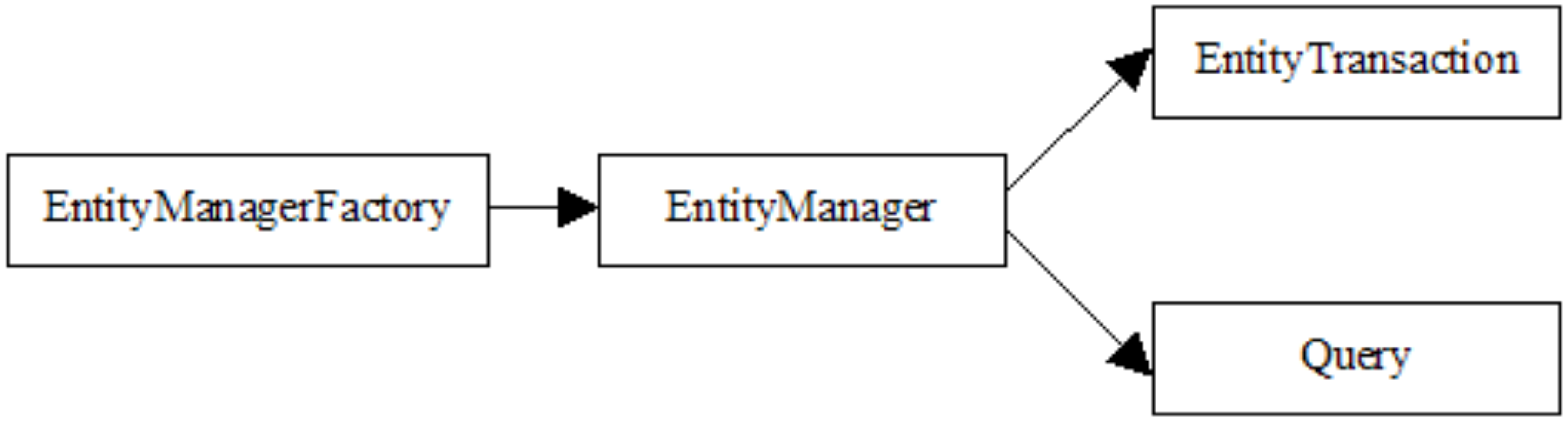
JPA - drugi deo

Database Connection using JPA

Working with the Java Persistence API (JPA) consists of using the following interfaces:



Database Connection using JPA

Overview

A connection to a database is represented by an `EntityManager` instance, which also provides functionality for performing operations on a database. Many applications require multiple database connections during their lifetime. For instance, in a web application it is common to establish a separate database connection, using a separate `EntityManager` instance, for every HTTP request.

The main role of an `EntityManagerFactory` instance is to support instantiation of `EntityManager` instances. An `EntityManagerFactory` is constructed for a specific database, and by managing resources efficiently (e.g. a pool of sockets), provides an efficient way to construct multiple `EntityManager` instances for that database. The instantiation of the `EntityManagerFactory` itself might be less efficient, but it is a one time operation. Once constructed, it can serve the entire application.

Operations that modify the content of a database require active transactions. Transactions are managed by an `EntityTransaction` instance obtained from the `EntityManager`.

An `EntityManager` instance also functions as a factory for `Query` instances, which are needed for executing queries on the database.

Every JPA implementation defines classes that implement these interfaces.

Database Connection using JPA

EntityManagerFactory

An EntityManagerFactory instance is obtained by using a static factory method of the JPA bootstrap class, Persistence:

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("myDbFile.odb");
```

Another form of the createEntityManagerFactory method takes a map of persistence unit properties as a second parameter:

```
Map<String, String> properties = new HashMap<String, String>();
properties.put("javax.persistence.jdbc.user", "admin");
properties.put("javax.persistence.jdbc.password", "admin");
EntityManagerFactory emf = Persistence.createEntityManagerFactory(
    "objectdb://localhost:6136/myDbFile.odb", properties);
```

The EntityManagerFactory instance, when constructed, opens the database.
When the application is finished using the EntityManagerFactory it has to be closed:

```
emf.close();
```

Database Connection using JPA

Connection URL

The `createEntityManagerFactory` method takes as an argument a name of a persistence unit.

Connection URL Parameters

The following parameters are supported as part of an connection url:

- user - for specifying a username in client server mode.
- password - for specifying a user password in client server mode.
- drop - for deleting any existing database content (useful for tests).

To connect to an server registered username and password have to be specified:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory(
    “objectdb://localhost/myDbFile.odb;user=admin;password=admin");
```

This is equivalent to specifying a username and a password in the persistence unit or in a map of properties (as demonstrated above).
To obtain a connection to an empty database (discarding existing content if any) the drop parameter has to be specified:

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("objectdb:myDbFile.tmp;drop");
```

Getting an empty clean database easily is very useful in tests. However, to avoid the risk of losing data - the drop parameter is ignored.

Database Connection using JPA

EntityManager

```
EntityManager em = emf.createEntityManager();
try {
    // TODO: Use the EntityManager to access the database
}
finally {
    em.close();
}
```

The EntityManager instance is obtained from the owning EntityManagerFactory instance. Calling the close method is essential to release resources (such as a socket in client-server mode) back to the owning EntityManagerFactory.
EntityManagerFactory defines another method for instantiation of EntityManager that, like the factory, takes a map of properties as an argument. This form is useful when a user name and a password other than the EntityManagerFactory's default user name and password have to specified:

```
 Map<String, String> properties = new HashMap<String, String>();
properties.put("javax.persistence.jdbc.user", "user1");
properties.put("javax.persistence.jdbc.password", "user1pwd");
EntityManager em = emf.createEntityManager(properties);
```

Database Connection using JPA

EntityTransaction

Operations that affect the content of the database (store, update, delete) must be performed within an active transaction. The EntityTransaction interface represents and manages database transactions. Every EntityManager holds a single attached EntityTransaction instance that is available via the getTransaction method:

