Session: 12

Concurrency, Listeners, and Caching

Objectives

- ❑ Describe concurrency utilities provided by Java EE
- ❑ Explain locking techniques used to enable concurrency in entities
- ❑ Describe different locking modes
- ❑ Explain entity lifecycle callback events
- ❑ Explain how to inject external listeners to handle lifecycle callback events
- ❑ Explain caching techniques
- ❑ Explain how to specify caching modes in applications

Introduction 1-2

- Multiple applications access entity data simultaneously.
- Concurrent access to data is protected through transaction isolation techniques.

- Database providers provide locking techniques to maintain data integrity.

Introduction 2-2

- JPA specification has defined two important mechanisms to tune concurrent access to entities:
  - Optimistic locking
  - Explicit read and write locks

Optimistic Locking 1-2

- Does not extensively acquire locks.
- Provides high degree of concurrent access.
- Checks the consistency of the data read from the database before committing the transaction.
- JPA specification configures transaction isolation level to `READ COMMITTED` by default.
- `javax.persistence.Version` attribute of entities used to check the consistency of the data.

Optimistic Locking 2-2

- **Working of optimistic locking:**
  - Before the transaction commits, optimistic locking technique ensures the following:
    - If an update operation is performed on an old version of the entity instance corresponding to the database table row whose data is already being updated by an entity instance running in other transaction boundary.
    - Then, an exception is thrown. The exception named `OptimisticLockException` is thrown by the persistence provider when such a conflict occurs.

Version Attribute 1-2

Persistence provider modifies the `Version` attribute whenever the entity state data is modified.

Following are the requirements of `Version` attribute:

- Each entity class is associated with only one `Version` attribute
- Only persistence provider can set or update the value of `Version` attribute
- `Version` attribute should be present in the primary table of the database
- `Version` attribute can be of type - `int, Integer, long, Long, short, Short, or java.sql.Timestamp`

Version Attribute 2-2

- Following code snippet shows the `Version` attribute provided to an entity class:

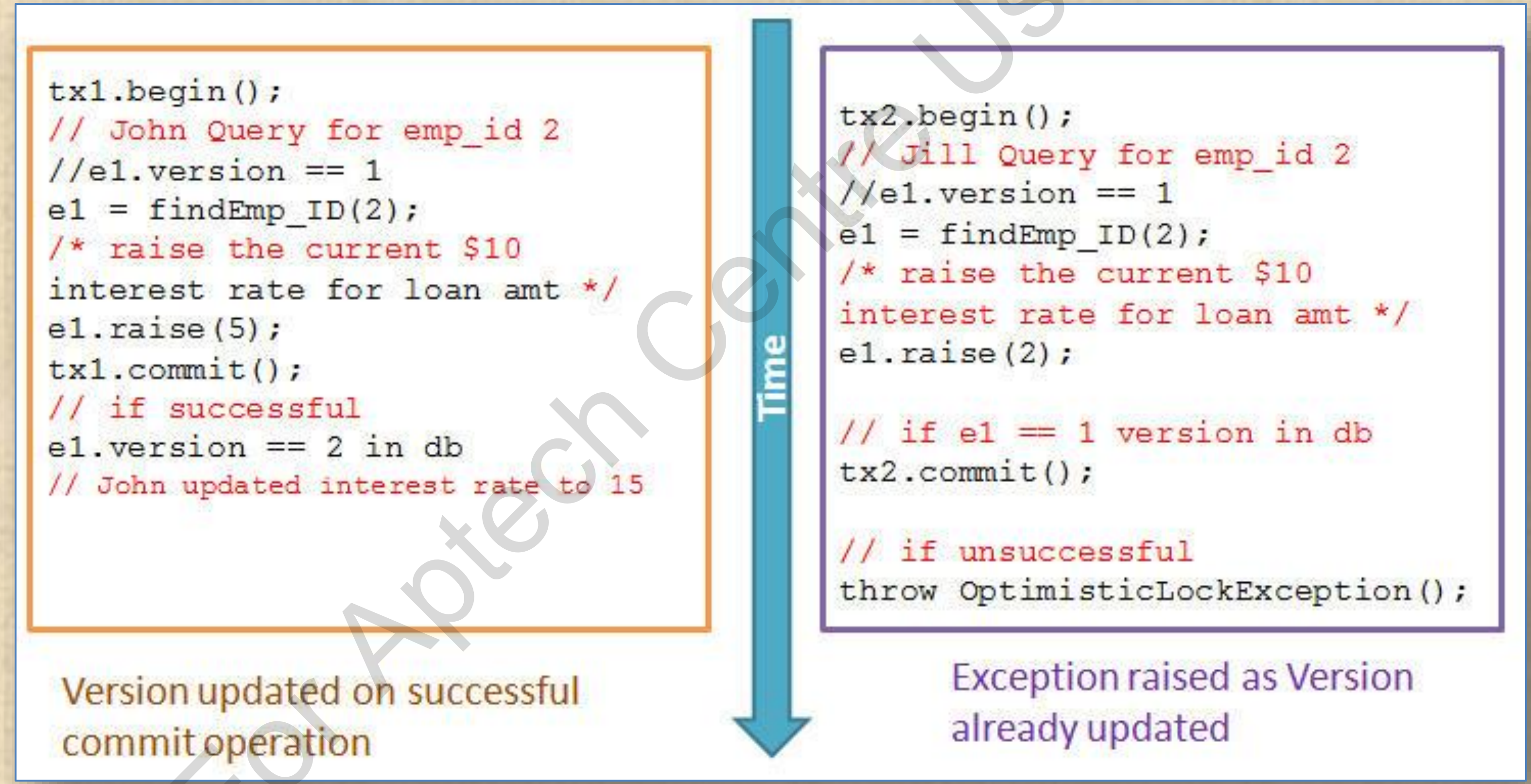
```
 public class Employee {
  @ID
  int id;

  @Version
  int version;
. . .
}
```

