Sun Microsystems

JSR 220: Enterprise JavaBeans^{TM},Version 3.0

Java Persistence API

EJB 3.0 Expert Group

Specification Lead:

Linda DeMichiel, Sun Microsystems

Please send comments to: ejb3-pdr-feedback@sun.com

Version 3.0, Public Draft

June 25, 2005

Specification: JSR-220, Enterprise Java Beans ("Specification")
Version: 3.0
Status: Pre-FCS, Public Review
Release: June 27, 2005

LIMITED EVALUATION LICENSE

Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Sun's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation. This includes (i) developing applications intended to run on an implementation of the Specification, provided that such applications do not themselves implement any portion(s) of the Specification, and (ii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Technology. No license of any kind is granted hereunder for any other purpose including, for example, creating and distributing implementations of the Specification, modifying the Specification (other than to the extent of your fair use rights), or distributing the Specification to third parties. Also, no right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. If you wish to create and distribute an implementation of the Specification, a license for that purpose is available at http://www.jcp.org. The foregoing license is expressly conditioned on your acting within its scope, and will terminate immediately without notice from Sun if you breach the Agreement or act outside the scope of the licenses granted above. Java, and Java-related logos, marks and names are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT (INCLUDING AS A CONSEQUENCE OF ANY PRACTICE OR IMPLEMENTATION OF THE SPECIFICATION), OR THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE. This document does not represent any commitment to release or implement any portion of the Specification in any product. In addition, the Specification could include technical inaccuracies or typographical errors.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, RELATED IN ANY WAY TO YOUR HAVING OR USING THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Acknowledgments

I would like to specially recognize Gavin King and the Hibernate community for their leadership in achieving a developer-driven solution to the problem of object/relational persistence. I would like to thank Gavin for bringing his energy, his experience, and his vision of a developer-centric persistence solution to the EJB 3 effort, and for converging the Hibernate community in support of this work. Without his many significant contributions, it would not have been possible for this work to be where it is today.

I would like to extend my thanks to Mike Keith for sharing with the expert group his extensive experience with Oracle TopLink, for all his technical proposals to the group, and for his many other technical contributions to the success of this specification.

Table of Contents

List of Tables

Chapter 1 Introduction

This document is the specification of the Java API for the management of persistence and object/relational mapping with J2EE and J2SE.

This persistence API—together with the query language and object/relational mapping metadata defined in this document—is required to be supported under Enterprise JavaBeans 3.0. It is also targeted at being used stand-alone with J2SE.

Leading experts throughout the entire Java community have come together to build this Java persistence standard. This work incorporates contributions from the Hibernate, TopLink, and JDO communities, as well as from the EJB community.

1.1 Expert Group

This work is being conducted as part of JSR-220 under the Java Community Process Program. This specification is the result of the collaborative work of the members of the JSR 220 Expert Group. These include the following present and former expert group members: Apache Software Foundation: Jeremy Boynes; BEA: Seth White; Borland: Jishnu Mitra; E.piphany: Karthik Kothandaraman; Fujitsu-Siemens: Anton Vorsamer; Google: Cedric Beust; IBM: Jim Knutson, Randy Schnier; IONA: Conrad O'Dea; Ironflare: Hani Suleiman; JBoss: Gavin King, Bill Burke, Marc Fleury; Macromedia: Hemant Khandelwal; Nokia: Vic Zaroukian; Novell: YongMin Chen; Oracle: Michael Keith, Debu Panda, Olivier Caudron; Pramati: Deepak Anupalli; SAP: Steve Winkler, Umit Yalcinalp; SAS Institute: Rob Saccoccio; SeeBeyond: Ugo Corda; SolarMetric: Patrick Linskey; Sun Microsystems: Linda DeMichiel, Mark Reinhold; Sybase: Evan Ireland; Tibco: Shivajee Samdarshi; Tmax Soft: Woo Jin Kim; Versant: David Tinker; Xcalia: Eric Samson; Reza Behforooz; Emmanuel Bernard; Wes Biggs; David Blevins; Scott Crawford; Geoff Hendrey; Oliver Ihns; Oliver Kamps; Richard Monson-Haefel; Dirk Reinshagen; Carl Rosenberger; Suneet Shah.

1.2 Document Conventions

The regular Times font is used for information that is prescriptive by the EJB specification.

The italic Times font is used for paragraphs that contain descriptive information, such as notes describing typical use, or notes clarifying the text with prescriptive specification.

`The Courier font is used for code examples.`

The Helvetica font is used to specify the BNF of EJB QL.

This document is written in terms of the use of Java language metadata annotations to specify the semantics of persistent classes and their object/relational mapping. An XML descriptor (as specified in Chapter 10) may be used as an alternative to annotations. The elements of this descriptor mirror the annotations and have the same semantics.

Chapter 2 Entities

An entity is a lightweight persistent domain object.

The primary programming artifact is the entity class. An entity class may make use of auxiliary classes that serve as helper classes or that are used to represent the state of the entity.

2.1 Requirements on the Entity Class

The entity class must be annotated with the `Entity` annotation or denoted in the XML descriptor as an entity.

The entity class must have a no-arg constructor. The entity class may have other constructors as well. The no-arg constructor must be public or protected.

If an entity instance is to be passed by value as a detached object (e.g., through a remote interface), the entity class must implement the `Serializable` interface.

The entity class must not be final. No methods of the entity class may be final.

Entities support inheritance, polymorphic associations, and polymorphic queries. Both abstract and concrete classes can be entities. Entities may extend non-entity classes as well as entity classes, and non-entity classes may extend entity classes.

The state of an entity is represented by instance variables, which may correspond to JavaBeans properties. An instance variable may be directly accessed only from within the methods of the entity by the entity instance itself. Instance variables must not be accessed by clients of the entity. The state of the entity is available to clients only through the entity's accessor methods (getter/setter methods) or other business methods. Instance variables must be private, protected, or package visibility.

2.1.1 Persistent Fields and Properties

The persistent state of an entity is accessed by the persistence provider runtime[1] either via JavaBeans style property accessors or via instance variables.

- If the entity is annotated with the annotation element value `access=FIELD`, the persistence provider runtime accesses instance variables directly and all non-`transient` instance variables that are not annotated with the `Transient` annotation are persistent.
- If the entity is annotated with the annotation element value `access=PROPERTY`, or if the `access` annotation element value is not specified, the persistence provider runtime accesses persistent state via the property accessor methods and all properties not annotated with the `Transient` annotation are persistent. The property accessor methods must be public or protected.
- When the `FIELD` access type is used, the object/relational mapping annotations for the entity class annotate the instance variables. When the `PROPERTY` access type is used, the object/relational mapping annotations for the entity class annotate the getter property accessors.[2]

It is required that the entity class follow the method conventions for a JavaBean when persistent properties are used.

In this case, for every persistent property *property* of type *T* of the entity, there is a getter method, *getProperty*, and setter method *setProperty*. For boolean properties, *isProperty* is an alternative name for the getter method.

For single-valued persistent properties, these method signatures are:

- T getProperty()
- void setProperty(T t)

[1] The term "persistence provider runtime" refers to the runtime environment of the persistence implementation. In EJB environments, this may be the EJB container or a third-party persistence implementation integrated with it.

[2] Note that the order in which the persistence provider runtime calls these methods when loading or storing persistent state is not defined. Business logic contained in such methods therefore cannot rely upon a specific invocation order.

Collection-valued persistent fields and properties must be defined in terms of `java.util.Collection` interfaces regardless of whether the entity class otherwise adheres to the JavaBeans conventions noted above. The following collection interfaces are supported: `java.util.Collection`, `java.util.Set`, `java.util.List`[3], `java.util.Map`.

For collection-valued persistent properties, type *T* must be one of these Collection interface types in the method signatures above. Generic variants of these Collection types may also be used (for example, `Set<Order>`).

In addition to returning and setting the persistent state of the instance, the property accessor methods may contain other business logic as well, for example, to perform validation.

> *Note that the persistence runtime executes this validation logic when the access type* `PROPERTY` *is specified or defaulted. Caution should be exercised in adding business logic to the accessor methods when the* `PROPERTY` *access type is used.*

Runtime exceptions thrown by property accessor methods will cause the current transaction to be rolled back. Application exceptions thrown by such methods when used by the persistence runtime to load or store persistent state will cause the persistence runtime to rollback the transaction and to throw a PersistenceException that wraps the application exception.

