Java Enterprise Edition

ANUDIP FOUNDATION

JPA with Hibernate

Objective: • Understanding ORM tools • Entities • Entity Relationships	**Materials Required:** 1. Eclipse IDE/IntelliJ 2. ORM tool 3. SQL
Theory:240mins	**Practical:120mins**
Total Duration: 360 mins	

JPA with Hibernate 3.0

Contents:

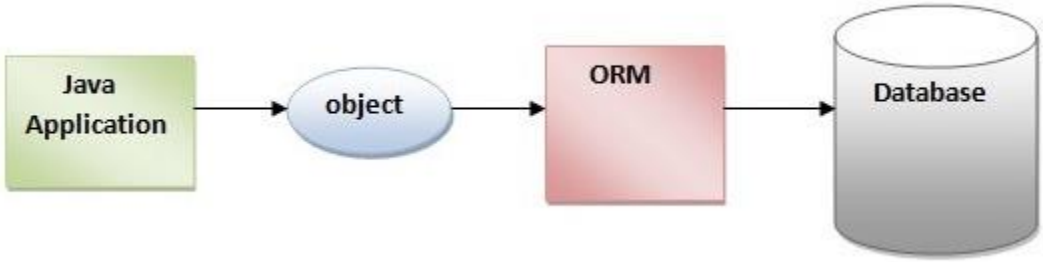
- **Introduction**
  - Introduction & overview of data persistence
  - Overview of ORM tools
  - Understanding JPA
  - JPA Specifications
- **Entities**
  - Requirements for Entity Classes
  - Persistent Fields and Properties in Entity Classes
  - Persistent Fields
  - Persistent Properties
  - Using Collections in Entity Fields and Properties
  - Validating Persistent Fields and Properties
  - Primary Keys in Entities
- **Managing Entities**
  - The EntityManager Interface
  - Container-Managed Entity Managers
  - Application-Managed Entity Managers
  - Finding Entities Using the EntityManager
  - Managing an Entity Instance's Lifecycle
  - Persisting Entity Instances
  - Removing Entity Instances
  - Synchronizing Entity Data to the Database
  - Persistence Units
- **Querying Entities**
  - Java Persistence query language (JPQL)
  - Criteria API
- **Entity Relationships**
  - Direction in Entity Relationships
  - Bidirectional Relationships
  - Unidirectional Relationships
  - Queries and Relationship Direction
  - Cascade Operations and Relationship

Hibernate Framework

Hibernate is a Java framework that simplifies the development of Java application to interact with the database. It is an open source, lightweight, ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.

ORM Tool

An ORM tool simplifies the data creation, data manipulation and data access. It is a programming technique that maps the object to the data stored in the database.



The ORM tool internally uses the JDBC API to interact with the database.

What is JPA?

Java Persistence API (JPA) is a Java specification that provides certain functionality and standard to ORM tools. The **javax.persistence** package contains the JPA classes and interfaces.

Advantages of Hibernate Framework

Following are the advantages of hibernate framework:

1) Open Source and Lightweight

Hibernate framework is open source under the LGPL license and lightweight.

2) Fast Performance

The performance of hibernate framework is fast because cache is internally used in hibernate framework. There are two types of cache in hibernate framework first level cache and second level cache. First level cache is enabled by default.

3) Database Independent Query

HQL (Hibernate Query Language) is the object-oriented version of SQL. It generates the database independent queries. So you don't need to write database specific queries. Before Hibernate, if database is changed for the project, we need to change the SQL query as well that leads to the maintenance problem.

4) Automatic Table Creation

Hibernate framework provides the facility to create the tables of the database automatically. So there is no need to create tables in the database manually.

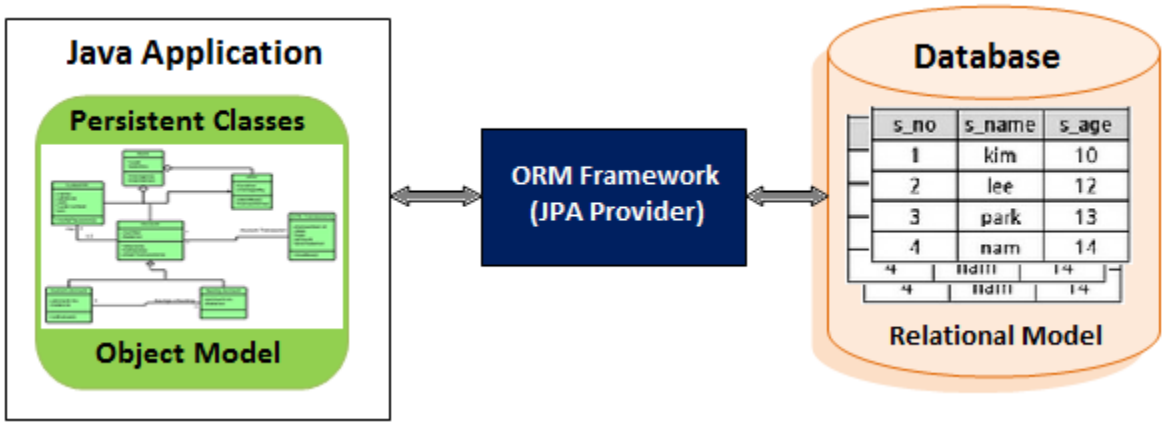
5) Simplifies Complex Join

Fetching data from multiple tables is easy in hibernate framework.

6) Provides Query Statistics and Database Status

Hibernate supports Query cache and provide statistics about query and database status.

Understanding JPA



Components of JPA

In order to develop JPA based Java applications, you need to understand its major components such as Entity, EntityManager, Persistence Unit and EntityTransaction. The relationship among them is depicted the Figure-2.

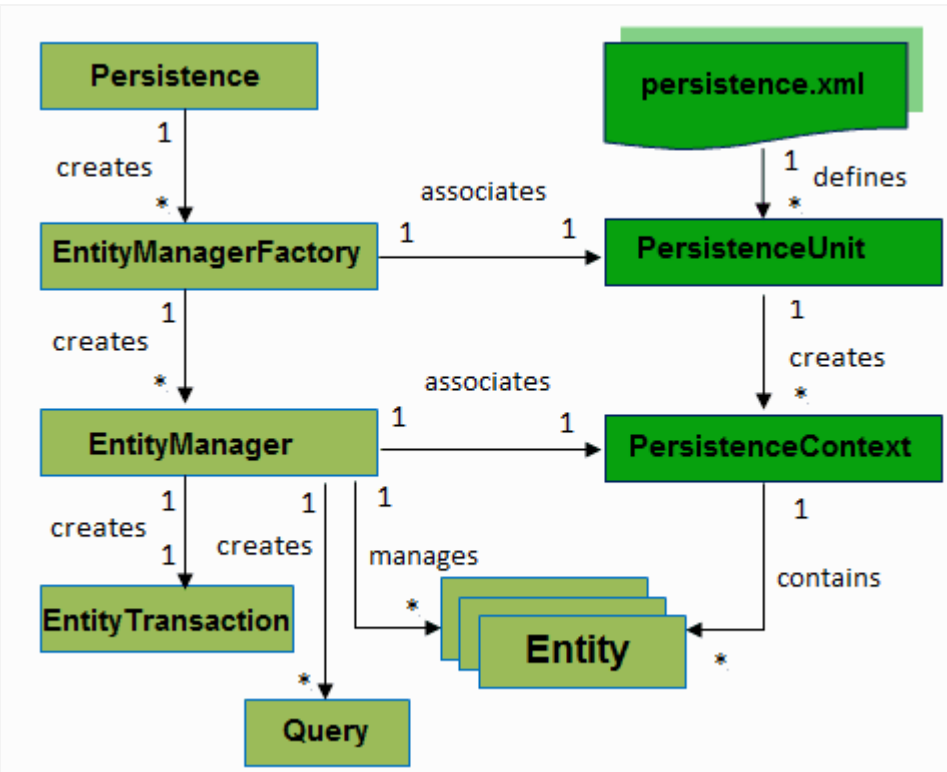


Figure-2: Components of JPA

Entity:

Entities form the heart of any JPA based application. An entity is a POJO class that models a persistence domain object in the application. It represents a table in a relational database. The persistence properties of an entity class represent the columns in the table. An instance of the entity class corresponds to a row in the table. Annotations are used for this object-relational mapping.

