbc:derby://localhost:1527/EmpServDB;create
    <property name="toplink.jdbc.user" value="APP"/>
    <property name="toplink.jdbc.password" value="APP"/>
  </properties>
</persistence-unit>
</persistence>
```

42. A persistence archive is a SIMPLE JAR file containing PERSISTANCE.XML in META-INF folder

43. DEPENDENCY MANAGEMENT can be done through Dependency Lookup and Dependency Injection.
44. Dependency Look up is traditional form of Dependency management in JEE, where application code is responsible for using JNDI to look up named references.

45. The process of automatically looking up a resource and setting it into the class is called DEPENDENCY INJECTION.

46. The server is said to inject the resolved dependency into the class and this is known as DEPENDENCY INJECTION.

47. There are Few FORMS of DEPENDENCY INJECTION, FIELD INJECTION and SETTER INJECTION.

48. Referencing a PERSISTENCE CONTEXT

Listing 3-14. Injecting an EntityManager Instance

```java
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    // ...
}
```

49. Referencing a PERSISTENCE UNIT
a.

**Listing 3-15. Injecting an EntityManagerFactory Instance**

```java
@Stateful
public class EmployeeServiceBean implements EmployeeService {
  @PersistenceUnit(unitName="EmployeeService")
  private EntityManagerFactory emf;
  private EntityManager em;

  @PostConstruct
  public void init() {
    em = emf.createEntityManager();
  }

  // ...
}
```

50. REFERENCING AN EJB

**Listing 3-16. Qualifying an EJB Reference Using the Bean Name**

```java
@Stateless
public class DeptServiceBean implements DeptService {
  @EJB(beanName="AuditServiceBean")
  AuditService audit;

  // ...
}
```

a.

51. Referencing Server RESOURCE
RESOURCE LOCAL transactions are ALWAYS demarcated by the APPLICATION

CONTAINER MANAGED transactions may EITHER be demarcated AUTOMATICALLY by the CONTAINER or by using JTA interface that support APPLICATION CONTROLLED demarcation

EJB may use wither CONTAINER MANAGED or BEAN MANAGED TRANSACTION

SERVLETS are limited to somewhat poorly named BEAN MANAGED TRANSACTIONS

The DEFAULT transaction management STYLE for EJB is CONTAINER MANAGED TRANSACTIONS

A Message Driven Bean fully support Injecting Entity Manager and can leverage CONTAINER MANAGED transactions as well

Adding the STATELESS SESSION beans as components to MANAGE PERSISTANCE operations is the PREFERRED strategy for JAVA EE APPLICATIONS
STATEFUL SESSION BEANS are also well suited to MANAGING PERSISTENCE OPERATIONS within an APPLICATION COMPONENT MODEL.

The ability to STORE the STATE on the SESSION BEAN allows creating QUERY criteria or other CONVERSATIONAL state CONSTRUCTED ACROSS multiple METHOD CALLS.

In JAVA EE environment, many properties required in the PERSISTENCE.XML file for JAVA EE may be omitted. Instead of JDBC properties for creating a connection, it is possible to add DATA SOURCE NAME, jdbc/EmployeeDS.
64. At some situations when not all the data is required, LAZY FETCHING can be used. The FETCH TYPE of a BASIC mapping can be declared as LAZY as follows

```java
@Basic(fetch=FetchType.LAZY)
@Column(name="COMM")
private String comments;
```

65. Persistence provider is not REQUIRED to provide LAZY fetching, this is only a HINT to the PERSISTENCE provider

66. It is NEVER a good idea to LAZY FETCH SIMPLE types in an ENTITY

67. LAZY FETCHING is better when it comes to RELATIONSHIP MAPPINGS

68. CLOB holds CHARACTER LARGE OBJECT while BLOB holds BINARY LARGE OBJECTS

```java
@Entity
public class Employee{
    @Id
    private int id;
    @Basic(fetch=FetchType.LAZY)
    @Lob @Column(name="PIC")
    private byte[] picture;
}
```

69. Enumerations can have two types, ORDINAL (will take the ordinal values such as 1,2,....n) or STRING (Place the String value), such as