Entity subclasses may override the property accessor methods. However, portable applications must not override the object/relational mapping metadata that applies to the persistent fields or properties of entity superclasses.

The persistent fields or properties of an entity may be of the following types: Java primitive types; `java.lang.String`; other Java serializable types (including wrappers of the primitive types, `java.math.BigInteger`, `java.math.BigDecimal`, `java.util.Date`, `java.util.Calendar`[4], `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`, user-defined serializable types, `byte[]`, `Byte[]`, `char[]`, and `Character[]`); enums; entity types and/or collections of entity types; and embeddable classes (see section 2.1.5).

Object/relational mapping metadata may be specified to customize the object-relational mapping, and the loading and storing of the entity state and relationships. See Chapter 9.

[3] Portable applications should not expect the order of lists to be maintained across persistence contexts unless the `@OrderBy` construct is used and modications to the list observe the specified ordering.

[4] Note that an instance must of Calendar must be fully initialized for the type that it is mapped to.

2.1.2 Example

```
@Entity
public class Customer implements Serializable {

  private Long id;

  private String name;

  private Address address;

  private Collection<Order> orders = new HashSet();

  private Set<PhoneNumber> phones = new HashSet();

  // No-arg constructor
  public Customer() {}

  @Id
  public Long getId() {
    return id;
  }

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

  public String getName() {
    return name;
  }

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

  public Address getAddress() {
    return address;
  }

  public void setAddress(Address address) {
    this.address = address;
  }

@OneToMany
  public Collection<Order> getOrders() {
    return orders;
  }

  public void setOrders(Collection<Order> orders) {
    this.orders = orders;
  }

  @ManyToMany
  public Set<PhoneNumber> getPhones() {
    return phones;
  }

  public void setPhones(Set<PhoneNumber> phones) {
    this.phones = phones;
  }
```

```
    // Business method to add a phone number to the customer
    public void addPhone(PhoneNumber phone) {
      this.getPhones().add(phone);
      // Set the phone’s ref to this customer
      phone.setCustomer(this);
    }
  }

  }
```

2.1.3 Entity Instance Creation

Entity instances are created by means of the `new` operation. An entity instance, when first created by `new` is not yet persistent. An instance becomes persistent by means of the `EntityManager` API. The lifecycle of entity instances is described in Section 3.2.

2.1.4 Primary Keys and Entity Identity

Every entity must have a primary key.

A simple (i.e., non-composite) primary key must correspond to a single persistent field or property of the entity class. A composite primary key must correspond to either a single persistent field or property or to a set of such fields or properties as described below. A primary key class must be defined to represent a composite primary key. Composite primary keys typically arise when mapping from legacy databases when the database key is comprised of several columns.

The primary key (or field or property of a composite primary key) must be one of the following types: any Java primitive type; any primitive wrapper type; `java.lang.String`; `java.util.Date`; `java.sql.Date`. In general, however, approximate numeric types (e.g., floating point types) should never be used in primary keys.

Both field and property access is allowed for primary key classes, as for entity classes.

The following rules apply for composite primary keys.

- The primary key class must be public and must have a public no-arg constructor.
- If `access=PROPERTY`, the properties of the primary key class must be public or protected.
- The primary key class must be serializable.
- The primary key class must define `equals` and `hashCode` methods. The semantics of value equality for these methods must be consistent with the database equality for the database types to which the key is mapped.
- A composite primary key may either be represented and mapped as an embeddable class or may be represented and mapped to multiple fields or properties of the entity class.
- If the composite primary key class is mapped to multiple fields or properties of the entity class, then the names of primary key fields or properties in the primary key class and those of the

entity class must correspond and their types must be the same. It is permitted for the entity class and the primary key class to use different access types (`PROPERTY` or `FIELD`).

- The application must not change the value of the primary key. The behavior is undefined if this occurs.[5]

2.1.5 Embeddable Classes

An entity may use other fine-grained classes to represent entity state. Instances of these classes, unlike entity instances themselves, do not have persistent identity. Instead, they exist only as embedded objects of the entity to which they belong. Such embedded objects belong strictly to their owning entity, and are not sharable across persistent entities. Attempting to share an embedded object across entities has undefined semantics. Because these objects have no persistent identity, they are typically mapped together with the entity instance to which they belong.[6]

Embeddable classes must adhere to the requirements specified in Section 2.1 for entities with the exception that embeddable classes are not annotated as `Entity`.

2.1.6 Mapping Defaults for Non-Relationship Fields or Properties

If a persistent field or property other than a relationship property is not annotated with one of the mapping annotations defined in Chapter 9 (or equivalent mapping information is not specified in the XML descriptor), the following default mapping rules are applied in order:

- If the type is a class that is annotated with the `@Embeddable` annotation, it is mapped as `@Embedded`.

- If the type of the field or property is one of the following, it is mapped as `@Basic`: Java primitive types, wrappers of the primitive types, `java.lang.String`, `java.math.BigInteger`, `java.math.BigDecimal`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`, `byte[]`, `Byte[]`, `char[]`, `Character[]`, enums, any other type that implements Serializable.

It is an error if no annotation is present and none of the above rules apply.

2.1.7 Entity Relationships

Relationships among entities may be one-to-one, one-to-many, many-to-one, or many-to-many. Relationships are polymorphic.

[5] The implementation may, but is not required to, throw an exception if this occurs.

[6] Support for collections of embedded objects and for the polymorphism and inheritance of embeddable classes will be required in a future release of this specification.

If there is an association between two entities, one of the following relationship modeling annotations must be applied to the corresponding persistent property or instance variable of the referencing entity: `OneToOne`, `OneToMany`, `ManyToOne`, `ManyToMany`. For associations that do not specify the target type (e.g., where Java generic types are not used for collections), it is necessary to specify the entity that is the target of the relationship.

These annotations mirror common practice in relational database schema modeling. The use of the relationship modeling annotations allows the object/relationship mapping of associations to the relational database schema to be fully defaulted, to provide an ease-of-development facility. This is described in Section 2.1.8, "Relationship Mapping Defaults".

Relationships may be 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 the updates to the relationship in the database, as described in section 3.2.3.

The following rules apply to bidirectional relationships:

- The inverse side of a bidirectional relationship must refer to its owning side by use of 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 one-to-many / many-to-one bidirectional relationships must be the owning side, hence the `mappedBy` element cannot be specified on the `ManyToOne` annotation.
- 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.

The relationship modeling annotation constrains the use of the `cascade=REMOVE` specification. The `cascade=REMOVE` specification should only be applied to associations that are specified as `OneToOne` or `OneToMany`. Applications that apply `cascade=REMOVE` to other associations are not portable.

Additional mapping annotations (e.g., column and table mapping annotations) may be specified to override or further refine the default mappings described in Section 2.1.8. For example, a foreign key mapping may be used for a unidirectional one-to-many mapping. Any such overriding must be consistent with the relationship modeling annotation that is specified. For example, if a many-to-one relationship mapping is specified, it is not permitted to specify a unique key constraint on the foreign key for the relationship. Such schema-level mapping annotations must be specified on the owning side of the relationship.

The persistence provider handles the object-relational mapping of the relationships, including their loading and storing to the database as specified in the metadata of the entity class, and the referential integrity of the relationships as specified in the database (e.g., by foreign key constraints).

2.1.8 Relationship Mapping Defaults

This section describes the mapping defaults that apply to the use of the `OneToOne`, `OneToMany`, `ManyToOne`, and `ManyToMany` relationship modeling annotations. The same mapping defaults apply when the XML descriptor is used to denote the relationship cardinalities.

2.1.8.1 Bidirectional OneToOne Relationships

Assuming that:

Entity A references a single instance of Entity B.

Entity B references a single instance of Entity A.

Entity A is specified as the owner of the relationship.

The following mapping defaults apply:

Entity A is mapped to a table named `A`.

Entity B is mapped to a table named `B`.

Table `A` contains a foreign key to table `B`. The foreign key column name is formed as the concatenation of the following: the name of the relationship property or field of entity A; "_"; the name of the primary key column in table `B`. The foreign key column has the same type as the primary key of table `B` and there is a unique key constraint on it.