An entity has three main characteristics such as persistence, identity and transactional. The persistence characteristic deals with storing and retrieving of entity instances to and from a database. The identity characteristic is usually used to identify an entity in a database uniquely. All the CRUD operations (Create, Read, Update and Delete) for entity instances will occur within a transactional context as the real state of an entity depends whether a transaction completes or not.

Any Java class implementing serializable interface with a zero-argument constructor, getter and setter methods for its properties can be made an entity. The class is marked with @Entity annotation. By default the table name will be name of the class. We can change the default table name by setting a value to the name attribute of the annotation. A property of the class that corresponds to a table column can take @Column annotation. The name of a column, its size, whether it can accept null value or not etc., can be customized using the @Column annotation. A property that is marked with @Id annotation will be treated as a primary key for the table and is used to identify an entity instance

uniquely. By default, all the properties of a class are persistent except those marked with the @Transient annotation. All these annotations are available in the javax.persistence package.

The Java class in Listing-1 forms an entity for an employee table with the following schema:

```
EMPLOYEE (EMPID INTEGER PK NOT NULL,
        EMPNAME VARCHAR (50) NOT NULL,
        DESIGNATION VARCHAR (20))
```

Listing-1: The Employee entity

```
import javax.persistence.*;
import java.io.Serializable;

@Entity (name = "EMPLOYEE") // Name of the entity
public class Employee { // This column is the primary key

@Id // default column name is the same as the property name
@Column(name = "EMPID", nullable = false)
private long empId;

@Column(name = "EMPNAME", nullable = false, length = 50)
private String empName; @Column(name = "DESIGNATION", nullable = true, length = 20)

private String designation;

public Employee() { }

- - -
// getter and setter methods go here

 }
```

EntityManager

Entities in an application are managed by EntityManager instances. Each EntityManager instance is associated with a persistence context. A persistence context is an active collection of entity instances that are being used by the application and exist in a particular data store. In other words, a persistence context contains in-memory entity instances representing entity instances in the data store. Within the persistence context, the entity instance and their life cycle are managed.

The EntityManager has methods to create and remove persistent entity instances, find entities by their primary key, and query over entities. An EntityManager can be one of two types: container-managed or application-managed.

Container-Managed EntityManager: A container-managed EntityManager is that whose life-cycle is managed by the Java EE container. Only one instance of the EntityManager is available to the application components directly through dependency injection. The associated persistence context is automatically propagated to all application components that use the EntityManager instance.

The instance of the EntityManager is obtained using annotation:

```
@PersistenceContext
EntityManager em;
```

Application-Managed EntityManager: If the life-cycle of the EntityManager is managed by the application, several EntityManager instances are used in the application. The persistence context is not automatically propagated to application components. In this case, each EntityManager creates a new, isolated persistence context. The application explicitly creates and destroys EntityManager instances and associated persistence contexts. The createEntityManager() method of an EntityManagerFactory instance is used to create an EntityManager instance. An EntityManagerFactory instance is associated with one persistence unit. A persistence unit defines a set of entities existing in a data store. The EntityManagerFactory instance can be obtained either through dependency injection or manually using lookup method.

The following code is used to obtain EntityManagerFactory using dependency injection in Java EE environment.

```
@PersistenceUnit
EntityManagerFactory emf;
EntityManager em = emf.createEntityManager();
```

The cretateEntityManagerFactory()method of the Persistence bootstrap class is used to obtain an EntityManagerFactory in Java SE environments. It is available in a Java EE container environment as well.

```
EntityManagerFactory  emf =
```

```
Persistence.createEntityManagerFactory("PersistenceUnitName");

EntityManager em =
emf.createEntityManager();
```

PersistenceUnit

A PersistenceUnit defines a logical grouping of all entity classes managed by EntityManager instances in an application. These entity classes represent the tables in a database. One or more persistence units can be defined in a persistence.xml configuration file. The following is an example persistence.xml file:

```
<persistence>
<persistence-unit name="EmployeePU" transaction-type=" RESOURCE_LOCAL">
<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
<non-jta-data-source>empData&lt/non-jta-data-source>
<exclude-unlisted-classes>false</exclude-unlisted-classes>
<properties>
<property name="eclipselink.ddl-generation" value="create-tables">
</property>
</properties>
</persistence-unit>
</persistence>
```

A persistence unit is defined by a XML element. The required name attribute ("EmployeePU" in the example) identifies the persistence unit when creating an instance of EntityManagerFactory. The transaction-type attribute indicates whether JTA (Java Transaction API) transaction or non-JTA transaction will be used on entity operations. The provider element indicates which JPA implementation should be used. It may also have many optional elements.

EntityTransaction

Entity operations that modify the database must be performed within an active transaction context. The transaction context can be based on either JTA transaction or non-JTA transaction. JTA transaction is usually used for container-managed EntityManager while non-JTA transaction can be used for an application-managed EntityManager which is resource-local. An EntityTransaction instance represents the transaction context and manages database transactions for a resource-local EntityManager. It has methods to demarcate a transaction boundary during entity operations. Modifications to the entity are confined to the memory until a transaction is ended. The begin() method is used to begin a transaction and either commit() or rollback() is used to end the transaction. When all the entity operations are successfully ended, commit() method is called to synchronize data with the underlying database. If the

transaction is ended unsuccessful, rollback() method is called to rollback the current transaction discarding the changes made to the database if any. 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. An EntityManager instance invokes its getTransaction() method to create one instance of EntityTransaction.

The following code illustrates transaction management described above:

```
EntityTransaction etx = em.getTransaction();
try {
    etx.begin();
 // Entity operations modifying the database go here.
    etx.commit();
}catch (Exception e) {
    etx.rollback();
  }
```

Query

The EntityManager uses a query language, JPQL (Java Persistence Query language) to query entities. An instance of the EntityManager may create several instances of Query in order to query on entities. An instance of Query represents a JPQL query or a native SQL query and is used to control execution of the query. The Query instance supports applying lock on the query and allows paging on query results.

1.Specifications of JPA

build a **Search/Filter REST API** using Spring Data JPA and Specifications.

We started looking at a query language in the first article of this series – with a JPA Criteria based solution.

So – **why a query language?** Because – for any complex-enough API – searching/filtering your resources by very simple fields is simply not enough. **A query language is more flexible** and allows you to filter down to exactly the resources you need.

2. ***User*** Entity

First – let's start with a simple *User* entity for our Search API:

```
@Entity
```

```
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String firstName;
    private String lastName;
    private String email;

    private int age;

    // standard getters and setters
}
```

3. Filter Using ***Specification***

Now – let's get straight into the most interesting part of the problem – querying with custom Spring Data JPA *Specifications*.