- The inclusion of version attribute marked with the annotation `@Version` instructs the persistence provider to check the entity instance for:
  - Any concurrent modifications and increments of the version attribute in each update operation.

Concurrent Transactions

❑ Following figure shows the example of two concurrent transactions trying to update the `Employee` entity:



```
tx1.begin();
// John Query for emp_id 2
//e1.version == 1
e1 = findEmp_ID(2);
/* raise the current $10
interest rate for loan amt */
e1.raise(5);
tx1.commit();
// if successful
e1.version == 2 in db
// John updated interest rate to 15
```

Version updated on successful commit operation

```
tx2.begin();
// Jill Query for emp_id 2
//e1.version == 1
e1 = findEmp_ID(2);
/* raise the current $10
interest rate for loan amt */
e1.raise(2);

// if e1 == 1 version in db
tx2.commit();

// if unsuccessful
throw OptimisticLockException();
```

Exception raised as Version already updated

Pessimistic Locking 1-3

- Pessimistic locking acquires locks extensively on entity data.
- Pessimistic locking techniques ensure that:
  - No other transaction can read or update the entity instance until the current transaction is committed.
  - If the pessimistic lock is applied as an exclusive lock, then only the holding transaction can update or delete.
- A pessimistic lock mode can be set to:
  - `PESSIMISTIC_READ`
  - `PESSIMISTIC_WRITE`
  - `PESSIMISTIC_FORCE_INCREMENT`

Pessimistic Locking 2-3

- An application cannot wait indefinitely to acquire a pessimistic lock.
- A timeout period is set through `javax.persistence.lock.timeout` property.
- The lock timeout property can be set by the `EntityManager`.
- `Query.setLockMode` and `TypedQuery.setLockMode` methods are used to set the timeout.

Pessimistic Locking 3-3

Following is the order followed in determining the timeout period:

- Timeout period set through instances of `EntityManager` and `Query`
- Timeout period set through @`NamedQuery` annotation
- Timeout period set through `createEntityManagerFactory`()
- Timeout period defined in the deployment descriptor

Lock Modes 1-6

- JPA provides lock modes which are layered on the **`Version`** attribute.
- Following table shows various lock modes provided by JPA:

Locking Mode	Description
`OPTIMISTIC`	This mode acquires optimistic read locks on all the entities used by the application with corresponding version attributes.
`OPTIMISTIC_FORCE_INCREMENT`	This mode acquires optimistic read locks on all the entities required by the application and version attributes. It force increments on the version attribute value.
`PESSIMISITIC_READ`	This mode acquires a long term read lock on the data accessed by the application. This lock is acquired to prevent data to be accessed or modified by other applications. Other applications can read this data when the pessimistic read lock is held on the entity data. The pessimistic read lock can be upgraded to a pessimistic write lock.