```java
@Entity
public class Employee{
    @Id
    private int id;
    @Enumerated(EnumType.STRING)
```
private EmployeeType type;
}

70. TEMPARAL types are TIME BASED types used in PERSISTENCE mapping. The type SQL are completely hassle free and do not need special consideration. Java.Util types need additional attention. There are THREE enumerated values of DATE, TIME and TIMESTAMP to use with java.util types.
   a. java.sql.Date
   b. java.sql.Time
   c. java.sql.Timestamp
   d. java.util.Date
   e. java.util.Calendar

@Entity
public class Employee{
   @Id
   private int id;
   @Temporal(TemporalType.DATE)
   private Calendar dob;
}

71. Use @Transient, or transient keyword to keep Entity properties from saving in to persistence store

72. There are different ID GENERATION algorithms in use in JPA

73. AUTO, TABLE, SEQUENCE, IDENTITY would be used for generating id
   a. AUTO - Application does not care what kind of generation is used by the PROVIDER but wants generation to occur. The AUTO mode is really a DEVELOPMENT or PROTOTYPE strategy
b. TABLE – Id generation using table is the most flexible and portable way.
c. SEQUENCE – Id generation using a DATABASE sequence
d. IDENTIFY – Id generation using Database Identity. Some DBs support PRIMARY key IDENTIFY columns.

74. SINGLE VALUED ASSOCIATION – An association from an ENTITY instance to ANOTHER (where the CARDINALITY of the TARGET is ONE) is called a SINGLE VALUED ASSOCIATION, Many-to-One and One-to-One fall to this category

75. COLLECTION VALUED ASSOCIATION – when the source entity references one or more TARGET entity instances, a MANY VALUES association is used – One to Many and Many to Many fall in to this category

76. There are few mapping type
   a. Many-to-One
b. One-to-One

**Figure 4-12. EMPLOYEE and PARKING_SPACE tables**
i. Bi Directional One-to-One

![Figure 4-12. EMPLOYEE and PARKING_SPACE tables](image)

```java
@Entity
class Employee {
    @Id private int id;
    private String name;
    @OneToOne
    @JoinColumn(name="PSPACE_ID")
    private ParkingSpace parkingSpace;
    // ...
}
```
iii.

![Diagram](image)

**Figure 4-13. One-to-one relationship between Employee and ParkingSpace**

```java
@Entity
public class ParkingSpace {
    @Id private int id;
    private int lot;
    private String location;
    @OneToOne(mappedBy="parkingSpace")
    private Employee employee;
    // ...
}
```

iv.

```java
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @OneToOne
    @JoinColumn(name="PSPACE_ID")
    private ParkingSpace parkingSpace;
    // ...
}
```

v.
vi. One-to-One Primary Key Mapping

vii. A specific case of a UNIQUE ONE-to-ONE relationship is when the primary keys of the related entities are GUARANTEED to match

```java
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @OneToOne(mappedBy="employee")
    private ParkingSpace parkingSpace; // ...
}

@Entity
public class ParkingSpace {
    @Id private int id;
    private int lot;
    private String location;
    @OneToOne
    @PrimaryKeyJoinColumn
    private Employee employee; // ...
}
```

viii. 

![Diagram](image)

Figure 4-14. EMPLOYEE and PARKING_SPACE tables with shared primary keys

c.
d. One-to-Many

i. A BI DIRECTIONAL One-to-Many mapping ALWAYS implies Many-to-One mapping back to the SOURCE
ii. One to Many association is almost ALWAYS BI DIRECTIONAL

Listing 4-20. One-to-Many Relationship

```java
@Entity
public class Department {
    @Id private int id;
    private String name;
    @OneToMany(mappedBy="department")
    private Collection<Employee> employees;
    // ...
}
```

iv. There are important points to remember when mapping One to Many associations, Many-to-One side is the OWNING side, so the JOIN COLUMN is defined on that side

vi. The One-to-Many mapping is the INVERSE side, so the MAPPED BY is on that side

vii. There are One-To-Many UNIDIRECTIONAL mappings also available

viii. Unidirectional mappings need a JOIN TABLE
ix.

x.

Listing 4-24. Unidirectional One-to-Many Relationship