We'll create a *UserSpecification* which implements the *Specification* interface and we're going to **pass in our own constraint to construct the actual query**:

```
public class UserSpecification implements Specification<User> {

    private SearchCriteria criteria;

    @Override
    public Predicate toPredicate
      (Root<User> root, CriteriaQuery<?> query, CriteriaBuilder builder) {

        if (criteria.getOperation().equalsIgnoreCase(">")) {
            return builder.greaterThanOrEqualTo(
              root.<String> get(criteria.getKey()), criteria.getValue().toString());
        }
        else if (criteria.getOperation().equalsIgnoreCase("<")) {
            return builder.lessThanOrEqualTo(
              root.<String> get(criteria.getKey()), criteria.getValue().toString());
        }
        else if (criteria.getOperation().equalsIgnoreCase(":")) {
            if (root.get(criteria.getKey()).getJavaType() == String.class) {
                return builder.like(
                  root.<String>get(criteria.getKey()), "%" + criteria.getValue() + "%");
            } else {
                return builder.equal(root.get(criteria.getKey()), criteria.getValue());
            }
        }
        return null;
    }
}
```

As we can see – **we create a *Specification* based on some simple constrains** which we represent in the following "*SearchCriteria*" class:

```
public class SearchCriteria {
   private String key;
   private String operation;
   private Object value;
}
```

The *SearchCriteria* implementation holds a basic representation of a constraint – and it's based on this constraint that we're going to be constructing the query:

- *key*: the field name – for example, *firstName*, *age*, … etc.
- *operation*: the operation – for example, equality, less than, … etc.
- *value*: the field value – for example, john, 25, … etc.

Of course, the implementation is simplistic and can be improved; it is however a solid base for the powerful and flexible operations we need.

4. The ***UserRepository***

Next – let's take a look at the *UserRepository*; we're simply extending the *JpaSpecificationExecutor* to get the new Specification APIs:

```
public interface UserRepository
  extends JpaRepository<User, Long>, JpaSpecificationExecutor<User> {}
```

5. Test the Search Queries

Now – let's test out the new search API.

First, let's create a few users to have them ready when the tests run:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = { PersistenceJPAConfig.class })
@Transactional
@TransactionConfiguration
public class JPASpecificationsTest {

   @Autowired
   private UserRepository repository;

   private User userJohn;
   private User userTom;

   @Before
   public void init() {
      userJohn = new User();
      userJohn.setFirstName("John");
      userJohn.setLastName("Doe");
      userJohn.setEmail("john@doe.com");
      userJohn.setAge(22);
```

```
        repository.save(userJohn);

        userTom = new User();
        userTom.setFirstName("Tom");
        userTom.setLastName("Doe");
        userTom.setEmail("tom@doe.com");
        userTom.setAge(26);
        repository.save(userTom);
    }
}
```

Next, let's see how to find users with **given last name**:

```
@Test
public void givenLast_whenGettingListOfUsers_thenCorrect() {
    UserSpecification spec =
      new UserSpecification(new SearchCriteria("lastName", ":", "doe"));

    List<User> results = repository.findAll(spec);

    assertThat(userJohn, isIn(results));
    assertThat(userTom, isIn(results));
}
```

Now, let's see how to find a user with given **both first and last name**:

```
@Test
public void givenFirstAndLastName_whenGettingListOfUsers_thenCorrect() {
    UserSpecification spec1 =
      new UserSpecification(new SearchCriteria("firstName", ":", "john"));
    UserSpecification spec2 =
      new UserSpecification(new SearchCriteria("lastName", ":", "doe"));

    List<User> results = repository.findAll(Specification.where(spec1).and(spec2));

    assertThat(userJohn, isIn(results));
    assertThat(userTom, not(isIn(results)));
}
```

Note: We used "*where*" and "*and*" to **combine Specifications**.

Next, let's see how to find a user with given **both last name and minimum age**:

```
@Test
public void givenLastAndAge_whenGettingListOfUsers_thenCorrect() {
    UserSpecification spec1 =
      new UserSpecification(new SearchCriteria("age", ">", "25"));
    UserSpecification spec2 =
      new UserSpecification(new SearchCriteria("lastName", ":", "doe"));

    List<User> results =
      repository.findAll(Specification.where(spec1).and(spec2));

    assertThat(userTom, isIn(results));
```

```
    assertThat(userJohn, not(isIn(results)));
}
```

Now, let's see how to search for *User* that **doesn't actually exist**:

```
@Test
public void givenWrongFirstAndLast_whenGettingListOfUsers_thenCorrect() {
    UserSpecification spec1 =
      new UserSpecification(new SearchCriteria("firstName", ":", "Adam"));
    UserSpecification spec2 =
      new UserSpecification(new SearchCriteria("lastName", ":", "Fox"));

    List<User> results =
      repository.findAll(Specification.where(spec1).and(spec2));

    assertThat(userJohn, not(isIn(results)));
    assertThat(userTom, not(isIn(results)));
}
```

Finally – let's see how to find a *User* **given only part of the first name**:

```
@Test
public void givenPartialFirst_whenGettingListOfUsers_thenCorrect() {
    UserSpecification spec =
      new UserSpecification(new SearchCriteria("firstName", ":", "jo"));

    List<User> results = repository.findAll(spec);

    assertThat(userJohn, isIn(results));
    assertThat(userTom, not(isIn(results)));
}
```

6. Combine ***Specifications***

Next – let's take a look at combining our custom *Specifications* to use multiple constraints and filter according to multiple criteria.

We're going to implement a builder – *UserSpecificationsBuilder* – to easily and fluently combine *Specifications*:

```
public class UserSpecificationsBuilder {

    private final List<SearchCriteria> params;

    public UserSpecificationsBuilder() {
        params = new ArrayList<SearchCriteria>();
    }

    public UserSpecificationsBuilder with(String key, String operation, Object value) {
        params.add(new SearchCriteria(key, operation, value));
        return this;
    }
```

```
    public Specification<User> build() {
        if (params.size() == 0) {
            return null;
        }

        List<Specification> specs = params.stream()
          .map(UserSpecification::new)
          .collect(Collectors.toList());

        Specification result = specs.get(0);

        for (int i = 1; i < params.size(); i++) {
            result = params.get(i)
              .isOrPredicate()
                ? Specification.where(result)
                  .or(specs.get(i))
                : Specification.where(result)
                  .and(specs.get(i));
        }
        return result;
    }
}
```

7. ***UserController***

Finally – let's use this new persistence search/filter functionality and **set up the REST API** – by creating a *UserController* with a simple *search* operation:

```
@Controller
public class UserController {

    @Autowired
    private UserRepository repo;

    @RequestMapping(method = RequestMethod.GET, value = "/users")
    @ResponseBody
    public List<User> search(@RequestParam(value = "search") String search) {
        UserSpecificationsBuilder builder = new UserSpecificationsBuilder();
        Pattern pattern = Pattern.compile("(\\w+?)(:|<|>)(\\w+?),");
        Matcher matcher = pattern.matcher(search + ",");
        while (matcher.find()) {
            builder.with(matcher.group(1), matcher.group(2), matcher.group(3));
        }

        Specification<User> spec = builder.build();
        return repo.findAll(spec);
    }
}
```

Note that to support other non-English systems, the *Pattern* object could be changed like:

```
Pattern pattern = Pattern.compile("(\\w+?)(:|<|>)(\\w+?),", Pattern.UNICODE_CHARACTER_CLASS);
```

Here is a test URL example to test out the API:

```
http://localhost:8080/users?search=lastName:doe,age>25
```

And the response:

```
[{
   "id":2,
   "firstName":"tom",
   "lastName":"doe",
   "email":"tom@doe.com",
   "age":26
}]
```

Since the searches are split by a ",” in our *Pattern* example, the search terms can't contain this character. The pattern also doesn't match whitespace.

If we want to search for values containing commas, then we can consider using a different separator such as ";”.

Another option would be to change the pattern to search for values between quotes, then strip these from the search term:

```
Pattern pattern = Pattern.compile("(\\w+?)(:|<|>)(\"([^\"]+)\")");
```

Entities

Requirements for Entity Classes

An entity class must follow these requirements.

- The class must be annotated with the javax.persistence.Entity annotation.
- The class must have a public or protected, no-argument constructor. The class may have other constructors.
- The class must not be declared final. No methods or persistent instance variables must be declared final.
- If an entity instance is passed by value as a detached object, such as through a session bean's remote business interface, the class must implement the Serializable interface.
- Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
- Persistent instance variables must be declared private, protected, or package-private and can be accessed directly only by the entity class's methods. Clients must access the entity's state through accessor or business methods.

Persistent Fields and Properties in Entity Classes

The persistent state of an entity can be accessed through either the entity's instance variables or properties. The fields or properties must be of the following Java language types:

- Java primitive types
- java.lang.String
- Other serializable types, including:

  - Wrappers of Java primitive types
  - java.math.BigInteger
  - java.math.BigDecimal
  - java.util.Date
  - java.util.Calendar
  - java.sql.Date
  - java.sql.Time
  - java.sql.TimeStamp
  - User-defined serializable types
  - byte[]
  - Byte[]
  - char[]
  - Character[]
- Enumerated types
- Other entities and/or collections of entities
- Embeddable classes

Entities may use persistent fields, persistent properties, or a combination of both. If the mapping annotations are applied to the entity's instance variables, the entity uses persistent fields. If the mapping annotations are applied to the entity's getter methods for JavaBeans-style properties, the entity uses persistent properties.

Persistent Fields

If the entity class uses persistent fields, the Persistence runtime accesses entity-class instance variables directly. All fields not annotated javax.persistence.Transient or not marked as Java transient will be persisted to the data store. The object/relational mapping annotations must be applied to the instance variables.

Persistent Properties

If the entity uses persistent properties, the entity must follow the method conventions of JavaBeans components. JavaBeans-style properties use getter and setter methods that are typically named after the entity class's instance variable names. For every persistent property *property* of type *Type* of the entity, there is a getter method get*Property* and setter method set*Property*. If the property is a Boolean, you may use is*Property* instead of get*Property*. For example, if a Customer entity uses persistent properties and has a private instance variable called firstName, the class defines a getFirstName and setFirstName method for retrieving and setting the state of the firstName instance variable.

The method signature for single-valued persistent properties are as follows:

Type getProperty()
void setProperty(Type type)

The object/relational mapping annotations for persistent properties must be applied to the getter methods. Mapping annotations cannot be applied to fields or properties annotated @Transient or marked transient.

Using Collections in Entity Fields and Properties

Collection-valued persistent fields and properties must use the supported Java collection interfaces regardless of whether the entity uses persistent fields or properties. The following collection interfaces may be used:

- java.util.Collection
- java.util.Set
- java.util.List
- java.util.Map

If the entity class uses persistent fields, the type in the preceding method signatures must be one of these collection types. Generic variants of these collection types may also be used. For example, if it has a persistent property that contains a set of phone numbers, the Customer entity would have the following methods:

```
Set<PhoneNumber> getPhoneNumbers() { ... }
void setPhoneNumbers(Set<PhoneNumber>) { ... }
```

If a field or property of an entity consists of a collection of basic types or embeddable classes, use the javax.persistence.ElementCollection annotation on the field or property.

The two attributes of @ElementCollection are targetClass and fetch. The targetClass attribute specifies the class name of the basic or embeddable class and is optional if the field or property is defined using Java programming language generics. The optional fetch attribute is used to specify whether the collection should be retrieved lazily or eagerly, using the javax.persistence.FetchType constants of either LAZY or EAGER, respectively. By default, the collection will be fetched lazily.

The following entity, Person, has a persistent field, nicknames, which is a collection of String classes that will be fetched eagerly. The targetClass element is not required, because it uses generics to define the field.

```
@Entity
public class Person {
   ...
   @ElementCollection(fetch=EAGER)
   protected Set<String> nickname = new HashSet();
   ...
}
```

Collections of entity elements and relationships may be represented by java.util.Map collections. A Map consists of a key and a value.

When using Map elements or relationships, the following rules apply.

- The Map key or value may be a basic Java programming language type, an embeddable class, or an entity.
- When the Map value is an embeddable class or basic type, use the @ElementCollection annotation.
- When the Map value is an entity, use the @OneToMany or @ManyToMany annotation.

- Use the Map type on only one side of a bidirectional relationship.

If the key type of a Map is a Java programming language basic type, use the annotation javax.persistence.MapKeyColumn to set the column mapping for the key. By default, the name attribute of @MapKeyColumn is of the form *RELATIONSHIP-FIELD/PROPERTY-NAME*_KEY. For example, if the referencing relationship field name is image, the default name attribute is IMAGE_KEY.

If the key type of a Map is an entity, use the javax.persistence.MapKeyJoinColumn annotation. If the multiple columns are needed to set the mapping, use the annotation javax.persistence.MapKeyJoinColumns to include multiple @MapKeyJoinColumn annotations. If no @MapKeyJoinColumn is present, the mapping column name is by default set to *RELATIONSHIP-FIELD/PROPERTY-NAME*_KEY. For example, if the relationship field name is employee, the default name attribute is EMPLOYEE_KEY.

If Java programming language generic types are not used in the relationship field or property, the key class must be explicitly set using the javax.persistence.MapKeyClass annotation.

If the Map key is the primary key or a persistent field or property of the entity that is the Map value, use the javax.persistence.MapKey annotation. The @MapKeyClass and @MapKey annotations cannot be used on the same field or property.

If the Map value is a Java programming language basic type or an embeddable class, it will be mapped as a collection table in the underlying database. If generic types are not used, the @ElementCollection annotation's targetClass attribute must be set to the type of the Map value.

If the Map value is an entity and part of a many-to-many or one-to-many unidirectional relationship, it will be mapped as a join table in the underlying database. A unidirectional one-to-many relationship that uses a Map may also be mapped using the @JoinColumn annotation.

If the entity is part of a one-to-many/many-to-one bidirectional relationship, it will be mapped in the table of the entity that represents the value of the Map. If generic types are not used, the targetEntity attribute of the @OneToMany and @ManyToMany annotations must be set to the type of the Map value.

Validating Persistent Fields and Properties

The Java API for JavaBeans Validation (Bean Validation) provides a mechanism for validating application data. Bean Validation is integrated into the Java EE containers, allowing the same validation logic to be used in any of the tiers of an enterprise application.

Bean Validation constraints may be applied to persistent entity classes, embeddable classes, and mapped superclasses. By default, the Persistence provider will automatically perform validation on entities with persistent fields or properties annotated with Bean Validation constraints immediately after the PrePersist, PreUpdate, and PreRemove lifecycle events.

Bean Validation constraints are annotations applied to the fields or properties of Java programming language classes. Bean Validation provides a set of constraints as well as an API for defining custom constraints. Custom constraints can be specific combinations of the default constraints, or new constraints that don't use the default constraints. Each constraint is associated with at least one validator class that

validates the value of the constrained field or property. Custom constraint developers must also provide a validator class for the constraint.

Bean Validation constraints are applied to the persistent fields or properties of persistent classes. When adding Bean Validation constraints, use the same access strategy as the persistent class. That is, if the persistent class uses field access, apply the Bean Validation constraint annotations on the class's fields. If the class uses property access, apply the constraints on the getter methods.

Table 9–2 lists Bean Validation's built-in constraints, defined in the javax.validation.constraints package.

All the built-in constraints listed in Table 9–2 have a corresponding annotation, *ConstraintName*.List, for grouping multiple constraints of the same type on the same field or property. For example, the following persistent field has two @Pattern constraints:

```
@Pattern.List({
   @Pattern(regexp="..."),
   @Pattern(regexp="...")
})
```

The following entity class, Contact, has Bean Validation constraints applied to its persistent fields.