Lock Modes 2-6

Locking Mode	Description
PESSIMISTIC_WRITE	Pessimistic write lock is a long term write lock acquired on entity data. This lock prevents other applications from both reading and writing the locked entity data.
PESSIMISTIC_FORCE_INCREMENT	This mode of lock obtains a long term lock on the entity data and increments the value of the version attribute. This lock prevents the data from being modified by other applications.
READ	This mode represents an acquired optimistic read lock.
WRITE	This mode represents an optimistic write lock acquired on the entity data. It forces to increment the version attribute.
NONE	No locking mode is used on the database.

Lock Modes 3-6

Locking can be specified at multiple levels using the following methods:

- `find()`, `refresh()` and `lock()` methods of Entity Manager
- `setLockMode()` method of `Query`
- `lockMode` attribute of `@NamedQuery` annotation

Lock Modes 4-6

- Following code snippet shows the usage of **`lock()`** method:

```
. . .
EntityManager em = ...;
Customer c = ...;
em.lock(c, LockModeType.OPTIMISTIC);
. . .
```

- Following code snippet shows the usage of **`find()`** method:

```
EntityManager em = ...;
String customerPK = ...;
Customer C = em.find(Customer.class, customerPK,
LockModeType.PESSIMISTIC_WRITE);
. . .
```

Lock Modes 5-6

- Following code snippet shows the usage of **`refresh()`** method to modify the currently existing lock:

```
EntityManager em = ...;
String customerPK = ...;
Customer C = em.find(Customer.class, customerPK);
….
em.refresh(C,
LockModeType.OPTIMISTIC_FORCE_INCREMENT);
```

Lock Modes 6-6

❑ Following code snippet demonstrates the usage of **`setLockMode()`** method:

```
. . .
Query q = em.createQuery(...);
q.setLockMode(LockModeType.OPTIMISTIC);
. . .
```

❑ A lock mode element can be added to **`@NamedQuery`** annotation:

```
@NamedQuery(name="lockCustomerQuery",
query="SELECT c FROM Customer c WHERE c.name
LIKE :name",
lockMode=OPTIMISTIC_FORCE_INCREMENT);
```

Entity Callbacks and Listeners

- An entity contains a predefined set of lifecycle events that are triggered on the execution of methods called from `EntityManager` or `Query` API.

JPA allows developers to setup callback methods on entity classes.

- They are invoked when the entity instances are notified about the events.

Developers can also register separate listener classes on the callback events.

- Entity listener class methods are invoked in response to life cycle events on an entity.

Entity Callback Events

- Callback methods are defined within the entity class.
- Following code snippet shows the implementation of callback methods in an entity class:

```
@Entity
public static class MyEntity {
    @PrePersist void onPrePersist() {}
    @PostPersist void onPostPersist() {}
    @PostLoad void onPostLoad() {}
    @PreUpdate void onPreUpdate() {}
    @PostUpdate void onPostUpdate() {}
    @PreRemove void onPreRemove() {}
    @PostRemove void onPostRemove() {}
}
```

Lifecycle Callback Methods 1-2

- Following code snippet shows the usage of lifecycle callback methods:

```
@Entity
@EntityListeners(com.acme.AlertMonitor.class)
public class Account {
Long accountId;
Integer balance;
boolean preferred;
@Id
public Long getAccountId() { ... }
...
public Integer getBalance() { ... }
...
public void deposit(Integer amount) { ... }
public Integer withdraw(Integer amount) throws NSFException
{... }
….
```

Lifecycle Callback Methods 2-2

```
@PrePersist
protected void validateCreate() {
if (getBalance() < MIN_REQUIRED_BALANCE)
throw new AccountException("Insufficient balance to open an
account");
}
@PostLoad
protected void adjustPreferredStatus() {
preferred = (getBalance() >=
AccountManager.getPreferredStatusLevel());
}
}
 public class AlertMonitor {
@PostPersist
public void newAccountAlert(Account acct) {
Alerts.sendMarketingInfo(acct.getAccountId(), acct.getBalance());
}
}
```

Entity Listeners 1-3

- Entity listeners are classes that are intercept entity callback events.
- They are external classes that are attached to the entity class through annotations.
- Following code snippet shows the listener class annotated with the lifecycle callback methods:

```
public class Auditor {
@PostPersist
void postInsert(final Object entity)
{
   System.out.println("Inserted entity: " +
   entity.getClass().getName( ));
}
. . .
```

Entity Listeners 2-3

```
 @PostLoad
 void postLoad(final Object entity)
 {
  System.out.println("Loaded entity: " +
  entity.getClass().getName( ));
 }
}
. . .
```

- Declares the class `Auditor` that defines two methods namely, `PostInsert()` and `PostLoad()`.
- Both the methods are annotated with the lifecycle callback event annotations.