Example:

```
@Entity
public class Employee {
   private Cubicle assignedCubicle;

   @OneToOne
   public Cubicle getAssignedCubicle() {
     return assignedCubicle;
   }
   public void setAssignedCubicle(Cubicle cubicle) {
     this.assignedCubicle = cubicle;
   }
  ...
}


@Entity
public class Cubicle {
    private Employee residentEmployee;

    @OneToOne(mappedBy="assignedCubicle")
    public Employee getResidentEmployee() {
      return residentEmployee;
    }
    public void setResidentEmployee(Employee employee) {
      this.residentEmployee = employee;
    }
  ...
}
```

In this example:

Entity `Employee` references a single instance of Entity `Cubicle`.

Entity `Cubicle` references a single instance of Entity `Employee`.

Entity `Employee` is the owner of the relationship.

The following mapping defaults apply:

Entity `Employee` is mapped to a table named `EMPLOYEE`.

Entity `Cubicle` is mapped to a table named `CUBICLE`.

Table `EMPLOYEE` contains a foreign key to table `CUBICLE`. The foreign key column is named `ASSIGNEDCUBICLE_`<PK of CUBICLE>, where <PK of CUBICLE> denotes the name of the primary key column of table `CUBICLE`. The foreign key column has the same type as the primary key of `CUBICLE`, and there is a unique key constraint on it.

2.1.8.2 Bidirectional ManyToOne / OneToMany Relationships

Assuming that:

Entity A references a single instance of Entity B.

Entity B references a collection of Entity A.

Entity A must be the owner of the relationship.

The following mapping defaults apply:

Entity A is mapped to a table named `A`.

Entity B is mapped to a table named `B`.

Table `A` contains a foreign key to table `B`. The foreign key column name is formed as the concatenation of the following: the name of the relationship property or field of entity A; "`_`"; the name of the primary key column in table `B`. The foreign key column has the same type as the primary key of table `B`.

Example:

```
@Entity
public class Employee {
   private Department department;

   @ManyToOne
   public Department getDepartment() {
     return department;
   }
   public void setDepartment(Department department) {
     this.department = department;
   }
  ...
}


@Entity
public class Department {
   private Collection<Employee> employees = new HashSet();

   @OneToMany(mappedBy="department")
   public Collection<Employee> getEmployees() {
     return employees;
   }

   public void setEmployees(Collection<Employee> employees) {
     this.employees = employees;
   }
  ...
}
```

In this example:

Entity `Employee` references a single instance of Entity `Department`.

Entity `Department` references a collection of Entity `Employee`.

Entity `Employee` is the owner of the relationship.

The following mapping defaults apply:

Entity `Employee` is mapped to a table named `EMPLOYEE`.

Entity `Department` is mapped to a table named `DEPARTMENT`.

Table `EMPLOYEE` contains a foreign key to table `DEPARTMENT`. The foreign key column is named `DEPARTMENT_`<PK of DEPARTMENT>, where <PK of DEPARTMENT> denotes the name of the primary key column of table `DEPARTMENT`. The foreign key column has the same type as the primary key of `DEPARTMENT`.

2.1.8.3 Unidirectional Single-Valued Relationships

Assuming that:

Entity A references a single instance of Entity B.

Entity B does not reference Entity A.

A unidirectional relationship has only an owning side, which in this case must be Entity A.

The unidirectional single-valued relationship modeling case can be specified as either a unidirectional `OneToOne` or as a unidirectional `ManyToOne` relationship.

2.1.8.3.1 Unidirectional OneToOne Relationships

The following mapping defaults apply:

Entity A is mapped to a table named `A`.

Entity B is mapped to a table named `B`.

Table `A` contains a foreign key to table `B`. The foreign key column name is formed as the concatenation of the following: the name of the relationship property or field of entity A; "_"; the name of the primary key column in table `B`. The foreign key column has the same type as the primary key of table `B` and there is a unique key constraint on it.

Example:

```
@Entity
public class Employee {
   private TravelProfile profile;

   @OneToOne
   public TravelProfile getProfile() {
     return profile;
   }
   public void setProfile(TravelProfile profile) {
     this.profile = profile;
   }
  ...
}


@Entity
public class TravelProfile {
   ...
}
```

In this example:

Entity `Employee` references a single instance of Entity `TravelProfile`.

Entity `TravelProfile` does not reference Entity `Employee`.

Entity `Employee` is the owner of the relationship.

The following mapping defaults apply:

Entity `Employee` is mapped to a table named `EMPLOYEE`.

Entity `TravelProfile` is mapped to a table named `TRAVELPROFILE`.

Table EMPLOYEE contains a foreign key to table TRAVELPROFILE. The foreign key column is named PROFILE_<PK of TRAVELPROFILE>, where <PK of TRAVELPROFILE> denotes the name of the primary key column of table TRAVELPROFILE. The foreign key column has the same type as the primary key of TRAVELPROFILE, and there is a unique key constraint on it.

2.1.8.3.2 Unidirectional ManyToOne Relationships

The following mapping defaults apply:

Entity A is mapped to a table named A.

Entity B is mapped to a table named B.

Table A contains a foreign key to table B. The foreign key column name is formed as the concatenation of the following: the name of the relationship property or field of entity A; "_"; the name of the primary key column in table B. The foreign key column has the same type as the primary key of table B.

Example:

```
@Entity
public class Employee {
   private Address address;


   @ManyToOne
   public Address getAddress() {
     return address;
   }
   public void setAddress(Address address) {
     this.address = address;
   }
  ...
}

@Entity
public class Address {
   ...
}
```

In this example:

Entity Employee references a single instance of Entity Address.

Entity Address does not reference Entity Employee.

Entity Employee is the owner of the relationship.

The following mapping defaults apply:

Entity Employee is mapped to a table named EMPLOYEE.

Entity Address is mapped to a table named ADDRESS.

Table EMPLOYEE contains a foreign key to table ADDRESS. The foreign key column is named ADDRESS_<PK of ADDRESS>, where <PK of ADDRESS> denotes the name of the primary key column of table ADDRESS. The foreign key column has the same type as the primary key of ADDRESS.

2.1.8.4 Bidirectional ManyToMany Relationships

Assuming that:

Entity A references a collection of Entity B.

Entity B references a collection of Entity A.

Entity A is the owner of the relationship.

The following mapping defaults apply:

Entity A is mapped to a table named A.

Entity B is mapped to a table named B.

There is a join table that is named A_B (owner name first). This join table has two foreign key columns. One foreign key column refers to table A and has the same type as the primary key of table A. The name of this foreign key column is formed as the concatenation of the following: the name of the relationship property or field of entity B; "_"; the name of the primary key column in table A. The other foreign key column refers to table B and has the same type as the primary key of table B. The name of this foreign key column is formed as the concatenation of the following: the name of the relationship property or field of entity A; "_"; the name of the primary key column in table B.

Example:

```
@Entity
public class Project {
   private Collection<Employee> employees;

   @ManyToMany
   public Collection<Employee> getEmployees() {
     return employees;
   }

   public void setEmployees(Collection<Employee> employees) {
     this.employees = employees;
   }
   ...
}

@Entity
public class Employee {
   private Collection<Project> projects;

   @ManyToMany(mappedBy="employees")
   public Collection<Project> getProjects() {
     return projects;
   }

   public void setProjects(Collection<Project> projects) {
     this.projects = projects;
   }
  ...
}
```

In this example:

Entity `Project` references a collection of Entity `Employee`.

Entity `Employee` references a collection of Entity `Project`.

Entity `Project` is the owner of the relationship.

The following mapping defaults apply:

Entity `Project` is mapped to a table named `PROJECT`.

Entity `Employee` is mapped to a table named `EMPLOYEE`.

There is a join table that is named `PROJECT_EMPLOYEE` (owner name first). This join table has two foreign key columns. One foreign key column refers to table `PROJECT` and has the same type as the primary key of `PROJECT`. The name of this foreign key column is `PROJECTS_`<PK of PROJECT>, where <PK of PROJECT> denotes the name of the primary key column of table `PROJECT`. The other foreign key column refers to table `EMPLOYEE` and has the same type as the primary key of `EMPLOYEE`. The name of this foreign key column is `EMPLOYEES_`<PK of EMPLOYEE>, where <PK of EMPLOYEE> denotes the name of the primary key column of table `EMPLOYEE`.

2.1.8.5 Unidirectional Multi-Valued Relationships

Assuming that:

Entity A references a collection of Entity B.