```java
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @OneToMany
    @JoinTable(name="EMP_PHONE",
                joinColumns=@JoinColumn(name="EMP_ID"),
                inverseJoinColumns=@JoinColumn(name="PHONE_ID"))
    private Collection<Phone> phones;
    // ...
}
```

xi.

Figure 4-19. Join table for a unidirectional one-to-many relationship

e. Many-to-Many

i. The ONLY way to implement a Many-To-Many relationship is with a SEPARATE JOIN table
**Listing 4-22. Many-to-Many Relationship Between Employee and Project**

```java
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @ManyToMany
    private Collection<Project> projects;
    // ...
}

@Entity
public class Project {
    @Id private int id;
    private String name;
    @ManyToMany(mappedBy="projects")
    private Collection<Employee> employees;
    // ...
}
```

**Figure 4-16. Bidirectional many-to-many relationship**

```java
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @ManyToMany
    @JoinTable(name="EMP_PROJ",
                joinColumns=@JoinColumn(name="EMP_ID"),
                inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
    private Collection<Project> projects;
    // ...
}
```
iv. When one side of the Many-to-Many relationship does not have a mapping to the other, then it is a UNIDIRECTIONAL relationship.

vi. The Join table must still be used, the difference is that only ONE of the TWO entity types ACTUALLY uses the table to LOAD its related entities or UPDATES it to store additional entity ASSOCIATIONS.

vii. In BOTH two UNIDIRECTIONAL collection VALUED cases, there is no COLLECTION ATTRIBUTE in the TARGET ENTITY, **mappedBy** elements will not be PRESENT in the @OneToMany annotation on the SOURCE ENTITY.

77. PERSISTENCE UNIT - Named configuration of ENTITY classes
78. PERSISTENCE CONTEXT - Managed set of ENTITY INSTANCES
79. There are THREE different types ENTITY MANGERS
   a. CONTAINER MANAGED ENTITY MANAGERS - Container manages the LIFECYCLE of the ENTITY MANAGER
i. TRANSACTION SCOPED CONTAINER MANAGED ENTITY MANAGERS – The persistence context that the Transaction Manager works with would be decided by the ACTIVE JTA transaction. Whenever the ENTITY MANAGER is injected into the em field using @PersistenceContext. Transaction Scoped ENTITY MANAGERS are STATELESS

1. All CONTAINER MANAGED ENTITY MANAGERS depend on JTA TRANSACTIONS
2. ENTITY MANAGER can use the JTA as a way to track PERSISTANCE CONTEXT associated with it
3. Every time when an operation is invoked on ENTITY MANAGER it checks in the JTA whether existing PERSISTANCE CONTEXT, if yes use that, if no CREATE NEW
4. When the TRANSACTION ENDS, PERSISTANCE CONTEXT goes away

ii. EXTENDER CONTAINER MANAGED ENTITY MANAGERS – Works with a SINGLE persistence context that is tied to the life cycle of a STATEFULL SESSION BEAN

1. Prevents ENTITIES becoming DETACHED when transaction ENDS
2. The EXTENDED persistence context allows STATEFUL session beans to be written in a way that is more NATURAL.

```java
@Stateful
public class DepartmentManagerBean implements DepartmentManager {
    @PersistenceContext(unitName="EmployeeService",
                        type=PersistenceContextType.EXTENDED)
    EntityManager em;
    Department dept;

    public void init(int deptId) {
        dept = em.find(Department.class, deptId);
    }

    public void setName(String name) {
        dept.setName(name);
    }

    public void addEmployee(int empId) {
        Employee emp = em.find(Employee.class, empId);
        dept.getEmployees().add(emp);
        emp.setDepartment(dept);
    }

    // ...