Entity Listeners 3-3

- Following code snippet shows the listener class that can be applied using `@javax.persistence.EntityListeners`:

```
@Entity
@EntityListeners ({Auditor.class})
public class EntityListenerEmployee
{
    ...
}
```

- By using the `@EntityListeners` annotation on the `EntityListenerEmployee` entity class, any callback methods within those entity listener classes will be invoked,

Default Entity Listeners

- Are a set of default entity listeners that are applied to every entity class in the persistence unit by using **`<entity-listeners>`** and **`<entity-mappings>`**.
- Following code snippet shows the default entity listener:

```
<entity-mappings>
<entity-listeners>
   <entity-listener class="com.listener.Auditor">
     <post-persist name="postInsert"/>
     <post-load name="postLoad"/>
   </entity-listener>
</entity-listeners>
</entity-mappings>
```

Invocation Order of Callback Methods

Following are the rules which define the invocation order of callback methods:

- Callback methods on listener classes are invoked before the callback methods of entity classes.
- Default listeners are handled before the listeners of the top level entity class listeners until listeners of the actual entity class.
- Internal callback methods are invoked starting at the top level entity class down the hierarchy until the callback methods in the actual entity class are invoked.

Using Second Level Cache in Persistence Applications 1-4

- Caching is a technique used to improve application performance.
- Caching optimizes the execution of expensive database calls.
- Caching can be implemented in applications at two levels:
  - First level cache – also known as **`EntityManager`** cache
  - Second level cache – can span across multiple **`EntityManagers`**.
- Applications use **`javax.persistence.Cache`** interface to manage the second level cache.

Using Second Level Cache in Persistence Applications 2-4

- Following table shows the cache modes defined in JPA:

Cache Mode Setting	Description
`ALL`	All the data accessed from the persistence provider is stored in the second level cache.
`NONE`	None of the data accessed from the persistence provider is stored in the second level cache.
`ENABLE_SELECTIVE`	This enables caching of the entities which are explicitly annotated with `@Cacheable` annotation.
`DISABLE_SELECTIVE`	This enables caching of all the entities except those entities which are explicitly annotated with `@Cacheable(false)` annotation.
`UNSPECIFIED`	When the cache mode is not specified then the caching behavior is set to the default behavior of the persistence provider.

Using Second Level Cache in Persistence Applications 4-4

- Applications can define whether the entity data can be cached or not through the `javax.persistence.Cacheable` annotation.
- Following code snippet demonstrates the usage of `Cacheable` annotation:

```
@Cacheable(true)
@Entity
public class Customer{ ... }

....
@Cacheable(false)
@Entity
public class Loan_Application{ ... }
```

Specifying the Cache Mode Settings

- Cache mode settings are done through the deployment descriptor.
- Following code snippet shows how to define the cache mode in the deployment descriptor:

```
. . .
<persistence-unit name="BankPU" transaction-
type="JTA">
<provider>org.eclipse.persistence.jpa.PersistenceProvi
der</provider>
<jta-data-source>java:comp/DefaultDataSource</jta-
data-source>
<shared-cache-mode>ENABLE_SELECTIVE</shared-cache-
mode>
</persistence-unit>
. . .
```

Specifying the Cache Retrieval and Store Mode 1-3

- Developer has to set **`javax.persistence.cache.retrieveMode`** and **`javax.persistence.cache.storeMode`** through the **`EntityManager`**.
- Cache retrieval mode is an enumerated type with values **`USE`** and **`BYPASS`**.
- Cache store mode defines how data is stored on the database.
- Cache store mode can assume one of the three values – **`USE`**, **`BYPASS`**, and **`REFRESH`**.

Specifying the Cache Retrieval and Store Mode 2-3

- Following code snippet demonstrates how to set cache retrieval or store mode:

```
EntityManager em = ...;
em.setProperty("javax.persistence.cache.storeMode",
"BYPASS");
```

- The cache mode can also be set through the **`Query`** and **`TypedQuery`** objects through **`setHint()`** method.

Specifying the Cache Retrieval and Store Mode 3-3

- Following code snippet demonstrates setting cache mode through **`Query`** and **`TypedQuery`** classes:

```
EntityManager em = ...;
CriteriaQuery<Customer> cq = ...;
TypedQuery<Customer> q = em.createQuery(cq);
q.setHint("javax.persistence.cache.storeMode",
"REFRESH");
```

Controlling the Cache Settings Programmatically 1-8

- Developers can use methods defined in **`javax.persistence.Cache`** to define the caching behavior of the application.