Entity B does not reference Entity A.

A unidirectional relationship has only an owning side, which in this case must be Entity A.

The unidirectional multi-valued relationship modeling case can be specified as either a unidirectional `OneToMany` or as a unidirectional `ManyToMany` relationship.

2.1.8.5.1 Unidirectional OneToMany Relationships

The following mapping defaults apply:

Entity A is mapped to a table named `A`.

Entity B is mapped to a table named `B`.

There is a join table that is named `A_B` (owner name first). This join table has two foreign key columns. One foreign key column refers to table `A` and has the same type as the primary key of table `A`. The name of this foreign key column is formed as the concatenation of the following: the name of entity A; "_"; the name of the primary key column in table `A`. The other foreign key column refers to table `B` and has the same type as the primary key of table `B` and there is a unique key constraint on it. The name of this foreign key column is formed as the concatenation of the following: the name of the relationship property or field of entity A; "_"; the name of the primary key column in table `B`.

Example:

```
@Entity
public class Employee {
   private Collection<AnnualReview> annualReviews;

   @OneToMany
   public Collection<AnnualReview> getAnnualReviews() {
     return annualReviews;
   }

   public void setAnnualReviews(Collection<AnnualReview> annualRe-
views) {
     this.annualReviews = annualReviews;
   }
  ...
}


@Entity
public class AnnualReview {
   ...
}
```

In this example:

Entity Employee references a collection of Entity AnnualReview.

Entity AnnualReview does not reference Entity Employee.

Entity Employee is the owner of the relationship.

The following mapping defaults apply:

Entity Employee is mapped to a table named EMPLOYEE.

Entity AnnualReview is mapped to a table named ANNUALREVIEW.

There is a join table that is named EMPLOYEE_ANNUALREVIEW (owner name first). This join table has two foreign key columns. One foreign key column refers to table EMPLOYEE and has the same type as the primary key of EMPLOYEE. This foreign key column is named EMPLOYEE_<PK of EMPLOYEE>, where <PK of EMPLOYEE> denotes the name of the primary key column of table EMPLOYEE. The other foreign key column refers to table ANNUALREVIEW and has the same type as the primary key of ANNUALREVIEW. This foreign key column is named ANNUALREVIEWS_<PK of ANNUALREVIEW>, where <PK of ANNUALREVIEW> denotes the name of the primary key column of table ANNUALREVIEW. There is a unique key constraint on the foreign key that refers to table ANNUALREVIEW.

2.1.8.5.2 Unidirectional ManyToMany Relationships

The following mapping defaults apply:

Entity A is mapped to a table named A.

Entity B is mapped to a table named B.

There is a join table that is named A_B (owner name first). This join table has two foreign key columns. One foreign key column refers to table A and has the same type as the primary key of table A. The name of this foreign key column is formed as the concatenation of the following: the name of entity A; "_"; the name of the primary key column in table A. The other foreign key column refers to table B and has the same type as the primary key of table B. The name of this foreign key column is formed as the concatenation of the following: the name of the relationship property or field of entity A; "_"; the name of the primary key column in table B.

Example:

```
@Entity
public class Employee {
   private Collection<Patent> patents;

   @ManyToMany
   public Collection<Patent> getPatents() {
     return patents;
   }

   public void setPatents(Collection<Patent> patents) {
     this.patents = patents;
   }
  ...
}


@Entity
public class Patent {
   ...
}
```

In this example:

Entity `Employee` references a collection of Entity `Patent`.

Entity `Patent` does not reference Entity `Employee`.

Entity `Employee` is the owner of the relationship.

The following mapping defaults apply:

Entity `Employee` is mapped to a table named `EMPLOYEE`.

Entity `Patent` is mapped to a table named `PATENT`.

There is a join table that is named `EMPLOYEE_PATENT` (owner name first). This join table has two foreign key columns. One foreign key column refers to table `EMPLOYEE` and has the same type as the primary key of `EMPLOYEE`. This foreign key column is named `EMPLOYEE_`<PK of EMPLOYEE>, where <PK of EMPLOYEE> denotes the name of the primary key column of table `EMPLOYEE`. The other foreign key column refers to table `PATENT` and has the same type as the primary key of `PATENT`. This foreign key column is named `PATENTS_`<PK of PATENT>, where <PK of PATENT> denotes the name of the primary key column of table `PATENT`.

2.1.9 Inheritance

An entity may inherit from another entity class. Entities support inheritance, polymorphic associations, and polymorphic queries.

Both abstract and concrete classes can be entities. Both abstract and concrete classes may be annotated with the `Entity` annotation, mapped as entities, and queried for as entities.

Entities may extend non-entity classes and non-entity classes may extend entity classes.

When an entity is defined as a subclass of another entity, the primary keys of the entities must be of the same type.

These concepts are described further in the following sections.

2.1.9.1 Abstract Entity Classes

An abstract class can be specified as an entity. An abstract entity differs from a concrete entity only in that it cannot be directly instantiated. An abstract entity is mapped as an entity and can be the target of queries (which will operate over and/or retrieve instances of its concrete subclasses).

An abstract entity class is annotated with the `Entity` annotation or denoted in the XML descriptor as an entity.

The following example shows the use of an abstract entity class in the entity inheritance hierarchy.

Example: Abstract class as an Entity

```
@Entity(access=FIELD)
@Table(name="EMP")
@Inheritance(strategy=JOINED)
public abstract class Employee {
     @Id protected Integer empId;
     @Version protected Integer version;
     @ManyToOne protected Address address;
     ...
}


@Entity
@Table(name="FT_EMP")
@Inheritance(discriminatorValue="FT")
@PrimaryKeyJoinColumn(name="FT_EMPID")
public class FullTimeEmployee extends Employee {

    // Inherit empId, but mapped in this class to FT_EMP.FT_EMPID
    // Inherit version mapped to EMP.VERSION
    // Inherit address mapped to EMP.ADDRESS fk

     private Integer salary;
     // Defaults to FT_EMP.SALARY
     public Integer getSalary() { return salary; }
     ...
}


@Entity(access=FIELD)
@Table(name="PT_EMP")
@Inheritance(discriminatorValue="PT")
// PK field is PT_EMP.EMPID due to PrimaryKeyJoinColumn default
public class PartTimeEmployee extends Employee {
     protected Float hourlyWage;
     ...
}
```

2.1.9.2 Non-Entity Classes in the Entity Inheritance Hierarchy

An entity may have a non-entity superclass, which may be either a concrete or abstract class.

The non-entity superclass serves for inheritance of behavior only. The state of a non-entity superclass is not persistent. Any state inherited from non-entity superclasses is non-persistent in an inheriting entity class. This non-persistent state is not managed by the EntityManager, nor it is required to be retained across transactions.

Non-entity classes cannot be passed as arguments to methods of the EntityManager or Query interfaces and cannot bear mapping information.

The following example illustrates the use of a non-entity class as a superclass of an entity.

Example: Non-entity superclass

```
public class Cart {

    // This state is transient
    Integer operationCount;

    public Cart() { operationCount = 0; }
    public Integer getOperationCount() { return operationCount; }
    public void incrementOperationCount() { operationCount++; }
}

@Entity
public class ShoppingCart extends Cart {

    Collection<Item> items = new Vector<Item>();

    public ShoppingCart() { super(); }

    @OneToMany
    public Collection<Item> getItems() { return items; }
    public void addItem(Item item) {
        items.add(item);
        incrementOperationCount();
    }
}
```

2.1.9.3 Embeddable Superclasses

An entity may have an embeddable superclass, which provides persistent entity state and mapping information, but which is not itself an entity. Typically, the purpose of an embeddable superclass is to define state and mapping information that is common to multiple entity classes.

An embeddable superclass, unlike an entity, is not queryable and cannot be passed as an argument to EntityManager or Query operations. An embeddable superclass cannot be the target of a persistent relationship.

Both abstract or concrete classes may be specified as embeddable superclasses. The `EmbeddableSuperclass` annotation (or `embeddable-superclass` XML descriptor element) is used to designate an embeddable superclass.

A class designated as an embeddable superclass has no separate table defined for it. Its mapping information is applied to the entities that inherit from it.