```
try {
    em.getTransaction().begin();
    // Operations that modify the database should come here.
    em.getTransaction().commit();
}
finally {
    if (em.getTransaction().isActive())
        em.getTransaction().rollback();
}
```

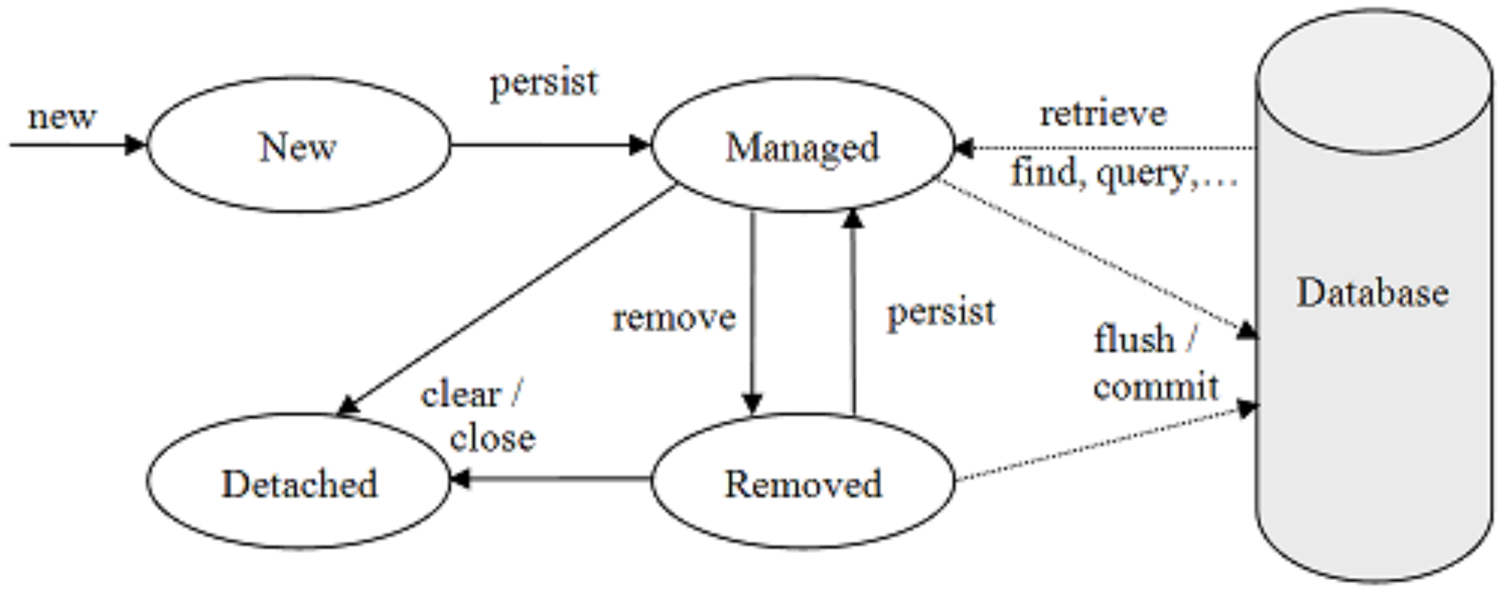
A transaction is started by a call to begin and ended by a call to either commit or rollback. All the operations on the database within these boundaries are associated with that transaction and are kept in memory until the transaction is ended. If the transaction is ended with a rollback, all the modifications to the database are discarded. However, by default, the in-memory instance of the managed entity is not affected by the rollback and is not returned to its pre-modified state.
Ending a transaction with a commit propagates all the modifications physically to the database. If for any reason a commit fails, the transaction is rolled back automatically (including rolling back modifications that have already been propagated to the database prior to the failure) and a RollbackException is thrown.

Working with JPA Entity Objects

Entity objects are in-memory instances of entity classes (persistable user defined classes), which can represent physical objects in the database.

Entity Object Life Cycle

The life cycle of entity objects consists of four states: New, Managed, Removed and Detached.



When an entity object is initially created its state is **New**. In this state the object is not yet associated with an EntityManager and has no representation in the database.

Working with JPA Entity Objects

An entity object becomes **Managed** when it is persisted to the database via an EntityManager's persist method, which must be invoked within an active transaction. On transaction commit, the owning EntityManager stores the new entity object to the database. More details on storing objects are provided in the Storing Entities section.
Entity objects retrieved from the database by an EntityManager are also in the **Managed** state. Object retrieval is discussed in more detail in the Retrieving Entities section.
If a managed entity object is modified within an active transaction the change is detected by the owning EntityManager and the update is propagated to the database on transaction commit. See the Updating Entities section for more information about making changes to entities.
A managed entity object can also be retrieved from the database and marked for deletion, by using the EntityManager's remove method within an active transaction. The entity object changes its state from Managed to **Removed**, and is physically deleted from the database during commit. More details on object deletion are provided in the Deleting Entities section.
The last state, **Detached**, represents entity objects that have been disconnected from the EntityManager. For instance, all the managed objects of an EntityManager become detached when the EntityManager is closed. Working with detached objects, including merging them back to an EntityManager, is discussed in the Detached Entities section.

Working with JPA Entity Objects

The Persistence Context

The persistence context is the collection of all the managed objects of an `EntityManager`. If an entity object that has to be retrieved already exists in the persistence context, the existing managed entity object is returned without actually accessing the database (except retrieval by `refresh`, which always requires accessing the database).
The main role of the persistence context is to make sure that a database entity object is represented by no more than one in-memory entity object within the same `EntityManager`. Every `EntityManager` manages its own persistence context. Therefore, a database object can be represented by different memory entity objects in different `EntityManager` instances. But retrieving the same database object more than once using the same `EntityManager` should always result in the same in-memory entity object.

Working with JPA Entity Objects

Another way of looking at it is that the persistence context also functions as a local cache for a given `EntityManager`. By default, managed entity objects that have not been modified or removed during a transaction are held in the persistence context by weak references. Therefore, when a managed entity object is no longer in use by the application the garbage collector can discard it and it is automatically removed from the persistence context.
The `contains` method can check if a specified entity object is in the persistence context:

```
    boolean isManaged = em.contains(employee);
```

The persistence context can be cleared by using the `clear` method, as so:

```
    em.clear();
```

When the persistence context is cleared all of its managed entities become detached and any changes to entity objects that have not been flushed to the database are discarded. Detached entity objects are discussed in more detail in the Detached Entities section.

CRUD Operations with JPA

The following subsections explain how to use JPA for CRUD database operations:

- Storing JPA Entity Objects
- Retrieving JPA Entity Objects
- Updating JPA Entity Objects
- Deleting JPA Entity Objects

Storing JPA Entity Objects

New entity objects can be stored in the database either explicitly by invoking the persist method or implicitly as a result of a cascade operation.

Explicit Persist

The following code stores an instance of the Employee entity class in the database:

```
Employee employee = new Employee("Samuel", "Joseph", "Wurzelbacher");
em.getTransaction().begin();
em.persist(employee);
em.getTransaction().commit();
```

The Employee instance is constructed as an ordinary Java object and its initial state is New. An explicit call to persist associates the object with an owner EntityManager em and changes its state to Managed. The new entity object is stored in the database when the transaction is committed.
An IllegalArgumentException is thrown by persist if the argument is not an instance of an entity class. Only instances of entity classes can be stored in the database independently. Objects of other persistable types can only be stored in the database embedded in containing entities (as field values).
A TransactionRequiredException is thrown if there is no active transaction when persist is called because operations that modify the database require an active transaction.
If the database already contains another entity of the same type with the same primary key, an EntityExistsException is thrown. The exception is thrown either by persist (if that existing entity object is currently managed by the EntityManager) or by commit.

Storing JPA Entity Objects

Referenced Embedded Objects

The following code stores an `Employee` instance with a reference to an `Address` instance:

```
  Employee employee = new Employee("Samuel", "Joseph", "Wurzelbacher");
  Address address = new Address("Holland", "Ohio");
  employee.setAddress(address);

  em.getTransaction().begin();
  em.persist(employee);
  em.getTransaction().commit();
```

Instances of persistable types other than entity classes are automatically stored embedded in containing entity objects. Therefore, if Address is defined as an embeddable class the `Employee` entity object is automatically stored in the database with its `Address` instance as an embedded object.
Notice that embedded objects cannot be shared by multiple entity objects. Each containing entity object should have its own embedded objects.

Storing JPA Entity Objects

Referenced Entity Objects

On the other hand, suppose that the `Address` class in the code above is defined as an entity class. In this case, the referenced `Address` instance is not stored in the database automatically with the referencing `Employee` instance.

To avoid a dangling reference in the database, an `IllegalStateException` is thrown on commit if a persisted entity object has to be stored in the database in a transaction and it references another entity object that is not expected to be stored in the database at the end of that transaction.

It is the application's responsibility to verify that when an object is stored in the database, the entire closure of entity objects that are reachable from that object by navigation through persistent reference fields is also stored in the database. This can be done either by explicit persist of every reachable object or alternatively by setting automatic cascading persist.

Storing JPA Entity Objects

Cascading Persist

Marking a reference field with CascadeType.PERSIST (or CascadeType.ALL that also covers PERSIST) indicates that persist operations should be cascaded automatically to entity objects that are referenced by that field (multiple entity objects can be referenced by a collection field):

```
@Entity
class Employee {
     :
    @OneToOne(cascade=CascadeType.PERSIST)
    private Address address;
     :
}
```

In the example above, the Employee entity class contains an *address* field that references an instance of Address, which is another entity class. Due to the CascadeType.PERSIST setting, when an Employee instance is persisted the operation is automatically cascaded to the referenced Address instance which is then automatically persisted without the need for a separate persist call for Address. Cascading may continue recursively when applicable (e.g. to entity objects that the Address object references, etc.).

Storing JPA Entity Objects

Global Cascading Persist

Instead of specifying `CascadeType.PERSIST` individually for every relevant reference field, it can be specified globally for any persistent reference in a JPA portable way, by specifying the `cascade-persist` XML element in the XML mapping file:

```xml
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
 http://java.sun.com/xml/ns/persistence/orm_1_0.xsd" version="1.0">
   <persistence-unit-metadata>
     <persistence-unit-defaults>
       <cascade-persist/>
     </persistence-unit-defaults>
   </persistence-unit-metadata>
</entity-mappings>
```

The mapping file has to be located either in the default location, `META-INF/orm.xml`, or in another location that is specified explicitly in the persistence unit definition (in `persistence.xml`).

Storing JPA Entity Objects

Batch Store

Storing a large number of entity objects requires special consideration. The combination of the `clear` and `flush` methods can be used to save memory in large transactions:

```
em.getTransaction().begin();
for (int i = 1; i <= 1000000; i++) {
    Point point = new Point(i, i);
    em.persist(point);
    if ((i % 10000) == 0) {
        em.flush();
        em.clear();
    }
}
em.getTransaction().commit();
```

Managed entity objects consume more memory than ordinary non managed Java objects. Therefore, holding 1,000,000 managed `Point` instances in the persistence context might consume too much memory. The sample code above clears the persistence context after every 10,000 persists. Updates are flushed to the database before clearing, otherwise they would be lost.

Storing JPA Entity Objects

Updates that are sent to the database using `flush` are considered temporary and are only visible to the owner `EntityManager` until a commit. With no explicit `commit`, these updates are later discarded. The combination of `clear` and `flush` enables moving the temporary updates from memory to the database.
Note: Flushing updates to the database is sometimes also useful before executing queries in order to get up to date results.
Storing large amount of entity objects can also be performed by multiple transactions:

```
em.getTransaction().begin();
for (int i = 1; i <= 1000000; i++) {
    Point point = new Point(i, i);
    em.persist(point);
    if ((i % 10000) == 0) {
        em.getTransaction().commit();
        em.clear();
        em.getTransaction().begin();
    }
}
em.getTransaction().commit();
```

Splitting a batch store into multiple transactions is more efficient than using one transaction with multiple invocations of the `flush` and `clear` methods. So using multiple transactions is preferred when applicable.

Retrieving JPA Entity Objects

The Java Persistence API (JPA) provides various ways to retrieve objects from the database. The retrieval of objects does not require an active transaction because it does not change the content of the database.
The persistence context serves as a cache of retrieved entity objects. If a requested entity object is not found in the persistence context a new object is constructed and filled with data that is retrieved from the database (or from the L2 cache - if enabled). The new entity object is then added to the persistence context as a managed entity object and returned to the application.
Notice that construction of a new managed object during retrieval uses the no-arg constructor. Therefore, it is recommended to avoid unnecessary time consuming operations in no-arg constructors of entity classes and to keep them simple as possible.

Retrieving JPA Entity Objects

Retrieval by Class and Primary Key

Every entity object can be uniquely identified and retrieved by the combination of its class and its primary key. Given an `EntityManager` `em`, the following code fragment demonstrates retrieval of an `Employee` object whose primary key is `1`:

```
  Employee employee = em.find(Employee.class, 1);
```

Casting of the retrieved object to `Employee` is not required because `find` is defined as returning an instance of the same class that it takes as a first argument (using generics).
An `IllegalArgumentException` is thrown if the specified class is not an entity class.
If the `EntityManager` already manages the specified entity object in its persistence context no retrieval is required and the existing managed object is returned as is. Otherwise, the object data is retrieved from the database and a new managed entity object with that retrieved data is constructed and returned. If the object is not found in the database `null` is returned.

Retrieving JPA Entity Objects

A similar method, `getReference`, can be considered the lazy version of `find`:

```
  Employee employee = em.getReference(Employee.class, 1);
```

The `getReference` method works like the `find` method except that if the entity object is not already managed by the `EntityManager` a *hollow* object might be returned (`null` is never returned). A hollow object is initialized with the valid primary key but all its other persistent fields are uninitialized. The object content is retrieved from the database and the persistent fields are initialized, lazily, when the entity object is first accessed. If the requested object does not exist an `EntityNotFoundException` is thrown when the object is first accessed.
The `getReference` method is useful when a reference to an entity object is required but not its content, such as when a reference to it has to be set from another entity object.

Retrieving JPA Entity Objects

Retrieval by Eager Fetch

Retrieval of an entity object from the database might cause automatic retrieval of additional entity objects. By default, a retrieval operation is automatically cascaded through all the non collection and map persistent fields (i.e. through one-to-one and many-to-one relationships). Therefore, when an entity object is retrieved, all the entity objects that are reachable from it by navigation through non collection and map persistent fields are also retrieved. Theoretically, in some extreme situations this might cause the retrieval of the entire database into the memory, which is usually unacceptable.
A persistent reference field can be excluded from this automatic cascaded retrieval by using a lazy fetch type:

```
@Entity
class Employee {
    :
  @ManyToOne(fetch=FetchType.LAZY)
  private Employee manager;
    :
}
```

Retrieving JPA Entity Objects

The default for non collection and map references is `FetchType.EAGER`, indicating that the retrieval operation is cascaded through the field. Explicitly specifying `FetchType.LAZY` in either `@OneToOne` or `@ManyToOne` annotations excludes the field from participating in retrieval cascading.
When an entity object is retrieved all its persistent fields are initialized. A persistent reference field with the `FetchType.LAZY` fetch policy is initialized to reference a new managed hollow object (unless the referenced object is already managed by the `EntityManager`). In the example above, when an `Employee` instance is retrieved its `manager` field might reference a hollow `Employee` instance. In a hollow object the primary key is set but other persistent fields are uninitialized until the object fields are accessed.
On the other hand, the default fetch policy of persistent collection and map fields is `FetchType.LAZY`. Therefore, by default, when an entity object is retrieved any other entity objects that it references through its collection and map fields are not retrieved with it.

Retrieving JPA Entity Objects

This can be changed by an explicit FetchType.EAGER setting:

```
@Entity
class Employee {
    :
  @ManyToMany(fetch=FetchType.EAGER)
  private Collection<Project> projects;
    :
}
```

Specifying FetchType.EAGER explicitly in @OneToMany or @ManyToMany annotations enables cascading retrieval for the field. In the above example, when an Employee instance is retrieved all the referenced Project instances are also retrieved automatically.

Retrieving JPA Entity Objects

Retrieval by Navigation and Access

All the persistent fields of an entity object can be accessed freely, regardless of the current fetch policy, as long as the EntityManager is open. This also includes fields that reference entity objects that have not been loaded from the database yet and are represented by hollow objects. If the EntityManager is open when a hollow object is first accessed its content is automatically retrieved from the database and all its persistent fields are initialized.
From the point of view of the developer it looks like the entire graph of objects is present in memory. This illusion, which is based on lazy transparent activation and retrieval of objects, helps hide some of the direct interaction with the database and makes database programming easier.
For example, after retrieving an Employee instance from the database the manager field may include a hollow Employee entity object:

```
  Employee employee = em.find(Employee.class, 1);
  Employee managed = employee.getManager(); // might be hollow
```

Retrieving JPA Entity Objects

If *manager* is hollow transparent activation occurs when it is first accessed. For example:

```
String managerName = manager.getName();
```

Accessing a persistent field in a hollow object (e.g. the name of the *manager* in the example above) causes the retrieval of missing content from the database and initialization of all the persistent fields.
As seen, the entire graph of objects is available for navigation, regardless of the fetch policy. The fetch policy, however, does affect performance. Eager retrieval might minimize the round trips to the database and improve performance, but unnecessary retrieval of entity objects that are not in use will decrease performance.
The fetch policy also affects objects that become detached (e.g. when the EntityManager is closed). Transparent activation is not supported for detached objects. Therefore, only content that has already been fetched from the database is available in objects that are detached.
JPA 2 introduces methods for checking if a specified entity object or a specified persistent field is loaded. For example:

```
PersistenceUtil util = Persistence.getPersistenceUtil();
boolean isObjectLoaded = util.isLoaded(employee);
boolean isFieldLoaded = util.isLoaded(employee, “address");
```

As shown above, a `PersistenceUtil` instance is obtained from the static `getPersistenceUtil` method. It provides two `isLoaded` methods - one for checking an entity object and the other for checking a persistent field of an entity object.

Retrieving JPA Entity Objects

Retrieval by Query

The most flexible method for retrieving objects from the database is to use queries. The official query language of JPA is JPQL (Java Persistence Query Language). It enables retrieval of objects from the database by using simple queries as well as complex, sophisticated ones.

Retrieval by Refresh

Managed objects can be reloaded from the database by using the `refresh` method:

```
em.refresh(employee);
```

The content of the managed object in memory is discarded (including changes, if any) and replaced by data that is retrieved from the database. This might be useful to ensure that the application deals with the most up to date version of an entity object, just in case it might have been changed by another `EntityManager` since it was retrieved.
An `IllegalArgumentException` is thrown by `refresh` if the argument is not a managed entity (including entity objects in the New, Removed or Detached states). If the object does not exist in the database anymore an `EntityNotFoundException` is thrown.

Retrieving JPA Entity Objects

Cascading Refresh

Marking a reference field with CascadeType.REFRESH (or CascadeType.ALL, which includes REFRESH) indicates that refresh operations should be cascaded automatically to entity objects that are referenced by that field (multiple entity objects can be referenced by a collection field):

```
@Entity
class Employee {
     :
    @OneToOne(cascade=CascadeType.REFRESH)
    private Address address;
     :
}
```

In the example above, the Employee entity class contains an *address* field that references an instance of Address, which is another entity class. Due to the CascadeType.REFRESH setting, when an Employee instance is refreshed the operation is automatically cascaded to the referenced Address instance, which is then automatically refreshed as well. Cascading may continue recursively when applicable (e.g. to entity objects that the Address object references, if any).

Updating JPA Entity Objects

Modifying existing entity objects that are stored in the database is based on transparent persistence, which means that changes are detected and handled automatically.

Transparent Update

Once an entity object is retrieved from the database (no matter which way) it can simply be modified in memory from inside an active transaction:

```
  Employee employee = em.find(Employee.class, 1);

  em.getTransaction().begin();
  employee.setNickname("Joe the Plumber");
  em.getTransaction().commit();
```

The entity object is physically updated in the database when the transaction is committed. If the transaction is rolled back and not committed the update is discarded.
On commit the persist operation can be cascaded from all the entity objects that have to be stored in the database, including from all the modified entity objects. Therefore, entity objects that are referenced from modified entity objects by fields that are marked with `CascadeType.PERSIST` or `CascadeType.ALL` are also persisted. If global cascade persist is enabled all the reachable entity objects that are not managed yet are also persisted.

Updating JPA Entity Objects

Automatic Change Tracking

As shown above, an update is achieved by modifying a managed entity object from within an active transaction. No EntityManager's method is invoked to report the update. Therefore, to be able to apply database updates on commit, JPA must detect changes to managed entities automatically. One way to detect changes is to keep a snapshot of every managed object when it is retrieved from the database and to compare that snapshot to the actual managed object on commit. A more efficient way to detect changes automatically is described in the Enhancer section.
However, detecting changes to arrays requires using snapshots even if the entity classes are enhanced. Therefore, for efficiency purposes, the default behavior ignores array changes when using enhanced entity classes:

```
    Employee employee = em.find(Employee.class, 1);

    em.getTransaction().begin();
    employee.projects[0] = new Project(); // not detected automatically
    JDOHelper.makeDirty(employee, "projects"); // reported as dirty
    em.getTransaction().commit();
```

As demonstrated above, array changes are not detected automatically (by default) but it is possible to report a change explicitly by invoking the JDO's makeDirty method.
Alternatively, JPA can be configured to detect array changes using snapshots as well as when enhanced entity classes are in use.
It is usually recommended to use collections rather than arrays when using JPA. Collections are more portable to ORM JPA implementations and provide better automatic change tracking support.

Updating JPA Entity Objects

UPDATE Queries

UPDATE queries provide an alternative way for updating entity objects in the database. Modifying objects using an UPDATE query may be useful especially when many entity objects have to be modified in one operation.

Deleting JPA Entity Objects

Existing entity objects can be deleted from the database either explicitly by invoking the `remove` method or implicitly as a result of a cascade operation.

Explicit Remove

In order to delete an object from the database it has to first be retrieved (no matter which way) and then in an active transaction, it can be deleted using the `remove` method:

```
  Employee employee = em.find(Employee.class, 1);
  em.getTransaction().begin();
  em.remove(employee);
  em.getTransaction().commit();
```

The entity object is physically deleted from the database when the transaction is committed. Embedded objects that are contained in the entity object are also deleted. If the transaction is rolled back and not committed the object is not deleted.
An `IllegalArgumentException` is thrown by `remove` if the argument is not a an instance of an entity class or if it is a detached entity. A `TransactionRequiredException` is thrown if there is no active transaction when `remove` is called because operations that modify the database require an active transaction.

Deleting JPA Entity Objects

Cascading Remove

Marking a reference field with `CascadeType.REMOVE` (or `CascadeType.ALL`, which includes `REMOVE`) indicates that `remove` operations should be cascaded automatically to entity objects that are referenced by that field (multiple entity objects can be referenced by a collection field):