    @Remove
    public void finished() {
    }
}
```

3. The EXTENDED persistence context allows STATEFUL session beans to be written in a way that is more NATURAL.
4. EXTENDED entity manager CREATES a PERSISTENCE CONTEXT when a STATEFUL session bean instance is created that LASTS until the bean is REMOVED

5. The BIGGEST limitation of the EXTENDED persistence context is that it REQUIRES a STATEFULL SESSION BEAN

b. APPLICATION MANAGED ENTITY MANAGERS
   i. Any ENTITY MANAGER created by calling createEntityManager() method on an ENTITYMANAGERFACTORY is an APPLICATION MANAGED ENTITY MANAGER
   ii. Application manages the LIFECYCLE of the ENTITY MANAGER
   iii. APPLICATION MANAGED ENTITY MANAGERS are the only available option in J2SE environment
iv. To

```
Listing 5-5. Application-Managed Entity Managers in Java SE

public class EmployeeClient {
    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("EmployeeService");
        EntityManager em = emf.createEntityManager();

        Collection emps = em.createQuery("SELECT e FROM Employee e")
            .getResultList();
        for (Iterator i = emps.iterator(); i.hasNext();)
            System.out.println(i.next().getId() + ", " + i.next().getName());
    }
}
```

v.

vi. To create an APPLICATION MANAGED ENTITY MANAGER in JAVA EE environment, use @PersistenceUnit annotation
vii. In terms of PERSISTENCE CONTEXT, application managed entity manager is similar to CONTAINER MANAGED EXTENDED ENTITY MANAGER.

viii. If RESOURCE LOCAL transaction is required, then APPLICATION MANAGED ENTITY MANAGERS are the only option in J2EE environment.

Listing 5-6. Application-Managed Entity Managers in Java EE

```java
class LoginServlet extends HttpServlet {
    @PersistenceUnit(unitName="EmployeeService")
    EntityManagerFactory emf;

    protected void doPost(HttpServletRequest request, HttpServletResponse response) {
        String userId = request.getParameter("user");

        // check valid user
        EntityManager em = emf.createEntityManager();
        try {
            User user = em.find(User.class, userId);
            if (user == null) {
                // return error page
                // ...
            }
        } finally {
            em.close();
        }

        // ...
    }
```
80. There are TWO transaction MANAGEMENT types supported by JAVA PERSISTENCE API
   a. RESOURCE LOCAL TRANSACTION
   b. JTA TRANSACTIONS
81. CONTAINER MANAGED ENTITY MANAGES always use JTA while APPLICATION MANAGED ENTITY MANAGERS can use either of those
82. Only ONE persistence context could be PROPAGATED with a JTA transaction
83. EXTENDED persistence context would always try to make itself the ACTIVE PERSISTENCE CONTEXT
84. This leads to a situation where TWO PERSISTENCE CONTEXTS COLLIDE with each other
85. There can only be ONE ACTIVE PERSISTENCE CONTEXT per TRANSACTION
86. One way to WORKAROUND this problem is to change the DEFAULT transaction attribute for the STATEFUL SESSION bean that uses EXTENDED persistence context
87. One last option to CONSIDER using an APPLICATION MANAGED entity manager INSTEAD of an EXTENDED ENTITY MANAGER if there is no need to PROPAGATE the persistence context
88. There is no limit to the number of application managed PERSISTENCE CONTEXT that may be SYNCHRONIZED with a transaction, but only one CONTAINER MANAGED PERSISTENCE CONTEXT will ever be associated
89. APPLICATION MANAGED persistence context may be SYNCHRONIZED with JTA transactions. Synchronizing means FLUSH will occur if the transaction commits
90. APPLICATION MANAGED ENTITY MANAGER participate in a JTA transaction in one of two ways
   a. If the PERSISTENCE CONTEXT is created INSIDE the TRANSACTION, then the PERSISTENCE PROVIDER will AUTOMATICALLY SYNCHRONIZE the PERSISTENCE CONTEXT with the TRANSACTION
   b. If the PERSISTENCE CONTEXT is CREATED EARLIER, outside the transaction, the PERSISTENCE CONTEXT may be MANUALLY SYNCHRONIZED by calling `joinTransaction()` on the ENTITY MANAGER
   c. Once SYNCRHONIZED The persistence context will AUTOMATICALLY BE FLUSHED when the transaction COMMITS
PERSISTENCE CONTEXT INHERITANCE - When a STATEFUL SESSION BEAN with an EXTENDED PERSISTENCE context creates another STATEFUL SESSION BEAN that also uses an EXTENDED PERSISTENCE CONTEXT, the CHILD will INHERIT from the PARENT.

Listing 5-12. Using Application-Managed Entity Managers with JTA