A class designated as `EmbeddableSuperclass` can be mapped in the same way as an entity except that the mappings will apply only to its subclasses since no table exists for the embeddable superclass. When applied to the subclasses the inherited mappings will apply in the context of the subclass tables. Mapping information may be overridden in such subclasses by using the `AttributeOverride` annotation or `attribute-override` XML element.

All other entity mapping defaults apply equally to a class designated as `EmbeddableSuperclass`.

The following example illustrates the definition of a concrete class as an embeddable superclass.

Example: Concrete class as an Embeddable Superclass

```
@EmbeddableSuperclass(access=FIELD)
public class Employee {

    @Id protected Integer empId;
    @Version protected Integer version;
    @ManyToOne @JoinColumn(name="ADDR")
    protected Address address;

    public Integer getEmpId() { ... }
    public void setEmpId(Integer id) { ... }
    public Address getAddress() { ... }
    public void setAddress(Address addr) { ... }
}

// Default table is FTEMPLOYEE table
@Entity
public class FTEmployee extends Employee {

    // Inherited empId field mapped to FTEMPLOYEE.EMPID
    // Inherited version field mapped to FTEMPLOYEE.VERSION
    // Inherited address field mapped to FTEMPLOYEE.ADDR fk
    private Integer salary;

    public FTEmployee() {}

    // Defaults to FTEMPLOYEE.SALARY
    public Integer getSalary() { ... }
    public void setSalary(Integer salary) { ... }
}

@Entity(access=FIELD) @Table(name="PT_EMP")
@AttributeOverride(name="address", column=@Column(name="ADDR_ID"))
public class PartTimeEmployee extends Employee {

    // Inherited empId field mapped to PT_EMP.EMPID
    // Inherited version field mapped to PT_EMP.VERSION
    // address field mapping overridden to PT_EMP.ADDR_ID fk
    @Column(name="WAGE")
    protected Float hourlyWage;

    public PartTimeEmployee() {}

    public Float getHourlyWage() { ... }
    public void setHourlyWage(Float wage) { ... }
}
```

2.1.10 Inheritance Mapping Strategies

The mapping of class hierarchies is specified through metadata.

There are three basic strategies that are used when mapping a class or class hierarchy to a relational database schema:

- a single table per class hierarchy

- a single table per concrete entity class
- a strategy in which fields that are specific to a subclass are mapped to a separate table than the fields that are common to the parent class, and a join is performed to instantiate the subclass.

An implementation is required to support the single table per class hierarchy inheritance mapping strategy.

Support for the other inheritance mapping strategies is optional in this release and will be required in the next release of this specification. Support for the combination of inheritance strategies will be addressed further in a future draft of this specification.

2.1.10.1 Single Table per Class Hierarchy Strategy

In this strategy, all the classes in a hierarchy are mapped to a single table. The table has a column that serves as a “discriminator column”, that is, a column whose value identifies the specific subclass to which the instance that is represented by the row belongs.

This mapping strategy provides good support for polymorphic relationships between entities and for queries that range over the class hierarchy.

It has the drawback, however, that it requires that the columns that correspond to state specific to the subclasses be nullable.

2.1.10.2 Table per Class Strategy

In this mapping strategy, each class is mapped to a separate table. All properties of the class, including inherited properties, are mapped to columns of the table for the class.

This strategy has the following drawbacks:

- It provides poor support for polymorphic relationships.
- It typically requires that SQL UNION queries (or a separate SQL query per subclass) be issued for queries that are intended to range over the class hierarchy.

2.1.10.3 Joined Subclass Strategy

In the joined subclass strategy, the root of the class hierarchy is represented by a single table. Each subclass is represented by a separate table that contains those fields that are specific to the subclass (not inherited from its superclass), as well as the column(s) that represent its primary key. The primary key column(s) of the subclass table serves as a foreign key to the primary key of the superclass table.

This strategy provides support for polymorphic relationships between entities.

It has the drawback that it requires that one or more join operations be performed to instantiate instances of a subclass. In deep class hierarchies, this may lead to unacceptable performance. Queries that range over the class hierarchy likewise require joins.

Chapter 3 Entity Operations

This chapter describes the use of the `EntityManager` API to manage the entity instance lifecycle and the use of the `Query` API to retrieve and query entities and their persistent state.

3.1 EntityManager

An EntityManager instance is associated with a persistence context. A persistence context is a set of entity instances in which for any persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their lifecycle are managed.

The `EntityManager` interface defines the methods that are used to interact with the persistence context. The `EntityManager` API is used to create and remove persistent entity instances, to find entities by their primary key identity, and to query over entities.

The set of entities that can be managed by a given EntityManager instance is defined by a persistence unit. A persistence unit defines the set of all classes that are related or grouped by the application, and which must be colocated in their mapping to a single database.

Section 3.1 defines the `EntityManager` interface. The entity instance lifecycle is described in Section 3.2. The relationship between an EntityManager and a persistence unit is described in Chapter 6.

3.1.1 EntityManager Interface

```
package javax.persistence;

/**
 * Interface used to interact with the persistence context.
*/
public interface EntityManager {

     /**
      * Make an instance managed and persistent.
      * @param entity
      * @throws IllegalArgumentException if not an entity
      *                  or entity is detached
      * @throws TransactionRequiredException if there is
      *                  no transaction
      */
     public void persist(Object entity);

     /**
      * Merge the state of the given entity into the
      * current persistence context.
      * @param entity
      * @return the instance that the state was merged to
      * @throws IllegalArgumentException if instance is not an
      *                  entity or is a removed entity
      * @throws TransactionRequiredException if there is
      *                  no transaction
      */
     public <T> T merge(T entity);

     /**
      * Remove the instance.
      * @param entity
      * @throws IllegalArgumentException if not an entity
      *                  or if a detached entity
      * @throws TransactionRequiredException if there is
      *                  no transaction
      */
     public void remove(Object entity);

     /**
      * Find by primary key.
      * @param entityClass
      * @param primaryKey
      * @return the found entity instance or null
      * if the entity does not exist
      * @throws IllegalArgumentException if the first argument does
      *                      not denote an entity type or the second
      *                      argument is not a valid type for that
      *                      entity’s primary key
      */
     public <T> T find(Class<T> entityClass, Object primaryKey);


     /**
      * Get an instance, whose state may be lazily fetched.
      * If the requested instance does not exist in the database,
      * throws EntityNotFoundException when the instance state is
```

```
     * first accessed. (The container is permitted to throw
     * EntityNotFoundException when get is called.)
     * The application should not expect that the instance state will
     * be available upon detachment, unless it was accessed by the
     * application while the entity manager was open.
     * @param entityClass
     * @param primaryKey
     * @return the found entity instance
     * @throws IllegalArgumentException if the first argument does
     *                   not denote an entity type or the second
     *                   argument is not a valid type for that
     *                   entity's primary key
     * @throws EntityNotFoundException if the entity state
     *                   cannot be accessed
     */
    public <T> T getReference(Class<T> entityClass, Object prima-
ryKey);

    /**
     * Synchronize the persistence context to the
     * underlying database.
     * @throws TransactionRequiredException if there is
     *                no transaction
     * @throws PersistenceException if the flush fails
     */
    public void flush();

    /**
     * Refresh the state of the instance from the database,
     * overwriting changes made to the entity, if any.
     * @param entity
     * @throws IllegalArgumentException if not an entity
     *                or entity is not managed
     * @throws TransactionRequiredException if there is
     *                no transaction
     * @throws EntityNotFoundException if the entity no longer
     *                exists in the database
     */
    public void refresh(Object entity);

    /**
     * Check if the instance belongs to the current persistence
     * context.
     * @param entity
     * @return
     * @throws IllegalArgumentException if not an entity
     */
    public boolean contains(Object entity);

    /**
     * Create an instance of Query for executing an
     * EJB QL statement.
     * @param ejbqlString an EJB QL query string
     * @return the new query instance
     * @throws IllegalArgumentException if query string is not valid
     */
    public Query createQuery(String ejbqlString);

    /**
```

```
     * Create an instance of Query for executing a
     * named query (in EJB QL or native SQL).
     * @param name the name of a query defined in metadata
     * @return the new query instance
     * @throws IllegalArgumentException if query string is not valid
     */
    public Query createNamedQuery(String name);

    /**
     * Create an instance of Query for executing
     * a native SQL statement.
     * @param sqlString a native SQL query string
     * @return the new query instance
     * @throws IllegalArgumentException if query string is not valid
     */
    public Query createNativeQuery(String sqlString);

    /**
     * Create an instance of Query for executing
     * a native SQL query.
     * @param sqlString a native SQL query string
     * @param resultClass the class of the resulting instances
     * @return the new query instance
     * @throws IllegalArgumentException if query string is not valid
     */
    public Query createNativeQuery(String sqlString, Class result-
Class);

    /**
     * Create an instance of Query for executing
     * a native SQL query.
     * @param sqlString a native SQL query string
     * @param resultSetMapping the name of the result set mapping
     * @return the new query instance
     * @throws IllegalArgumentException if query string is not valid
     */
    public Query createNativeQuery(String sqlString, String result-
SetMapping);

    /**
     * Closes an application-managed EntityManager.
     * This method can only be called when the EntityManager
     * is not associated with an active transaction.
     * After an EntityManager has been closed, all methods on the
     * EntityManager instance will throw the IllegalStateException
     * except for isOpen, which will return false.
     * @throws IllegalStateException if the EntityManager is
     *       associated with an active transaction or if the
     *       EntityManager is container-managed.
     */
    public void close();

    /**
     * Indicates whether the EntityManager is open.
     * @return true until the EntityManager has been closed.
     */
    public boolean isOpen();

    /**
```

```
     * Returns the resource-level transaction object.
     * The EntityTransaction instance may be used serially to
     * begin and commit multiple transactions.
     * @return EntityTransaction instance
     * @throws IllegalStateException if invoked on a JTA
     *       EntityManager or an EntityManager that has been closed.
     */
    public EntityTransaction getTransaction();

}
```