```
@Entity
class Employee {
     :
    @OneToOne(cascade=CascadeType.REMOVE)
    private Address address;
     :
}
```

In the example above, the `Employee` entity class contains an *address* field that references an instance of `Address`, which is another entity class. Due to the `CascadeType.REMOVE` setting, when an `Employee` instance is removed the operation is automatically cascaded to the referenced `Address` instance, which is then automatically removed as well. Cascading may continue recursively when applicable (e.g. to entity objects that the `Address` object references, if any).

Deleting JPA Entity Objects

Orphan Removal

JPA 2 supports an additional and more aggressive remove cascading mode which can be specified using the `orphanRemoval` element of the `@OneToOne` and `@OneToMany` annotations:

```
@Entity
class Employee {
     :
    @OneToOne(orphanRemoval=true)
    private Address address;
     :
}
```

When an `Employee` entity object is removed the remove operation is cascaded to the referenced `Address` entity object. In this regard, `orphanRemoval=true` and `cascade=CascadeType.REMOVE` are identical, and if `orphanRemoval=true` is specified, `CascadeType.REMOVE` is redundant.

Deleting JPA Entity Objects

The difference between the two settings is in the response to disconnecting a relationship. For example, such as when setting the address field to null or to another Address object.

- If orphanRemoval=true is specified the disconnected Address instance is automatically removed. This is useful for cleaning up dependent objects (e.g. Address) that should not exist without a reference from an owner object (e.g. Employee).
- If only cascade=CascadeType.REMOVE is specified no automatic action is taken since disconnecting a relationship is not a remove operation.

To avoid dangling references as a result of orphan removal this feature should only be enabled for fields that hold private non shared dependent objects.
Orphan removal can also be set for collection and map fields. For example:

```
@Entity
class Employee {
     :
    @OneToMany(orphanRemoval=true)
    private List<Address> addresses;
     :
}
```

In this case, removal of an Address object from the collection leads to automatic removal of that object from the database.

Deleting JPA Entity Objects

DELETE Queries

DELETE queries provide an alternative way for removing entity objects from the database. Deleting objects using a DELETE query may be useful especially when many entity objects have to be deleted in one operation.

Advanced JPA Topics

This section discusses advanced JPA topics:

- Detached Entity Objects
- Locking in JPA
- JPA Lifecycle Events
- Shared (L2) Entity Cache
- JPA Metamodel API

Detached Entity Objects

Detached entity objects are objects in a special state in which they are not managed by any `EntityManager` but still represent objects in the database. Compared to managed entity objects, detached objects are limited in functionality:

- Many JPA methods do not accept detached objects (e.g. `lock`).
- Retrieval by navigation from detached objects is not supported, so only persistent fields that have been loaded before detachment should be used.
- Changes to detached entity objects are not stored in the database unless modified detached objects are merged back into an `EntityManager` to become managed again.

Detached objects are useful in situations in which an `EntityManager` is not available and for transferring objects between different `EntityManager` instances.

Detached Entity Objects

Explicit Detach

When a managed entity object is serialized and then deserialized, the deserialized entity object (but not the original serialized object) is constructed as a detached entity object since is not associated with any `EntityManager`.
In addition, in JPA 2 we can detach an entity object by using the `detach` method:

```
em.detach(employee);
```

An `IllegalArgumentException` is thrown by `detach` if the argument is not an entity object.

Detached Entity Objects

Cascading Detach

Marking a reference field with CascadeType.DETACH (or CascadeType.ALL, which includes DETACH) indicates that detach operations should be cascaded automatically to entity objects that are referenced by that field (multiple entity objects can be referenced by a collection field):

```
@Entity
class Employee {
     :
    @OneToOne(cascade=CascadeType.DETACH)
    private Address address;
     :
}
```

In the example above, the Employee entity class contains an *address* field that references an instance of Address, which is another entity class. Due to the CascadeType.DETACH setting, when an Employee instance is detached the operation is automatically cascaded to the referenced Address instance, which is then automatically detached as well. Cascading may continue recursively when applicable (e.g. to entity objects that the Address object references, if any).

Detached Entity Objects

Bulk Detach

The following operations clear the entire `EntityManager`'s persistence context and detach all managed entity objects:

- Invocation of the close method, which closes an `EntityManager`.
- Invocation of the clear method, which clears an `EntityManager`'s persistence context.
- Rolling back a transaction - either by invocation of `rollback` or by a `commit` failure.

Detached Entity Objects

Explicit Merge

Detached objects can be attached to any EntityManager by using the `merge` method:

```
em.merge(employee);
```

The content of the specified detached entity object is copied into an existing managed entity object with the same identity (i.e. same type and primary key). If the `EntityManager` does not manage such an entity object yet a new managed entity object is constructed. The detached object itself, however, remains unchanged and detached.
An `IllegalArgumentException` is thrown by `merge` if the argument is not an instance of an entity class or it is a removed entity. A `TransactionRequiredException` is thrown if there is no active transaction when `merge` is called because operations that might modify the database require an active transaction.

Detached Entity Objects

Cascading Merge

Marking a reference field with CascadeType.MERGE (or CascadeType.ALL, which includes MERGE) indicates that merge operations should be cascaded automatically to entity objects that are referenced by that field (multiple entity objects can be referenced by a collection field):

```
@Entity
class Employee {
     :
    @OneToOne(cascade=CascadeType.MERGE)
    private Address address;
     :
}
```

In the example above, the Employee entity class contains an *address* field that references an instance of Address, which is another entity class. Due to the CascadeType.MERGE setting, when an Employee instance is merged the operation is automatically cascaded to the referenced Address instance, which is then automatically merged as well. Cascading may continue recursively when applicable (e.g. to entity objects that the Address object references, if any).

Locking in JPA

JPA 2 supports both **optimistic locking** and **pessimistic locking**. Locking is essential to avoid update collisions resulting from simultaneous updates to the same data by two concurrent users. Locking in JPA is always at the database object level, i.e. each database object is locked separately.

Optimistic locking is applied on transaction commit. Any database object that has to be updated or deleted is checked. An exception is thrown if it is found out that an update is being performed on an old version of a database object, for which another update has already been committed by another transaction.

Optimistic locking should be the first choice for most applications, since compared to pessimistic locking it is easier to use and more efficient.

In the rare cases in which update collision must be revealed earlier (before transaction commit) **pessimistic locking** can be used. When using pessimistic locking, database objects are locked during the transaction and lock conflicts, if they happen, are detected earlier.

Locking in JPA

Optimistic Locking

JPA maintains a version number for every entity object. The initial version of a new entity object (when it is stored in the database for the first time) is 1. In every transaction in which an entity object is modified its version number is automatically increased by one. Version numbers are managed internally but can be exposed by defining a version field.
During `commit` (and `flush`), JPA checks every database object that has to be updated or deleted, and compares the version number of that object in the database to the version number of the in-memory object being updated. The transaction fails and an `OptimisticLockException` is thrown if the version numbers do not match, indicating that the object has been modified by another user (using another `EntityManager`) since it was retrieved by the current updater.

Locking in JPA

Pessimistic Locking

The main supported pessimistic lock modes are:

- `PESSIMISTIC_READ` - which represents a shared lock.
- `PESSIMISTIC_WRITE` - which represents an exclusive lock.

Setting a Pessimistic Lock

An entity object can be locked explicitly by the `lock` method:

```
em.lock(employee, LockModeType.PESSIMISTIC_WRITE);
```

The first argument is an entity object. The second argument is the requested lock mode.
A `TransactionRequiredException` is thrown if there is no active transaction when `lock` is called because explicit locking requires an active transaction.

Locking in JPA

A LockTimeoutException is thrown if the requested pessimistic lock cannot be granted:

- A PESSIMISTIC_READ lock request fails if another user (which is represented by another EntityManager instance) currently holds a PESSIMISTIC_WRITE lock on that database object.
- A PESSIMISTIC_WRITE lock request fails if another user currently holds either a PESSIMISTIC_WRITE lock or a PESSIMISTIC_READ lock on that database object.

For example, consider the following code fragment:

```
em1.lock(e1, lockMode1);
em2.lock(e2, lockMode2);
```

em1 and em2 are two EntityManager instances that manage the same Employee database object, which is referenced as e1 by em1 and as e2 by em2 (notice that e1 and e2 are two in-memory entity objects that represent one database object).
If both lockMode1 and lockMode2 are PESSIMISTIC_READ - these lock requests should succeed. Any other combination of pessimistic lock modes, which also includes PESSIMISTIC_WRITE, will cause a LockTimeoutException (on the second lock request).

Locking in JPA

Pessimistic Lock Timeout

By default, when a pessimistic lock conflict occurs a `LockTimeoutException` is thrown immediately. The "`javax.persistence.lock.timeout`" hint can be set to allow waiting for a pessimistic lock for a specified number of milliseconds. The hint can be set in several scopes: For the entire persistence unit - using a `persistence.xml` property:

```
    <properties>
       <property name="javax.persistence.lock.timeout" value="1000"/>
    </properties>
```

For an `EntityManagerFactory` - using the `createEntityManagerFacotory` method:

```
  Map<String,Object> properties = new HashMap();
  properties.put("javax.persistence.lock.timeout", 2000);
  EntityManagerFactory emf =
      Persistence.createEntityManagerFactory("pu", properties);
```

Locking in JPA

For an `EntityManager` - using the `createEntityManager` method:

```
Map<String,Object> properties = new HashMap();
properties.put("javax.persistence.lock.timeout", 3000);
EntityManager em = emf.createEntityManager(properties);
```

or using the `setProperty` method:

```
em.setProperty("javax.persistence.lock.timeout", 4000);
```

In addition, the hint can be set for a specific retrieval operation or query.

Releasing a Pessimistic Lock

Pessimistic locks are automatically released at transaction end (using either `commit` or `rollback`).
JPA supports also releasing a lock explicitly while the transaction is active, as so:

```
em.lock(employee, LockModeType.NONE);
```

Locking in JPA

Other Explicit Lock Modes

In addition to the two main pessimistic modes (`PESSIMISTIC_WRITE` *and* `PESSIMISTIC_READ`, which are discussed above), JPA defines additional lock modes that can also be specified as arguments for the `lock` method to obtain special effects:

- `OPTIMISTIC` (formerly `READ`)
- `OPTIMISTIC_FORCE_INCREMENT` (formerly `WRITE`)
- `PESSIMISTIC_FORCE_INCREMENT`

Since optimistic locking is applied automatically by JPA to every entity object, the `OPTIMISTIC` lock mode has no effect and, if specified, is silently ignored by JPA.
The `OPTIMISTIC_FORCE_INCREMENT` mode affects only clean (non dirty) entity objects. Explicit lock at that mode marks the clean entity object as modified (dirty) and increases its version number by 1.
The `PESSIMISTIC_FORCE_INCREMENT` mode is equivalent to the `PESSIMISTIC_WRITE` mode with the addition that it marks a clean entity object as dirty and increases its version number by one (i.e. it combines `PESSIMISTIC_WRITE` with `OPTIMISTIC_FORCE_INCREMENT`).

Locking in JPA

Locking during Retrieval

JPA 2 provides various methods for locking entity objects when they are retrieved from the database. In addition to improving efficiency (relative to a retrieval followed by a separate lock), these methods perform retrieval and locking as one atomic operation.
For example, the `find` method has a form that accepts a lock mode:

```
Employee employee = em.find(
    Employee.class, 1, LockModeType.PESSIMISTIC_WRITE);
```

Similarly, the `refresh` method can also receive a lock mode:

```
em.refresh(employee, LockModeType.PESSIMISTIC_WRITE);
```

Locking in JPA

A lock mode can also be set for a query in order to lock all the query result objects.
When a retrieval operation includes pessimistic locking, timeout can be specified as a property.
For example:

```
Map<String,Object> properties = new HashMap();
properties.put("javax.persistence.lock.timeout", 2000);

Employee employee = em.find(
    Employee.class, 1, LockModeType.PESSIMISTIC_WRITE, properties);

...

em.refresh(employee, LockModeType.PESSIMISTIC_WRITE, properties);
```

Setting timeout at the operation level overrides setting in higher scopes.

JPA Lifecycle Events

Callback methods are user defined methods that are attached to entity lifecycle events and are invoked automatically by JPA when these events occur.

Internal Callback Methods

Internal callback methods are methods that are defined within an entity class. For example, the following entity class defines all the supported callback methods with empty implementations:

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

Internal callback methods should always return `void` and take no arguments. They can have any name and any access level (`public`, `protected`, `package` and `private`) but should not be `static`.

JPA Lifecycle Events

The annotation specifies when the callback method is invoked:

- `@PrePersist` - before a new entity is persisted (added to the `EntityManager`).
- `@PostPersist` - after storing a new entity in the database (during `commit` or `flush`).
- `@PostLoad` - after an entity has been retrieved from the database.
- `@PreUpdate` - when an entity is identified as modified by the `EntityManager`.
- `@PostUpdate` - after updating an entity in the database (during `commit` or `flush`).
- `@PreRemove` - when an entity is marked for removal in the EntityManager.
- `@PostRemove` - after deleting an entity from the database (during `commit` or `flush`).

An entity class may include callback methods for any subset or combination of lifecycle events but no more than one callback method for the same event. However, the same method may be used for multiple callback events by marking it with more than one annotation.
By default, a callback method in a super entity class is also invoked for entity objects of the subclasses unless that callback method is overridden by the subclass.

JPA Lifecycle Events

Implementation Restrictions

To avoid conflicts with the original database operation that fires the entity lifecycle event (which is still in progress) callback methods should not call `EntityManager` or `Query` methods and should not access any other entity objects.
If a callback method throws an exception within an active transaction, the transaction is marked for rollback and no more callback methods are invoked for that operation.

Listeners and External Callback Methods

External callback methods are defined outside entity classes in a special listener class:

```
public class MyListener {
    @PrePersist void onPrePersist(Object o) {}
    @PostPersist void onPostPersist(Object o) {}
    @PostLoad void onPostLoad(Object o) {}
    @PreUpdate void onPreUpdate(Object o) {}
    @PostUpdate void onPostUpdate(Object o) {}
    @PreRemove void onPreRemove(Object o) {}
    @PostRemove void onPostRemove(Object o) {}
}
```

External callback methods (in a listener class) should always return `void` and take one argument that specifies the entity which is the source of the lifecycle event. The argument can have any type that matches the actual value (e.g. in the code above, `Object` can be replaced by a more specific type). The listener class should be stateless and should have a public no-arg constructor (or no constructor at all) to enable automatic instantiation.

JPA Lifecycle Events

The listener class is attached to the entity class using the `@EntityListeners` annotation:

```
@Entity @EntityListeners(MyListener.class)
public class MyEntityWithListener {
}
```

Multiple listener classes can also be attached to one entity class:

```
@Entity @EntityListeners({MyListener1.class, MyListener2.class})
public class MyEntityWithTwoListeners {
}
```

Listeners that are attached to an entity class are inherited by its subclasses unless the subclass excludes inheritance explicitly using the `@ExcludeSuperclassListeners` annotation:

```
@Entity @ExcludeSuperclassListeners
public class EntityWithNoListener extends EntityWithListener {
}
```

JPA Lifecycle Events

Default Entity Listeners

Default entity listeners are listeners that should be applied by default to all the entity classes. Currently, default listeners can only be specified in a mapping XML file because there is no equivalent annotation:

```
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
 http://java.sun.com/xml/ns/persistence/orm_1_0.xsd" version="1.0">
  <persistence-unit-metadata>
    <persistence-unit-defaults>
      <entity-listeners>
        <entity-listener class="samples.MyDefaultListener1" />
        <entity-listener class="samples.MyDefaultListener2" />
      </entity-listeners>
    </persistence-unit-defaults>
  </persistence-unit-metadata>
</entity-mappings>
```

JPA Lifecycle Events

The mapping file has to be located either in the default location, `META-INF/orm.xml`, or in another location that is specified explicitly in the persistence unit definition (in `persistence.xml`).
Default listeners are applied by default to all the entity classes. The `@ExcludeDefaultListeners` annotation can be used to exclude an entity class and all its descendant classes from using the default listeners:

```
@Entity @ExcludeDefaultListeners
public class NoDefaultListenersForThisEntity {
}

@Entity
public class NoDefaultListenersForThisEntityEither
  extends NoDefaultListenersForThisEntity {
}
```

JPA Lifecycle Events

Callback Invocation Order

If more than one callback method has to be invoked for a lifecycle event (e.g. from multiple listeners) the invocation order is based on the following rules:

- All the external callback methods (which are defined in listeners) are invoked before the internal callback methods (which are defined in entity classes).
- Default listeners are handled first, then listeners of the top level entity class, and then down the hierarchy until listeners of the actual entity class. If there is more than one default listener or more than one listener at the same level in the hierarchy, the invocation order follows the definition order.
- Internal callback methods are invoked starting at the top level entity class and then down the hierarchy until the callback methods in the actual entity class are invoked.

Shared (L2) Entity Cache

Every EntityManager owns a persistence context, which is a collection of all the entity objects that it manages. The persistence context serves as a first level cache. An attempt to retrieve an entity object that is already managed by the EntityManager returns the existing instance from the persistence context, rather than a new instantiated entity object.
The scope of the persistence context is one EntityManager. This section describes a level 2 (L2) cache of entity objects, which is managed by the EntityManagerFactory and shared by all its EntityManager objects. the broader scope of this cache makes it useful in applications that use many short term EntityManager instances.
In addition to the EntityManager's L1 cache and the EntityManagerFactory's L2 cache, which are managed on the client side - JPA may manage also several caches on the server side:

- Cache of database file pages.
- Cache of query programs.
- Cache of query execution results.

The scope of these server side caches is wider, since they exist per database and are shared by all the EntityManagerFactory and EntityManager instances of the same database - including on different client machines.

Shared (L2) Entity Cache

Setting the Shared Cache

The shared (L2) cache is configured in three scopes:

- Per persistence unit in the `persistence.xml` file.
- Per entity class - using annotations.

Shared (L2) Entity Cache

Persistence Unit Settings

The shared cache can also be enabled or disabled using a persistence unit property:

```
<persistence-unit name="my-pu">
  ...
  <properties>
    <property name="javax.persistence.sharedCache.mode" value="ALL"/>
  </properties>
  ...
</persistence-unit>
```

The `javax.persistence.sharedCache.mode` property can be set to one of the following values:

- `NONE` - cache is disabled.
- `ENABLE_SELECTIVE` - cache is disabled except for selected entity classes (see below).
- `DISABLE_SELECTIVE` - cache is enabled except for selected entity classes (see below).
- `ALL` (the default) - cache is enabled for all the entity classes.
- `UNSPECIFIED` - handled differently by different JPA providers.

If the cache size is `0` - the shared cache is disabled regardless of the set mode.

Shared (L2) Entity Cache

Entity Class Cache Settings

The `ENABLE_SELECTIVE` mode indicates that the cache is disabled for all the entity classes except classes that are specified as `Cacheable` explicitly. For example:

```
@Cacheable // or @Cacheable(true)
@Entity
public class MyCacheableEntityClass {
    ...
}
```

Similarly, the `DISABLE_SELECTIVE` value indicates that the cache is enabled for all the entity classes except classes that are specified as non `Cacheable` explicitly. For example:

```
@Cacheable(false)
@Entity
public class MyNonCacheableEntityClass extends MyCacheableEntityClass {
    ...
}
```

`Cacheable` is an inherited property - every entity class which is not marked with `@Cacheable` inherits cacheability setting from its super class.

Shared (L2) Entity Cache

Using the Shared Cache

The shared cache (when enabled) provides the following functionality automatically:

- **On retrieval** - shared cache is used for entity objects that are not in the persistence context. If an entity object is not available also in the shared cache - it is retrieved from the database and added to the shared cache.
- **On commit** - new and modified entity objects are added to the shared cache.

JPA provides two properties that can be used in order to change the default behavior

Shared (L2) Entity Cache

javax.persistence.cache.retrieveMode

The "javax.persistence.cache.retrieveMode" property specifies if the shared cache is used on retrieval. Two values are available for this property as constants of the CacheRetrieveMode enum:

CacheRetrieveMode.USE - cache is used.
CacheRetrieveMode.BYPASS - cache is not used.

The default setting is USE. It can be changed for a specific EntityManager:

```
em.setProperty(
    "javax.persistence.cache.retrieveMode", CacheRetrieveMode.BYPASS);
```

Setting can also be overridden for a specific retrieval operation:

```
// Before executing a query:
query.setHint("javax.persistence.cache.retrieveMode", CacheRetrieveMode.BYPASS);

// For retrieval by type and primary key:
em.find(MyEntity2.class, Long.valueOf(1),
    Collections.<String,Object>singletonMap(
      "javax.persistence.cache.retrieveMode", CacheRetrieveMode.BYPASS));
```

Shared (L2) Entity Cache

javax.persistence.cache.storeMode

The "javax.persistence.cache.storeMode" property specifies if new data should be added to the cache on commit and on retrieval. The property has three valid values, which are defined as constants of the CacheStoreMode enum:

CacheStoreMode.BYPASS - cache is not updated with new data.
CacheStoreMode.USE - new data is stored in the cache - but only for entity objects that are not in the cache already.
CacheStoreMode.REFRESH - new data is stored in the cache - refreshing entity objects that are already cached.

The default setting is USE. It can be changed for a specific EntityManager:

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

Setting can also be overridden for a specific retrieval operation. For example:

```
em.find(MyEntity2.class, Long.valueOf(1),
    Collections.<String,Object>singletonMap(
        "javax.persistence.cache.storeMode", CacheRetrieveMode.BYPASS));
```

The difference between CacheStoreMode.USE and CacheStoreMode.REFRESH is when bypassing the cache in retrieval operations. In this case, an entity object that is already cached is updated using the fresh retrieved data only when CacheStoreMode.REFRESH is used. This might be useful when the database might be updated by other applications (or using other EntityManagerFactory instances).

Shared (L2) Entity Cache

Using the Cache Interface

The shared cache is represented by the Cache interface. A Cache instance can be obtained by using the EntityManagerFactory's getCache method:

```
Cache cache = emf.getCache();
```

The Cache object enables checking if a specified entity object is cached:

```
boolean isCached = cache.contains(MyEntity.class, Long.valueOf(id));
```

Cached entity objects can be removed from the cache by one of the evict methods:

```
// Remove a specific entity object from the shared cache:
cache.evict(MyEntity.class, Long.valueOf(id));
```

```
// Remove all the instances of a specific class from the cache:
cache.evict(MyEntity.class);
```

```
// Clear the shared cache by removing all the cached entity objects:
cache.evictAll();
```

The Cache interface and its methods are unnecessary in most applications.

JPA Metamodel API

The JPA Metamodel API provides the ability to examine the persistent object model and retrieve details on managed classes and persistent fields and properties, similarly to the ability that Java reflection provides for general Java types.

The Metamodel Interface

The main interface of the JPA Metamodel API is `Metamodel`. It can be obtained either by the `EntityManagerFactory`'s `getMetamodel` method or by the `EntityManager`'s `getMetamodel` method (both methods are equivalent).
For example, given an `EntityManager`, `em`, a `Metamodel` instance can be obtained by:

```
Metamodel metamodel = em.getMetamodel();
```

The `Metamodel` interface provides several methods for exploring user defined persistable types (which are referred to as managed types) in the persistent object model.
Three methods can be used to retrieve sets of types:

```
// Get all the managed classes:
// (entity classes, embeddable classes, mapped super classes)
Set<ManagedType> allManagedTypes = metamodel.getManagedTypes();
```

```
// Get all the entity classes:
Set<EntityType> allEntityTypes = metamodel.getEntities();
```

```
// Get all the embeddable classes:
Set<EmbeddableType> allEmbeddableTypes = metamodel.getEmbeddables();
```

JPA Metamodel API

If managed classes are not listed in the persistence unit then only known managed types are returned. This includes all the types whose instances are already stored in the database. Three additional methods can be used to retrieve a specific type by its `Class` instance:

```
// Get a managed type (entity, embeddable or mapped super classes):
ManagedType<MyClass> type1 = metamodel.managedType(MyClass.class);

// Get an entity type:
EntityType<MyEntity> type2 = metamodel.entity(MyEntity.class);

// Get an embeddable type:
EmbeddableType<MyEmbeddableType> type3 =
    metamodel.embeddable(MyEmbeddableType.class);
```

These three methods can also be used with types that are still unknown to JPA (not listed in the persistence unit and have not been used yet). In this case, calling the method introduces the specified type to JPA.

JPA Metamodel API

Type Interface Hierarchy

Types are represented in the Metamodel API by descendant interfaces of the `Type` interface:

- `BasicType` - represents system defined types.
- `ManagedType` is an ancestor of interfaces that represent user defined types:
- `EmbeddableType` - represents user defined embeddable classes.
- `IdentifiableType` is as a super interface of:
- `MappedSuperclassType` - represents user defined mapped super classes.
- `EntityType` - represents user defined entity classes.

The `Type` interfaces provides a thin wrapper of `Class` with only two methods:

```
// Get the underlying Java representation of the type:
Class cls = type.getJavaType();
```

```
// Get one of BASIC, EMBEDDABLE, ENTITY, MAPPED_SUPERCLASS:
PersistenceType kind = type.getPersistenceType();
```

JPA Metamodel API

The ManagedType interface adds methods for exploring managed fields and properties (which are referred to as attributes). For example:

```
// Get all the attributes - including inherited:
Set<Attribute> attributes1 = managedType.getAttributes();

// Get all the attributes - excluding inherited:
Set<Attribute> attributes2 = managedType.getDeclaredAttributes();

// Get a specific attribute - including inherited:
Attribute<MyClass,String> strAttr1 = managedType.getAttribute("name");

// Get a specific attribute - excluding inherited:
Attribute<MyClass,String> strAttr2 = managedType.getDeclaredAttribute("name");
```

Additional methods are defined in ManagedType to return attributes of `Collection`, `List`, `Set` a Map types in a type safe manner.

JPA Metamodel API

The `IdentifiableType` adds methods for retrieving information on the primary key and the version attributes and the super type. For example:

```
// Get the super type:
IdentifiableType<MyEntity> superType = entityType.getSupertype();
```

```
// Checks if the type has a single ID attribute:
boolean hasSingleId = entityType.hasSingleIdAttribute();
```

```
// Gets a single ID attribute - including inherited:
SingularAttribute<MyEntity,Long> id1 = entityType.getId(Long.class);
```

```
// Gets a single ID attribute - excluding inherited:
SingularAttribute<MyEntity,Long> id2 = entityType.getDeclaredId(Long.class);
```

JPA Metamodel API

```
// Checks if the type has a version attribute:
boolean hasVersion = entityType.hasVersionAttribute();

// Gets the version attribute - excluding inherited:
SingularAttribute<MyEntity,Long> v1 = entityType.getVersion(Long.class);

// Gets the version attribute - including inherited:
SingularAttribute<MyEntity,Long> v2 = entityType.getDeclaredVersion(Long.class);
```

Additional methods are defined in `IdentifiableType` to support an ID class when using multiple ID fields or properties.
Finally, the `EntityType` interface adds only one additional method for getting the entity name:

```
String entityName = entityType.getName();
```

JPA Metamodel API

Attribute Interface Hierarchy

Managed fields and properties are represented by the `Attribute` interfaces and its descendant interfaces:

- `SingularAttribute` - represents single value attributes.
- `PluralAttribute` is an ancestor of interfaces that represent multi value attributes:
- `CollectionAttribute` - represents attributes of `Collection` types.
- `SetAttribute` - represents attributes of `Set` types.
- `ListAttribute` represents attributes of `List` types.
- `MapAttribute` - represents attributes of `Map` types.

The `Attribute` interface provides methods for retrieving field and property details. For example:

```
// Get the field (or property) name:
String name = attr.getName();
```

```
// Get Java representation of the field (or property) type:
Class<Integer> attr.getJavaType();
```

```
// Get Java reflection representation of the field (or property) type:
Member member = attr.getJavaMember();
```

```
// Get the type in which this field (or property) is defined:
ManagedType<MyEntity> entityType = attr.getDeclaringType();
```

Few other methods are defined in `Attribute` and in `MapAttribute` to support additional details.