```
@Entity
public class Contact implements Serializable {
   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   @NotNull
   protected String firstName;
   @NotNull
   protected String lastName;
   @Pattern(regexp="[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\."
      +"[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@"
      +"(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?",
         message="{invalid.email}")
   protected String email;
   @Pattern(regexp="^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
         message="{invalid.phonenumber}")
   protected String mobilePhone;
   @Pattern(regexp="^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
         message="{invalid.phonenumber}")
   protected String homePhone;
   @Temporal(javax.persistence.TemporalType.DATE)
   @Past
   protected Date birthday;
   ...
}
```

The @NotNull annotation on the firstName and lastName fields specifies that those fields are now required. If a new Contact instance is created where firstName or lastName have not been initialized,

Bean Validation will throw a validation error. Similarly, if a previously created instance of Contact has been modified so that firstName or lastName are null, a validation error will be thrown.

The email field has a @Pattern constraint applied to it, with a complicated regular expression that matches most valid email addresses. If the value of email doesn't match this regular expression, a validation error will be thrown.

The homePhone and mobilePhone fields have the same @Pattern constraints. The regular expression matches 10 digit telephone numbers in the United States and Canada of the form (*xxx*) *xxx–xxxx*.

The birthday field is annotated with the @Past constraint, which ensures that the value of birthday must be in the past.

Managing Entities

Entities are managed by the entity manager, which is represented by javax.persistence.EntityManager instances. Each EntityManager instance is associated with a persistence context: a set of managed entity instances that exist in a particular data store. A persistence context defines the scope under which particular entity instances are created, persisted, and removed. The EntityManager interface defines the methods that are used to interact with the persistence context.

The EntityManager Interface

The EntityManager API creates and removes persistent entity instances, finds entities by the entity's primary key, and allows queries to be run on entities.

Container-Managed Entity Managers

With a **container-managed entity manager**, an EntityManager instance's persistence context is automatically propagated by the container to all application components that use the EntityManager instance within a single Java Transaction API (JTA) transaction.

JTA transactions usually involve calls across application components. To complete a JTA transaction, these components usually need access to a single persistence context. This occurs when an EntityManager is injected into the application components by means of the javax.persistence.PersistenceContext annotation. The persistence context is automatically propagated with the current JTA transaction, and EntityManager references that are mapped to the same persistence unit provide access to the persistence context within that transaction. By automatically propagating the persistence context, application components don't need to pass references to EntityManager instances to each other in order to make changes within a single transaction. The Java EE container manages the lifecycle of container-managed entity managers.

To obtain an EntityManager instance, inject the entity manager into the application component:

```
@PersistenceContext
EntityManager em;
```

Application-Managed Entity Managers

With an **application-managed entity manager**, on the other hand, the persistence context is not propagated to application components, and the lifecycle of EntityManager instances is managed by the application.

Application-managed entity managers are used when applications need to access a persistence context that is not propagated with the JTA transaction across EntityManager instances in a particular persistence unit. In this case, each EntityManager creates a new, isolated persistence context. The EntityManager and its associated persistence context are created and destroyed explicitly by the application. They are also used when directly injecting EntityManager instances can't be done because EntityManager instances are not thread-safe. EntityManagerFactory instances are thread-safe.

Applications create EntityManager instances in this case by using the createEntityManager method of javax.persistence.EntityManagerFactory.

To obtain an EntityManager instance, you first must obtain an EntityManagerFactory instance by injecting it into the application component by means of the javax.persistence.PersistenceUnit annotation:

```
@PersistenceUnit
EntityManagerFactory emf;
```

Then obtain an EntityManager from the EntityManagerFactory instance:

```
EntityManager em = emf.createEntityManager();
```

Application-managed entity managers don't automatically propagate the JTA transaction context. Such applications need to manually gain access to the JTA transaction manager and add transaction demarcation information when performing entity operations. The javax.transaction.UserTransaction interface defines methods to begin, commit, and roll back transactions. Inject an instance of UserTransaction by creating an instance variable annotated with @Resource:

```
@Resource
UserTransaction utx;
```

To begin a transaction, call the UserTransaction.begin method. When all the entity operations are complete, call the UserTransaction.commit method to commit the transaction. The UserTransaction.rollback method is used to roll back the current transaction.

The following example shows how to manage transactions in an application that uses an application-managed entity manager:

```
@PersistenceContext
EntityManagerFactory emf;
EntityManager em;
@Resource
UserTransaction utx;
...
```

```
em = emf.createEntityManager();
try {
  utx.begin();
  em.persist(SomeEntity);
  em.merge(AnotherEntity);
  em.remove(ThirdEntity);
  utx.commit();
} catch (Exception e) {
  utx.rollback();
}
```

Finding Entities Using the EntityManager

The EntityManager.find method is used to look up entities in the data store by the entity's primary key:

```
@PersistenceContext
EntityManager em;
public void enterOrder(int custID, Order newOrder) {
    Customer cust = em.find(Customer.class, custID);
    cust.getOrders().add(newOrder);
    newOrder.setCustomer(cust);
}
```

Managing an Entity Instance's Lifecycle

You manage entity instances by invoking operations on the entity by means of an EntityManager instance. Entity instances are in one of four states: new, managed, detached, or removed.

- New entity instances have no persistent identity and are not yet associated with a persistence context.
- Managed entity instances have a persistent identity and are associated with a persistence context.
- Detached entity instances have a persistent identity and are not currently associated with a persistence context.
- Removed entity instances have a persistent identity, are associated with a persistent context, and are scheduled for removal from the data store.

Persisting Entity Instances

New entity instances become managed and persistent either by invoking the persist method or by a cascading persist operation invoked from related entities that have the cascade=PERSIST or cascade=ALL elements set in the relationship annotation. This means that the entity's data is stored to the database when the transaction associated with the persist operation is completed. If the entity is already managed, the persist operation is ignored, although the persist operation will cascade to related entities that have the cascade element set to PERSIST or ALL in the relationship annotation. If persist is called on a removed entity instance, the entity becomes managed. If the entity is detached, either persist will throw an IllegalArgumentException, or the transaction commit will fail.

```
@PersistenceContext
EntityManager em;
...
```

```
public LineItem createLineItem(Order order, Product product,
        int quantity) {
    LineItem li = new LineItem(order, product, quantity);
    order.getLineItems().add(li);
    em.persist(li);
    return li;
}
```

The persist operation is propagated to all entities related to the calling entity that have the cascade element set to ALL or PERSIST in the relationship annotation:

```
@OneToMany(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

Removing Entity Instances

Managed entity instances are removed by invoking the remove method or by a cascading remove operation invoked from related entities that have the cascade=REMOVE or cascade=ALL elements set in the relationship annotation. If the remove method is invoked on a new entity, the remove operation is ignored, although remove will cascade to related entities that have the cascade element set to REMOVE or ALL in the relationship annotation. If remove is invoked on a detached entity, either remove will throw an IllegalArgumentException, or the transaction commit will fail. If invoked on an already removed entity, remove will be ignored. The entity's data will be removed from the data store when the transaction is completed or as a result of the flush operation.

```
public void removeOrder(Integer orderId) {
    try {
        Order order = em.find(Order.class, orderId);
        em.remove(order);
    }...
```

In this example, all LineItem entities associated with the order are also removed, as Order.getLineItems has cascade=ALL set in the relationship annotation.