The `persist`, `merge`, `remove`, `flush`, and `refresh` methods must be invoked within a transaction context. If there is no transaction context, the `javax.persistence.TransactionRequiredException` is thrown.

If an argument to the `createQuery`, `createNamedQuery`, or `createNativeQuery` method is not a valid query string or result set specification for the method, the IllegalArgumentException may be thrown or the query execution will fail.

Runtime exceptions thrown by the methods of the `EntityManager` interface will cause the current transaction to be rolled back.

The methods `close`, `isOpen`, and `getTransaction` are used to manage application-managed entity managers and their lifecycle. See Section 5.2.2, "Obtaining an Application-managed Entity Manager".

3.1.2 Example of Use of EntityManager API

```
@Stateless public class OrderEntry {

  @PersistenceContext EntityManager em;

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

3.2 Entity Instance's Life Cycle

This section describes the `EntityManager` operations for managing an entity instance's lifecycle. An entity instance may be characterized as being new, managed, detached, or removed.

- A new entity instance has no persistent identity, and is not yet associated with a persistence context.
- A managed entity instance is an instance with a persistent identity that is currently associated with a persistence context.

- A detached entity instance is an instance with a persistent identity that is not (or no longer) associated with a persistence context.

- A removed entity instance is an instance with a persistent identity, associated with a persistence context, that is scheduled for removal from the database.

The following subsections describe the effect of lifecycle operations upon entities. Use of the `cascade` annotation element may be used to propagate the effect of an operation to associated entities. The cascade functionality is most typically used in parent-child relationships.

3.2.1 Persisting an Entity Instance

A new entity instance becomes both managed and persistent by invoking the `persist` method on it or by cascading the persist operation.

The semantics of the persist operation, applied to an entity *X* are as follows:

- If X is a new entity, it becomes managed. The entity X will be entered into the database at or before transaction commit or as a result of the flush operation.

- If X is a preexisting managed entity, it is ignored by the persist operation. However, the persist operation is cascaded to entities referenced by X, if the relationships from X to these other entities is annotated with the `cascade=PERSIST` or `cascade=ALL` annotation element value or specified with the equivalent XML descriptor element.

- If X is a removed entity, it becomes managed.

- If X is a detached object, an IllegalArgumentException will be thrown by the persist operation (or the transaction commit will fail).

- For all entities Y referenced by a relationship from X, if the relationship to Y has been annotated with the `cascade` element value `cascade=PERSIST` or `cascade=ALL`, the persist operation is applied to Y.

3.2.2 Removal

A managed entity instance becomes removed by invoking the `remove` method on it or by cascading the remove operation.

The semantics of the remove operation, applied to an entity X are as follows:

- If X is a new entity, it is ignored by the remove operation. However, the remove operation is cascaded to entities referenced by X, if the relationships from X to these other entities is annotated with the `cascade=REMOVE` or `cascade=ALL` annotation element value.

- If X is a managed entity, the remove operation causes it to become removed. The remove operation is cascaded to entities referenced by X, if the relationships from X to these other entities is annotated with the `cascade=REMOVE` or `cascade=ALL` annotation element value.

- If X is a detached entity, an IllegalArgumentException will be thrown by the remove operation (or the transaction commit will fail).
- If X is a removed entity, it is ignored by the remove operation.
- A removed entity X will be removed from the database at or before transaction commit or as a result of the flush operation. Accessing a removed entity is undefined.

3.2.3 Synchronization to the Database

The state of persistent entities is synchronized to the database at transaction commit. This synchronization involving writing to the database any updates to persistent entities and their relationships as specified above.[7]

Bidirectional relationships between managed entities will be persisted based on references held by the owning side of the relationship. It is the developer's responsibility to keep the in-memory references held on the owning side and those held on the inverse side consistent with each other when they change.

It is particularly important to ensure that changes to the inverse side of a relationship result in appropriate updates on the owning side, so as to ensure the changes are not lost when they are synchronized to the database. Developers may choose whether or not to update references held by the inverse side when the owning side changes, depending on whether the application can handle out-of-date references on the inverse side until the next database refresh occurs."

The persistence provider runtime is permitted to perform synchronization to the database at other times as well, for example, before query execution. The `flush` method can be used to force synchronization. It applies to entities associated with the persistence context. The `FlushMode` annotation can be used to further control synchronization semantics.

The semantics of the flush operation, applied to an entity *X* are as follows:

- If X is a managed entity, it is synchronized to the database.
  - For all entities Y referenced by a relationship from X, if the relationship to Y has been annotated with the `cascade` element value `cascade=PERSIST` or `cascade=ALL`, the persist operation is applied to Y.
  - For any entity Y referenced by a relationship from X, where the relationship to Y has not been annotated with the `cascade` element value `cascade=PERSIST` or `cascade=ALL`:
    - If Y is new or removed, an IllegalStateException will be thrown by the flush operation (and the transaction rolled back) or the transaction commit will fail.
    - If Y is detached, the semantics depend upon the ownership of the relationship. If X owns the relationship, any changes to the relationship are synchro-

[7] It does not involve a refresh of any managed entities unless the `refresh` operation is explicitly invoked on those entities.

nized with the database; otherwise, if Y owns the relationships, the behavior is undefined.

- If X is a removed entity, it is removed from the database. No cascade options are relevant.

3.2.4 Detached Entities

When the persistence context ends, all managed entity instances associated with the context become detached.

The application may safely access the available state of available detached entity instances after the persistence context ends. The available state includes:

- Any persistent field or property not marked `fetch=LAZY`
- Any persistent field or property that was accessed by the application

If the persistent field or property is an association, the state of an associated instance may only be safely accessed if the associated instance is available. The available instances include:

- Any entity instance retrieved using `find`()
- Any entity instances retrieved using a query or explicitly requested in a FETCH JOIN clause.
- Any entity instance for which an instance variable holding non-primary-key persistent state was accessed by the application
- Any entity instance that may be reached from another available instance by navigating associations marked `fetch=EAGER`

Detached entity instances continue to live outside of the persistence context in which they were persisted or retrieved, and their state is no longer guaranteed to be synchronized with the database state.

A detached entity may also result from serializing an entity, or otherwise passing it by value—e.g, to a separate application tier, through a remote interface, etc.—and the same rules apply.

3.2.4.1 Merging Detached Entity State

The `merge` operation allows for the propagation of state from detached entities onto persistent entities managed by the EntityManager.

The semantics of the `merge` operation applied to an entity X are as follows:

- If X is a detached entity, it is copied onto a pre-existing managed entity instance X' of the same identity or a new managed copy of X is created.
- If X is a new entity instance, a new managed entity instance X' is created and the state of X is *copied* into the new managed entity instance X'.

- If X is a removed entity instance, an IllegalArgumentException will be thrown by the merge operation (or the transaction commit will fail).
- If X is a managed entity, it is ignored by the merge operation, however, the merge operation is cascaded to entities referenced by relationships from X if these relationships have been annotated with the `cascade` element value `cascade=MERGE` or `cascade=ALL` annotation.
- For all entities Y referenced by relationships from X having the `cascade` element value `cascade=MERGE` or `cascade=ALL`, Y is merged recursively as Y'. For all such Y referenced by X, X' is set to reference Y'. (Note that if X is managed then X is the same object as X'.)
- If X is an entity merged to X', with a reference to another entity Y, where `cascade=MERGE` or `cascade=ALL` is not specified, then navigation of the same association from X' yields a reference to a managed object Y' with the same persistent identity as Y.

`Version` columns used by the entity should be checked by the persistence runtime implementation during the merge operation or at flush or commit time.

3.2.5 Managed Instances

The `contains()` method can be used to determine whether an entity instance is managed in the current persistence context.

The `contains` method returns true:

- If the entity has been retrieved from the database, and has not been removed or detached.
- If the entity instance is new, and the `persist` method has been called on the entity or the persist operation has been cascaded to it.

The `contains` method returns false:

- If the instance is detached.
- If the `remove` method has been called on the entity, or the remove operation has been cascaded to it.
- If the instance is new, and the `persist` method has not been called on the entity or the persist operation has not been cascaded to it.

Note that the effect of the cascading of persist or remove is immediately visible to the `contains` method, whereas the actual insertion or deletion of the database representation for the entity may be deferred until the end of transaction.

3.2.6 Transaction Rollback

Transaction rollback causes a pre-existing managed instance or removed instance to become detached.

3.3 Persistence Context

A persistence context may either be scoped to a transaction, or have a scope that extends beyond that of a single transaction (extended persistence context). The enum `PersistenceContextType` is used to define the persistence context scope:

```
public enum PersistenceContextType {
  TRANSACTION,
  EXTENDED
}
```

By default, a persistence context's lifecycle corresponds to the scope of a transaction (i.e., it is of type `PersistenceContextType.TRANSACTION`).

The `PersistenceContextType` is that defined when the EntityManager instance is created (whether explicitly, or in conjunction with injection or JNDI lookup). See Section 5.4.

3.3.1 Extended Persistence Context

A persistence context may be maintained across multiple transactions by specifying the persistence context as an extended persistence context.

When an extended persistence context is used, the extended persistence context exists from the time the EntityManager instance is created until it is closed. This persistence context might span multiple transactions and non-transactional invocations of the EntityManager.

An EntityManager with an extended persistence context maintains its references to the entity objects after a transaction has committed. Those objects remain managed by the EntityManager. See Section 5.4.

3.4 Entity Listeners and Callback Methods

A method may be designated as a callback method to receive notification of entity life cycle events. Callback methods are annotated with a callback annotation or denoted in the XML descriptor as such.[8]

A entity listener class may be used instead of callback methods defined directly on the entity class. An entity listener class is denoted using the `EntityListener` annotation on the entity class with which it is associated or denoted in the XML descriptor as such.

Entity listeners are stateless. The lifecycle of an entity listener is unspecified. Listeners are statically configured for an entity class by use of metadata annotations or the XML descriptor.

The entity listener class must have a public no-arg constructor.

[8] An entity class, even when used within the context of an EJB application, must not implement the `javax.ejb.EntityBean` callback interface.

The annotations used for callback methods on the entity class and for callback methods on the callback listener class are the same. The signatures of individual methods, however, differ. The same method may be annotated with more than one callback annotation, thus serving for more than one callback.

Any subset or combination of annotations appropriate to the entity may be specified on the entity class or on the associated listener class. The same callback may not be specified on both the entity class and the listener class or more than once on either class.

The following rules apply to callbacks:

- Callback methods may throw runtime exceptions. A runtime exception thrown by a callback method that executes within a transaction causes that transaction to be rolled back.
- Callback methods must not throw application exceptions.
- Callbacks can invoke JNDI, JDBC, JMS, and enterprise beans, but not the EntityManager.

Callback methods defined on an entity class have the following signature:

```
public void <METHOD>()
```

Callback methods defined on an entity listener class have the following signature:

```
public void <METHOD>(Object)
```

where `Object` may be declared as the actual entity type, which is the argument passed to the callback method at runtime.

Tthe following lifecycle event callbacks are supported. They may be defined on the entity class or entity listener class.

- `PrePersist`
- `PostPersist`
- `PreRemove`
- `PostRemove`
- `PreUpdate`
- `PostUpdate`
- `PostLoad`

3.4.1 Semantics of the Life Cycle Callback Methods for Entities

The `PrePersist` and `PreRemove` callback methods are invoked for a given entity before the respective EntityManager persist and remove operations for that entity are executed, as specified in section 3.2. These callbacks will also be invoked on all entities to which these operations are cascaded. The `PrePersist` and `PreRemove` methods will always be invoked as part of the synchronous persist and remove operations. Exceptions thrown by any of these callbacks cause the current transaction to be rolled back.

The `PostPersist` and `PostRemove` callback methods are invoked for an entity after the respective EntityManager persist and remove operations for that entity are executed. These callbacks will also be invoked on all entities to which these operations are cascaded. The `PostPersist` and `PostRemove` methods will be invoked after the database insert and delete operations respectively. This may be directly after the persist or remove operations have been invoked or it may be directly after a flush operation has occurred or it may be at the end of the transaction. Exceptions thrown by any of these callbacks cause the current transaction to be rolled back.

The `PreUpdate` and `PostUpdate` callbacks occur before and after the database update operations to entity data respectively. This may be at the time the entity state is updated or it may be at the time state is flushed to the database or at the end of the transaction.

> *Note that it is implementation-dependent as to whether* `PreUpdate` *and* `PostUpdate` *callbacks occur when an entity is persisted and subsequently modified in a single transaction or when an entity is modified and subsequently removed within a single transaction. Portable applications should not rely on such behavior.*

The `PostLoad` method for an entity is invoked after the entity has been loaded into the current persistence context from the database or after the refresh operation has been applied to it. The `PostLoad` method is invoked before a query result is returned or accessed or before an association is traversed.

If a superclass of an entity class on which a callback is defined (either directly or on a callback listener class for the entity class) specifies the same callback annotation, the callback defined by the superclass is not invoked. If the entity class does not define such a callback, the callback defined by the most specific superclass is invoked.

The entity callback methods are invoked in the transaction and security contexts of the calling application.

The JNDI context for the entity callback methods is defined by that of the calling component, if any.

Portable applications must not invoke `EntityManager` or `Query` operations or access other entity instances in a callback method.[9]

[9] The semantics of such operations may be standardized in a future release of this specification.

3.4.2 Example

```
@Entity
@EntityListener(com.acme.AlertMonitor.class)
public class AccountBean implements Account {

     Long accountId;
     Integer balance;
     boolean preferred;

    public Long getAccountId() { ... }
    public Integer getBalance() { ... }
    @Transient // because status depends upon non-persistent context
    public boolean isPreferred() { ... }

    public void deposit(Integer amount) { ... }
    public Integer withdraw(Integer amount) throws NSFException {... }

    @PrePersist
    public void validateCreate() {
        if (getBalance() < MIN_REQUIRED_BALANCE)
        throw new AccountException("Insufficient balance to open an
account");
    }

    @PostLoad
    public void adjustPreferredStatus() {
        preferred =
            (getBalance() >= AccountManager.getPreferredStatu-
sLevel());
    }
}

public class AlertMonitor {

    @PostPersist
    public void newAccountAlert(Account acct) {
        Alerts.sendMarketingInfo(acct.getAccountId(), acct.getBal-
ance());
```

3.5 Query API

The `Query` API is used for both static queries (i.e., named queries) and dynamic queries. The Query API also supports named parameter binding and pagination control.

3.5.1 Query Interface

```
package javax.persistence;

import java.math.BigDecimal;
import java.util.Calendar;
import java.util.Date;
import java.util.Iterator;
import java.util.List;


/**
 * Interface used to control query execution.
 */
public interface Query {

    /**
     * Execute a SELECT query and return the query results
     * as a List.
     * @return a list of the results
     * @throws IllegalStateException if called for an EJB QL
     *             UPDATE or DELETE statement
     */
    public List getResultList();