```java
@Stateful
public class DepartmentManagerBean implements DepartmentManager {
    @PersistenceUnit(unitName="EmployeeService")
    EntityManagerFactory emf;
    EntityManager em;
    Department dept;

    public void init(int deptId) {
        em = emf.createEntityManager();
        dept = em.find(Department.class, deptId);
    }

    public String getName() {
        return dept.getName();
    }

    public void addEmployee(int empId) {
        em.joinTransaction();
        Employee emp = em.find(Employee.class, empId);
        dept.getEmployees().add(emp);
        emp.setDepartment(dept);
    }
}
```

91. PERSISTENCE CONTEXT INHERITANCE - When a STATEFUL SESSION BEAN with an EXTENDED PERSISTENCE context creates another STATEFUL SESSION BEAN that also uses an EXTENDED PERSISTENCE CONTEXT, the CHILD will INHERIT from the PARENT.
92. A DETACHED entity is one that is not associated with a PERSISTENCE CONTEXT

93. MERGING is the opposite of the DETACHMENT

94. RESOURCE LOCAL TRANSACTIONS - Resource local transactions are controlled by the APPLICATION

95. Applications interact with the RESOURCE LOCAL TRANSACTION by acquiring an implementation of the EntityTransaction interface from the EntityManager

96. The getTransaction() method of the EntityManager interface is used for this purpose

97. The EntityTransaction interface is designed to IMITATE user transaction interface

98. ```java
public interface EntityTransaction {
    public void begin();
    public void commit();
    public void rollback();
    public void setRollbackOnly();
    public void getRollbackOnly();
    public void isActive();
}
```
One of the examples of using RESOURCE LOCAL transactions in Application server environment is LOGGING.

By Default every ENTITY MANAGER operation applies only to the ENTITY SUPPLIED as an ARGUMENT to the OPERATION. The operation will not CASCADE to other entities that have a relationship with the entity that is being operated on.

Cascading means propagating the operations to the relationships as well. Usually in JPA the CascadeType can be used. PERSIST, REFRESH, REMOVE, MERGE.
103. REFRESH method of the EntityManager is used to refresh the entity state
104. REFRESH operation is applied only when the ENTITY is MANAGED
105. When REFRESH is invoked it will OVERWRITE the ENTITY STATE from the DATA BASE
106. JPA entities can have CALLBACK methods which are executed whenever a lifecycle method is invoked
107. Those CALLBACK methods could be annotated with, @PrePersist, @PostPersist, @PreUpdate, @PostUpdate, @PreRemove, @PostRemove, @PostLoad
108. Entity LIFECYCLE CALLBACK methods can be defined on ENTITY LISTENERS classes or in the ENTITY itself

```java
@Entity
@EntityListeners(com.acme.AlertMonitor.class)
public class Account {
    Long accountId;
    Integer balance;
    boolean preferred;

    @Id
    public Long getAccountId() { ... }
    public Integer getBalance() { ... }
    ...
    @Transient // because status depends upon nc
    public boolean isPreferred() { ... }
}
```

```
public class AlertMonitor {
    @PostPersist
    public void newAccountAlert(Account acct) {
        Alerts.sendMarketingInfo(acct.getAccountId(), acct.getBalance());
    }
}
```
111. EXTENDED entity managers can only be used with STATEFUL EJB, with STATELESS and MDB it is not allowed to use
112. JPA ENTITY CLASSES can not use container services such as SECURITY, TRANSACTION ATTRIBUTES, RESOURCE INJECTIONS since those are not managed UNDER the CONTAINER
113. ENTITY MANAGERS are NOT THREAD SAFE
114. You can INJECT ENTITY MANAGERS to SERVLETS and even JSF BACKING BEANS
115. CONTAINER MANAGED TRANSACTION is not available in WEB CONTAINER, hence you have to use BMT and UserTransaction interface for transaction demarcation if you use JTA resource
116. If “EntityManager.getTransaction()” is invoked on a JTAEntity Manager it would throw an Exception. EntityManager.getTransaction() can only be invoked on NON JTA Entity Managers (RESOURCE_LOCAL). This would return EntityTransactionService interface which mimics the UserTransaction interface
LIYANA ARACHCHIGE RANIL

BEAN MANAGED TRANSACTION CONTAINER
MANAGED TRANSACTION

117.