Synchronizing Entity Data to the Database

The state of persistent entities is synchronized to the database when the transaction with which the entity is associated commits. If a managed entity is in a bidirectional relationship with another managed entity, the data will be persisted, based on the owning side of the relationship.

To force synchronization of the managed entity to the data store, invoke the flush method of the EntityManager instance. If the entity is related to another entity and the relationship annotation has the cascade element set to PERSIST or ALL, the related entity's data will be synchronized with the data store when flush is called.

If the entity is removed, calling flush will remove the entity data from the data store.

Persistence Units

A persistence unit defines a set of all entity classes that are managed by EntityManager instances in an application. This set of entity classes represents the data contained within a single data store.

Persistence units are defined by the persistence.xml configuration file. The following is an example persistence.xml file:

```
<persistence>
   <persistence-unit name="OrderManagement">
      <description>This unit manages orders and customers.
         It does not rely on any vendor-specific features and can
         therefore be deployed to any persistence provider.
      </description>
      <jta-data-source>jdbc/MyOrderDB</jta-data-source>
      <jar-file>MyOrderApp.jar</jar-file>
      <class>com.widgets.Order</class>
      <class>com.widgets.Customer</class>
   </persistence-unit>
</persistence>
```

This file defines a persistence unit named OrderManagement, which uses a JTA-aware data source: jdbc/MyOrderDB. The jar-file and class elements specify managed persistence classes: entity classes, embeddable classes, and mapped superclasses. The jar-file element specifies JAR files that are visible to the packaged persistence unit that contain managed persistence classes, whereas the class element explicitly names managed persistence classes.

The jta-data-source (for JTA-aware data sources) and non-jta-data-source (for non-JTA-aware data sources) elements specify the global JNDI name of the data source to be used by the container.

The JAR file or directory whose META-INF directory contains persistence.xml is called the root of the persistence unit. The scope of the persistence unit is determined by the persistence unit's root. Each persistence unit must be identified with a name that is unique to the persistence unit's scope.

Persistent units can be packaged as part of a WAR or EJB JAR file or can be packaged as a JAR file that can then be included in an WAR or EAR file.

- If you package the persistent unit as a set of classes in an EJB JAR file, persistence.xml should be put in the EJB JAR's META-INF directory.
- If you package the persistence unit as a set of classes in a WAR file, persistence.xml should be located in the WAR file's WEB-INF/classes/META-INF directory.
- If you package the persistence unit in a JAR file that will be included in a WAR or EAR file, the JAR file should be located in either
  - The WEB-INF/lib directory of a WAR
  - The EAR file's library directory

Note - In the Java Persistence API 1.0, JAR files could be located at the root of an EAR file as the root of the persistence unit. This is no longer supported. Portable applications should use the EAR file's library directory as the root of the persistence unit.

Querying Entities

The Java Persistence API provides the following methods for querying entities.

- The Java Persistence query language (JPQL) is a simple, string-based language similar to SQL used to query entities and their relationships.
- The Criteria API is used to create typesafe queries using Java programming language APIs to query for entities and their relationships.
- Both JPQL and the Criteria API have advantages and disadvantages.

Just a few lines long, JPQL queries are typically more concise and more readable than Criteria queries. Developers familiar with SQL will find it easy to learn the syntax of JPQL. JPQL named queries can be defined in the entity class using a Java programming language annotation or in the application's deployment descriptor. JPQL queries are not typesafe, however, and require a cast when retrieving the query result from the entity manager. This means that type-casting errors may not be caught at compile time. JPQL queries don't support open-ended parameters.

Criteria queries allow you to define the query in the business tier of the application. Although this is also possible using JPQL dynamic queries, Criteria queries provide better performance because JPQL dynamic queries must be parsed each time they are called. Criteria queries are typesafe and therefore don't require casting, as JPQL queries do. The Criteria API is just another Java programming language API and doesn't require developers to learn the syntax of another query language. Criteria queries are typically more verbose than JPQL queries and require the developer to create several objects and perform operations on those objects before submitting the query to the entity manager.

The Java Persistence Query Language

The Java Persistence Query Language

The Java Persistence query language defines queries for entities and their persistent state. The query language allows you to write portable queries that work regardless of the underlying data store.

The query language uses the abstract persistence schemas of entities, including their relationships, for its data model and defines operators and expressions based on this data model. The scope of a query spans the abstract schemas of related entities that are packaged in the same persistence unit. The query language uses an SQL-like syntax to select objects or values based on entity abstract schema types and relationships among them.

This chapter relies on the material presented in earlier chapters. For conceptual information, The following topics are addressed here:

- Query Language Terminology

- Creating Queries Using the Java Persistence Query Language
- Simplified Query Language Syntax
- Example Queries
- Full Query Language Syntax

Query Language Terminology

The following list defines some of the terms referred to in this chapter.

- **Abstract schema**: The persistent schema abstraction (persistent entities, their state, and their relationships) over which queries operate. The query language translates queries over this persistent schema abstraction into queries that are executed over the database schema to which entities are mapped.
- **Abstract schema type**: The type to which the persistent property of an entity evaluates in the abstract schema. That is, each persistent field or property in an entity has a corresponding state field of the same type in the abstract schema. The abstract schema type of an entity is derived from the entity class and the metadata information provided by Java language annotations.
- **Backus-Naur Form (BNF)**: A notation that describes the syntax of high-level languages. The syntax diagrams in this chapter are in BNF notation.
- **Navigation**: The traversal of relationships in a query language expression. The navigation operator is a period.
- **Path expression**: An expression that navigates to an entity's state or relationship field.
- **State field**: A persistent field of an entity.
- **Relationship field**: A persistent field of an entity whose type is the abstract schema type of the related entity

Creating Queries Using the Java Persistence Query Language

The EntityManager.createQuery and EntityManager.createNamedQuery methods are used to query the datastore by using Java Persistence query language queries.

The createQuery method is used to create **dynamic queries**, which are queries defined directly within an application's business logic:

```
public List findWithName(String name) {

return em.createQuery(

    "SELECT c FROM Customer c WHERE c.name LIKE :custName")

    .setParameter("custName", name)
```

```
    .setMaxResults(10)
    .getResultList();
}
```

The createNamedQuery method is used to create **static queries**, or queries that are defined in metadata by using the javax.persistence.NamedQuery annotation. The name element of @NamedQuery specifies the name of the query that will be used with the createNamedQuery method. The query element of @NamedQuery is the query:

```
@NamedQuery(
    name="findAllCustomersWithName",
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName"
)
```

Here's an example of createNamedQuery, which uses the @NamedQuery:

```
@PersistenceContext
public EntityManager em;
...
customers = em.createNamedQuery("findAllCustomersWithName")
    .setParameter("custName", "Smith")
    .getResultList();
```

39.2.1 Named Parameters in Queries

Named parameters are query parameters that are prefixed with a colon (:). Named parameters in a query are bound to an argument by the following method:

```
javax.persistence.Query.setParameter(String name, Object value)
```

In the following example, the name argument to the findWithName business method is bound to the :custName named parameter in the query by calling Query.setParameter:

```
public List findWithName(String name) {
    return em.createQuery(
```

```
        "SELECT c FROM Customer c WHERE c.name LIKE :custName")
        .setParameter("custName", name)
        .getResultList();
}
```

Named parameters are case-sensitive and may be used by both dynamic and static queries.