The **`Cache`** interface has the methods defined for following tasks:

- To check if a given entity has cached data or not
- To remove a particular entity from the cache
- To remove instances of a given entity class from the cache
- To clear cache and remove all entity data from it

Controlling the Cache Settings Programmatically 2-8

- Following are methods used in **`Cache`** interface:
  - **`contains()`**
  - **`evict()`**
  - **`evictAll()`**
- Following code snippet shows the entity class which is accessed through cache:

```
package data;
. . .
@Entity
@Table(name = "EMPLOYEE")
@Cacheable
@XmlRootElement
```

Controlling the Cache Settings Programmatically 3-8

```
public class Employee implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @NotNull
    @Column(name = "EMP_ID")
    private Integer empId;
    @Size(max = 15)
    @Column(name = "ENAME")
    private String ename;
    @Column(name = "SALARY")
    private Double salary;
```

Controlling the Cache Settings Programmatically 4-8

```
    public Employee() {
    }

    public Employee(Integer empId, String name,
Double sal) {
        this.empId = empId;
        this.ename = name;
        this.salary = sal;
    }

    public Integer getEmpId() {
        return empId;
    }

    public void setEmpId(Integer empId) {
        this.empId = empId;
    }
```

Controlling the Cache Settings Programmatically 5-8

```
 public String getEname() {
         return ename;
     }
     public void setEname(String ename) {
         this.ename = ename;
     }
     public Double getSalary() {
         return salary;
     }
     public void setSalary(Double salary) {
         this.salary = salary;
     }
     @Override
     public int hashCode() {
         int hash = 0;
         hash += (empId != null ?
empId.hashCode(): 0);
         return hash;
     }
```

Controlling the Cache Settings Programmatically 6-8

```
 @Override
    public boolean equals(Object object) {
        /* TODO: Warning - this method won't work in the case
the id fields are not set */
        if (!(object instanceof Employee)) {
            return false;
        }
        Employee other = (Employee) object;
        if ((this.empId == null && other.empId != null) ||
(this.empId != null && !this.empId.equals(other.empId))) {
            return false;
        }
        return true;
    }
    @Override
    public String toString() {
        return "data.Employee[ empId=" + empId + " ]";
    }
}
```

Controlling the Cache Settings Programmatically 7-8

- Following code snippet shows how an entity class is accessed through cache:

```
package access;
. . .
import javax.persistence.Persistence;
import javax.persistence.PersistenceContext;
@Stateful
@LocalBean
@Cacheable
public class AccessEmployee {
 @PersistenceContext (unitName = "CacheAccessPU")
  public EntityManagerFactory emf =
Persistence.createEntityManagerFactory("CacheAccessP
U");
```

Controlling the Cache Settings Programmatically 8-8

```
 public EntityManager em;
  Employee E = new Employee(101,"Alex",25000D);
  Cache C = emf.getCache();
   public void store(){
       String out = E.getEname();
         System.out.println("Here "+ out);
           if( C.contains(Employee.class, out))
              {
                    System.out.println("In cache");
              }

  }
  public static void main(String args[]){

      AccessEmployee A = new AccessEmployee();
      A.store();
  }
}
```

Summary

- ❑ Concurrent access to the database is always protected with the transaction isolation techniques.
- ❑ Database providers provide locking techniques to maintain data integrity. In addition to that, persistence providers also provide delay database writes, until the end of the transaction.
- ❑ Optimistic locking technique does not acquire locks for providing concurrency control, while performing any operation on certain data.
- ❑ Optimistic locking can be done either by annotating the entity field with javax.persistence. Version annotation or specify the version attribute in the XML descriptor file.
- ❑ Sometimes, it is desirable to acquire the database locks for a long-terms on entities. Such immediate obtained database locks are referred to as ‘pessimistic’ locks.
- ❑ JPA provides lock modes which are layered on the top of the @Version annotation provided in the optimistic and pessimistic locking.
- ❑ An entity contains a predefined set of lifecycle events that are triggered on the execution of methods called from EntityManager or Query API.
- ❑ The entity listeners are classes that are intercept entity callback events.
- ❑ You can specify a set of default entity listeners that are applied to every entity class in the persistence unit.
- ❑ Caching is a technique used by enterprise applications to improve the application performance.
- ❑ First level caching stores the data in the cache for the duration of the transaction or application request.
- ❑ Second level caching can span across multiple transactions and EntityManagers to improve application performance.
- ❑ Developers can use methods defined in javax.persistence.Cache to define the caching behavior of the application.