39.2.2 Positional Parameters in Queries

You may use positional parameters instead of named parameters in queries. Positional parameters are prefixed with a question mark (?) followed by the numeric position of the parameter in the query. The method Query.setParameter(integer position, Object value) is used to set the parameter values.

In the following example, the findWithName business method is rewritten to use input parameters:

```
public List findWithName(String name) {
    return em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE ?1")
        .setParameter(1, name)
        .getResultList();
}
```

Input parameters are numbered starting from 1. Input parameters are case-sensitive, and may be used by both dynamic and static queries.

Simplified Query Language Syntax

This section briefly describes the syntax of the query language so that you can quickly move on to Example Queries. When you are ready to learn about the syntax in more detail, see Full Query Language Syntax.

39.3.1 Select Statements

A select query has six clauses: SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY. The SELECT and FROM clauses are required, but the WHERE, GROUP BY, HAVING, and ORDER BY clauses are optional. Here is the high-level BNF syntax of a query language select query:

```
QL_statement ::= select_clause from_clause
  [where_clause][groupby_clause][having_clause][orderby_clause]
```

The BNF syntax defines the following clauses.

- The SELECT clause defines the types of the objects or values returned by the query.
- The FROM clause defines the scope of the query by declaring one or more **identification variables**, which can be referenced in the SELECT and WHERE clauses. An identification variable represents one of the following elements:
  - The abstract schema name of an entity
  - An element of a collection relationship
  - An element of a single-valued relationship
  - A member of a collection that is the multiple side of a one-to-many relationship
- The WHERE clause is a conditional expression that restricts the objects or values retrieved by the query. Although the clause is optional, most queries have a WHERE clause.
- The GROUP BY clause groups query results according to a set of properties.
- The HAVING clause is used with the GROUP BY clause to further restrict the query results according to a conditional expression.
- The ORDER BY clause sorts the objects or values returned by the query into a specified order.

39.3.2 Update and Delete Statements

Update and delete statements provide bulk operations over sets of entities. These statements have the following syntax:

```
update_statement :: = update_clause [where_clause]

delete_statement :: = delete_clause [where_clause]
```

The update and delete clauses determine the type of the entities to be updated or deleted. The WHERE clause may be used to restrict the scope of the update or delete operation.

4 Example Queries

The following queries are from the Player entity of the roster application, which is documented in The roster Application.

39.4.1 Simple Queries

If you are unfamiliar with the query language, these simple queries are a good place to start.

39.4.1.1 A Basic Select Query

```
SELECT p

FROM Player p
```

- **Data retrieved**: All players.
- **Description**: The FROM clause declares an identification variable named p, omitting the optional keyword AS. If the AS keyword were included, the clause would be written as follows:

- FROM Player AS p

The Player element is the abstract schema name of the Player entity.

- **See also**: Identification Variables.

39.4.1.2 Eliminating Duplicate Values

```
SELECT DISTINCT p

FROM Player p

WHERE p.position = ?1
```

- **Data retrieved**: The players with the position specified by the query's parameter.
- **Description**: The DISTINCT keyword eliminates duplicate values.

  The WHERE clause restricts the players retrieved by checking their position, a persistent field of the Player entity. The ?1 element denotes the input parameter of the query.

- **See also**: Input Parameters and The DISTINCT Keyword.

39.4.1.3 Using Named Parameters

```
SELECT DISTINCT p

FROM Player p

WHERE p.position = :position AND p.name = :name
```

- **Data retrieved**: The players having the specified positions and names.
- **Description**: The position and name elements are persistent fields of the Player entity. The WHERE clause compares the values of these fields with the named parameters of the query, set using

the Query.setNamedParameter method. The query language denotes a named input parameter using a colon (:) followed by an identifier. The first input parameter is :position, the second is :name.

39.4.2 Queries That Navigate to Related Entities

In the query language, an expression can traverse, or navigate, to related entities. These expressions are the primary difference between the Java Persistence query language and SQL. Queries navigates to related entities, whereas SQL joins tables.

39.4.2.1 A Simple Query with Relationships

```
SELECT DISTINCT p

FROM Player p, IN (p.teams) t
```

- **Data retrieved**: All players who belong to a team.
- **Description**: The FROM clause declares two identification variables: p and t. The p variable represents the Player entity, and the t variable represents the related Team entity. The declaration for t references the previously declared p variable. The IN keyword signifies that teams is a collection of related entities. The p.teams expression navigates from a Player to its related Team. The period in the p.teams expression is the navigation operator.

You may also use the JOIN statement to write the same query:

```
SELECT DISTINCT p

FROM Player p JOIN p.teams t
```

This query could also be rewritten as:

```
SELECT DISTINCT p

FROM Player p

WHERE p.team IS NOT EMPTY
```

39.4.2.2 Navigating to Single-Valued Relationship Fields

Use the JOIN clause statement to navigate to a single-valued relationship field:

```
SELECT t

FROM Team t JOIN t.league l

WHERE l.sport = 'soccer' OR l.sport ='football'
```

In this example, the query will return all teams that are in either soccer or football leagues.

39.4.2.3 Traversing Relationships with an Input Parameter

```
SELECT DISTINCT p

FROM Player p, IN (p.teams) AS t

WHERE t.city = :city
```

- **Data retrieved**: The players whose teams belong to the specified city.
- **Description**: This query is similar to the previous example but adds an input parameter. The AS keyword in the FROM clause is optional. In the WHERE clause, the period preceding the persistent variable city is a delimiter, not a navigation operator. Strictly speaking, expressions can navigate to relationship fields (related entities) but not to persistent fields. To access a persistent field, an expression uses the period as a delimiter.

  Expressions cannot navigate beyond (or further qualify) relationship fields that are collections. In the syntax of an expression, a collection-valued field is a terminal symbol. Because the teams field is a collection, the WHERE clause cannot specify p.teams.city (an illegal expression).
- **See also**: Path Expressions.

39.4.2.4 Traversing Multiple Relationships

```
SELECT DISTINCT p

FROM Player p, IN (p.teams) t

WHERE t.league = :league
```

- **Data retrieved**: The players who belong to the specified league.
- **Description**: The expressions in this query navigate over two relationships. The p.teams expression navigates the Player-Team relationship, and the t.league expression navigates the Team-League relationship.

  In the other examples, the input parameters are String objects; in this example, the parameter is an object whose type is a League. This type matches the league relationship field in the comparison expression of the WHERE clause.

39.4.2.5 Navigating According to Related Fields

```
SELECT DISTINCT p

FROM Player p, IN (p.teams) t
```

```
WHERE t.league.sport = :sport
```

- **Data retrieved**: The players who participate in the specified sport.
- **Description**: The sport persistent field belongs to the League entity. To reach the sport field, the query must first navigate from the Player entity to Team (p.teams) and then from Team to the League entity (t.league). Because it is not a collection, the league relationship field can be followed by the sport persistent field.

39.4.3 Queries with Other Conditional Expressions

Every WHERE clause must specify a conditional expression, of which there are several kinds. In the previous examples, the conditional expressions are comparison expressions that test for equality. The following examples demonstrate some of the other kinds of conditional expressions. For descriptions of all conditional expressions, see WHERE Clause.

39.4.3.1 The LIKE Expression

```
SELECT p

FROM Player p

WHERE p.name LIKE 'Mich%'
```

- **Data retrieved**: All players whose names begin with "Mich."
- **Description**: The LIKE expression uses wildcard characters to search for strings that match the wildcard pattern. In this case, the query uses the LIKE expression and the % wildcard to find all players whose names begin with the string "Mich." For example, "Michael" and "Michelle" both match the wildcard pattern.
- **See also**: LIKE Expressions.

39.4.3.2 The IS NULL Expression

```
SELECT t

FROM Team t

WHERE t.league IS NULL
```

- **Data retrieved**: All teams not associated with a league.
- **Description**: The IS NULL expression can be used to check whether a relationship has been set between two entities. In this case, the query checks whether the teams are associated with any leagues and returns the teams that do not have a league.
- **See also**: NULL Comparison Expressions and NULL Values.

39.4.3.3 The IS EMPTY Expression

```
SELECT p

FROM Player p

WHERE p.teams IS EMPTY
```

- **Data retrieved**: All players who do not belong to a team.
- **Description**: The teams relationship field of the Player entity is a collection. If a player does not belong to a team, the teams collection is empty, and the conditional expression is TRUE.
- **See also**: Empty Collection Comparison Expressions.

39.4.3.4 The BETWEEN Expression

```
SELECT DISTINCT p

FROM Player p

WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
```

- **Data retrieved**: The players whose salaries fall within the range of the specified salaries.
- **Description**: This BETWEEN expression has three arithmetic expressions: a persistent field (p.salary) and the two input parameters (:lowerSalary and :higherSalary). The following expression is equivalent to the BETWEEN expression:

```
- p.salary >= :lowerSalary AND p.salary <= :higherSalary
```

- **See also**: BETWEEN Expressions.

39.4.3.5 Comparison Operators

```
SELECT DISTINCT p1

FROM Player p1, Player p2

WHERE p1.salary > p2.salary AND p2.name = :name
```

- **Data retrieved**: All players whose salaries are higher than the salary of the player with the specified name.
- **Description**: The FROM clause declares two identification variables (p1 and p2) of the same type (Player). Two identification variables are needed because the WHERE clause compares the salary of one player (p2) with that of the other players (p1).

- **See also**: Identification Variables.

39.4.4 Bulk Updates and Deletes

The following examples show how to use the UPDATE and DELETE expressions in queries. UPDATE and DELETE operate on multiple entities according to the condition or conditions set in the WHERE clause. The WHERE clause in UPDATE and DELETE queries follows the same rules as SELECT queries.

39.4.4.1 Update Queries

```
UPDATE Player p

SET p.status = 'inactive'

WHERE p.lastPlayed < :inactiveThresholdDate
```

- **Description**: This query sets the status of a set of players to inactive if the player's last game was longer ago than the date specified in inactiveThresholdDate.

39.4.4.2 Delete Queries

```
DELETE

FROM Player p

WHERE p.status = 'inactive'

AND p.teams IS EMPTY
```

- **Description**: This query deletes all inactive players who are not on a team.

Full Query Language Syntax

This section discusses the query language syntax, as defined in the Java Persistence API 2.0 specification available at http://jcp.org/en/jsr/detail?id=317. Much of the following material paraphrases or directly quotes the specification.

1 BNF Symbols

Table 39-1 BNF Symbol Summary

Symbol	Description
::=	The element to the left of the symbol is defined by the constructs on the right.

Symbol	Description
*	The preceding construct may occur zero or more times.
{...}	The constructs within the braces are grouped together.
[...]	The constructs within the brackets are optional.
|	An exclusive OR.
BOLDFACE	A keyword; although capitalized in the BNF diagram, keywords are not case-sensitive.
White space	A whitespace character can be a space, a horizontal tab, or a line fee

2.BNF Grammar of the Java Persistence Query Language

Using the Criteria API to Create Queries

The Criteria API is used to define queries for entities and their persistent state by creating query-defining objects. Criteria queries are written using Java programming language APIs, are typesafe, and are portable. Such queries work regardless of the underlying data store.

The following topics are addressed here:

- Overview of the Criteria and Metamodel APIs
- Using the Metamodel API to Model Entity Classes
- Using the Criteria API and Metamodel API to Create Basic Typesafe Queries

Direction in Entity Relationships

The direction of a relationship can be either bidirectional or unidirectional. A bidirectional relationship has both an owning side and an inverse side. A unidirectional relationship has only an owning side. The owning side of a relationship determines how the Persistence runtime makes updates to the relationship in the database.

Bidirectional Relationships

In a *bidirectional* relationship, each entity has a relationship field or property that refers to the other entity. Through the relationship field or property, an entity class's code can access its related object. If an entity has a related field, the entity is said to "know" about its related object. For example, if Order knows what LineItem instances it has and if LineItem knows what Order it belongs to, they have a bidirectional relationship.

Bidirectional relationships must follow these rules.

- The inverse side of a bidirectional relationship must refer to its owning side by using the mappedBy element of the @OneToOne, @OneToMany, or @ManyToMany annotation. The mappedBy element designates the property or field in the entity that is the owner of the relationship.
- The many side of many-to-one bidirectional relationships must not define the mappedBy element. The many side is always the owning side of the relationship.
- For one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.
- For many-to-many bidirectional relationships, either side may be the owning side.

Unidirectional Relationships

In a *unidirectional* relationship, only one entity has a relationship field or property that refers to the other. For example, LineItem would have a relationship field that identifies Product, but Product would not have a relationship field or property for LineItem. In other words, LineItem knows about Product, but Product doesn't know which LineItem instances refer to it.

Queries and Relationship Direction

Java Persistence query language and Criteria API queries often navigate across relationships. The direction of a relationship determines whether a query can navigate from one entity to another. For example, a query can navigate from LineItem to Product but cannot navigate in the opposite direction. For Order and LineItem, a query could navigate in both directions because these two entities have a bidirectional relationship.

Cascade Operations and Relationships

Entities that use relationships often have dependencies on the existence of the other entity in the relationship. For example, a line item is part of an order; if the order is deleted, the line item also should be deleted. This is called a cascade delete relationship.

The javax.persistence.CascadeType enumerated type defines the cascade operations that are applied in the cascade element of the relationship annotations. Table lists the cascade operations for entities.

Table Cascade Operations for Entities

Cascade Operation	Description
ALL	All cascade operations will be applied to the parent entity's related entity. All is equivalent to specifying cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}
DETACH	If the parent entity is detached from the persistence context, the related entity will also be detached.
MERGE	If the parent entity is merged into the persistence context, the related entity will also be merged.
PERSIST	If the parent entity is persisted into the persistence context, the related entity will also be persisted.
REFRESH	If the parent entity is refreshed in the current persistence context, the related entity will also be refreshed.
REMOVE	If the parent entity is removed from the current persistence context, the related entity will also be removed.

Cascade delete relationships are specified using the cascade=REMOVE element specification for @OneToOne and @OneToMany relationships. For example:

```
@OneToMany(cascade=REMOVE, mappedBy="customer")
public Set<Order> getOrders() { return orders; }
```

Orphan Removal in Relationships

When a target entity in one-to-one or one-to-many relationship is removed from the relationship, it is often desirable to cascade the remove operation to the target entity. Such target entities are considered "orphans," and the orphanRemoval attribute can be used to specify that orphaned entities should be removed. For example, if an order has many line items and one of them is removed from the order, the removed line item is considered an orphan. If orphanRemoval is set to true, the line item entity will be deleted when the line item is removed from the order.

The orphanRemoval attribute in @OneToMany and @oneToOne takes a Boolean value and is by default false.

The following example will cascade the remove operation to the orphaned customer entity when it is removed from the relationship:

```
@OneToMany(mappedBy="customer", orphanRemoval="true")
public List<Order> getOrders() { ... }
```