    /**
     * Execute a SELECT query that returns a single result.
     * @return the result
     * @throws EntityNotFoundException if there is no result
     * @throws NonUniqueResultException if more than one result
     * @throws IllegalStateException if called for an EJB QL
     *             UPDATE or DELETE statement
     */
    public Object getSingleResult();

    /**
     * Execute an update or delete statement.
     * @return the number of entities updated or deleted
     * @throws IllegalStateException if called for an EJB QL
     *             SELECT statement
     * @throws TransactionRequiredException if there is
     *                   no transaction
     */
    public int executeUpdate();

    /**
     * Set the maximum number of results to retrieve.
     * @param maxResult
     * @return the same query instance
     * @throws IllegalArgumentException if argument is negative
     */
    public Query setMaxResults(int maxResult);

    /**
     * Set the position of the first result to retrieve.
     * @param start position of the first result, numbered from 0
     * @return the same query instance
     * @throws IllegalArgumentException if argument is negative
     */
    public Query setFirstResult(int startPosition);
```

```
    /**
     * Set an implementation-specific hint.
     * If the hint name is not recognized, it is silently ignored.
     * @param hintName
     * @param value
     * @return the same query instance
     * @throws IllegalArgumentException if the second argument is not
     *                                   valid for the implementation
     */
    public Query setHint(String hintName, Object value);

    /**
     * Bind an argument to a named parameter.
     * @param name the parameter name
     * @param value
     * @return the same query instance
     * @throws IllegalArgumentException if parameter name does not
     *                    correspond to parameter in query string
     *                    or argument is of incorrect type
     */
    public Query setParameter(String name, Object value);

    /**
     * Bind an instance of java.util.Date to a named parameter.
     * @param name
     * @param value
     * @param temporalType
     * @return the same query instance
     * @throws IllegalArgumentException if parameter name does not
     *                    correspond to parameter in query string
     */
    public Query setParameter(String name, Date value, TemporalType
temporalType);

    /**
     * Bind an instance of java.util.Calendar to a named parameter.
     * @param name
     * @param value
     * @param temporalType
     * @return the same query instance
     * @throws IllegalArgumentException if parameter name does not
     *                    correspond to parameter in query string
     */
    public Query setParameter(String name, Calendar value, Temporal-
Type temporalType);

    /**
     * Bind an argument to a positional parameter.
     * @param position
     * @param value
     * @return the same query instance
     * @throws IllegalArgumentException if position does not
     *               correspond to positional parameter of query
     *               or argument is of incorrect type
     */
    public Query setParameter(int position, Object value);
```

```
     /**
      * Bind an instance of java.util.Date to a positional parameter.
      * @param position
      * @param value
      * @param temporalType
      * @return the same query instance
      * @throws IllegalArgumentException if position does not
      *                 correspond to positional parameter of query
      */
     public Query setParameter(int position, Date value, TemporalType
temporalType);

     /**
      * Bind an instance of java.util.Calendar to a positional param-
eter.
      * @param position
      * @param value
      * @param temporalType
      * @return the same query instance
      * @throws IllegalArgumentException if position does not
      *                 correspond to positional parameter of query
      */
     public Query setParameter(int position, Calendar value, Temporal-
Type temporalType);

     /**
      * Set the flush mode type to be used for the query execution.
      * @param flushMode
      */
     public Query setFlushMode(FlushModeType flushMode);
}
```

The elements of a query result whose SELECT clause consists of more than one value are of type `Object[]`.

An `IllegalArgumentException` is thrown if a parameter name is specified that does not correspond to a named parameter in the query string, if a positional value is specified that does not correspond to a positional parameter in the query string, or if the type of the parameter is not valid for the query. This exception may be thrown when the parameter is bound, or the execution of the query may fail.

Runtime exceptions thrown by the methods of the `Query` interface will cause the current transaction to be rolled back.

3.5.1.1 Example

```
public List findWithName(String name) {
  return em.createQuery(
     "SELECT c FROM Customer c WHERE c.name LIKE :custName")
    .setParameter("custName", name)
    .setMaxResults(10)
    .getResultList();
}
```

3.5.2 Parameter Names

A named parameter is an identifier that is prefixed by the ":" symbol. It follows the rules for identifiers defined in Section 4.4.1. The use of named parameters applies to EJB QL, and is not defined for native queries. Only positional parameter binding may be portably used for native queries.

3.5.3 Named Queries

Named queries are static queries expressed in metadata. Named queries can be defined in EJB QL or in SQL.

The following is an example of the definition of an EJB QL named query:

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

The following is an example of the use of a named query:

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

3.5.4 Polymorphic Queries

By default, all queries are polymorphic. That is, the FROM clause of a query designates not only instances of the specific entity class(es) to which it explicitly refers, but subclasses as well. The instances returned by a query include instances of the subclasses that satisfy the query conditions.

For example, the query

```
select avg(e.salary) from Employee e where e.salary > 80000
```

returns the average salary of all employees, including subtypes of `Employee`, such as `Manager` and `Exempt`.

3.5.5 SQL Queries

Queries may be expressed in native SQL. The result of a native SQL query may consist of entities, scalar values, or a combination of the two. The entities returned by a query may be of different entity types.

> *The SQL query facility is intended to provide support for those cases where it is necessary to use the native SQL of the target database in use (and/or where EJB QL cannot be used). Native SQL queries are not expected to be portable across databases.*

When multiple entities are returned by a SQL query, the entities must be specified and mapped to the column results of the SQL statement in a `SqlResultSetMapping` metadata definition. This result set mapping metadata can then be used by the persistence provider runtime to map the JDBC results into the expected objects. See Section 8.3.4 for the definition of the `SqlResultSetMapping` metadata annotation and related annotations.

If the results of the query are limited to entities of a single entity class, a simpler form may be used and `SqlResultSetMapping` metadata is not required.

This is illustrated in the following example in which a native SQL query is created dynamically using the `createNativeQuery` method and the entity class that specifies the type of the result is passed in as an argument.

```
Query q = em.createNativeQuery(
         "SELECT o.id, o.quantity, o.item " +
         "FROM Order o, Item i " +
         "WHERE (o.item = i.id) AND (i.name = ‘widget’)",
       com.acme.Order.class);
```

When executed, this query will return a Collection of all Order entities for items named "widget". The same results could also be obtained using `SqlResultSetMapping`:

```
Query q = em.createNativeQuery(
              "SELECT o.id, o.quantity, o.item " +
              "FROM Order o, Item i " +
              "WHERE (o.item = i.id) AND (i.name = ‘widget’)",
              "WidgetOrderResults");
```

In this case, the metadata for the query result type might be specified as follows:

```
@SqlResultSetMapping(name="WidgetOrderResults",
       entities=@EntityResult(entityClass=com.acme.Order.class))
```

The following query and `SqlResultSetMapping` metadata illustrates the return of multiple entity types and assumes default metadata and column name defaults.

```
Query q = em.createNativeQuery(
  "SELECT o.id, o.quantity, o.item, i.id, i.name, i.description "+
  "FROM Order o, Item i " +
  "WHERE (o.quantity > 25) AND (o.item = i.id)",
  "OrderItemResults");


@SqlResultSetMapping(name="OrderItemResults",
    entities={
       @EntityResult(entityClass=com.acme.Order.class),
       @EntityResult(entityClass=com.acme.Item.class)
    })
```

When an entity is being returned, the SQL statement should select all of the columns that are mapped to the entity object. This should include foreign key columns to related entities. The results obtained when insufficient data is available are undefined.

An example of combining multiple entity types and that includes aliases in the SQL statement requires that the column names be explicitly mapped to the entity fields. The `FieldResult` annotation is used for this purpose.

```
Query q = em.createNativeQuery(
      "SELECT o.id AS order_id, " +
             "o.quantity AS order_quantity, " +
             "o.item AS order_item, " +
             "i.id, i.name, i.description " +
         "FROM Order o, Item i " +
         "WHERE (order_quantity > 25) AND (order_item = i.id)",
      "OrderItemResults");

   @SqlResultSetMapping(name="OrderItemResults",
       entities={
          @EntityResult(entityClass=com.acme.Order.class, fields={
             @FieldResult(name="id", column="order_id"),
             @FieldResult(name="quantity", column="order_quantity"),
             @FieldResult(name="item", column="order_item")}),
          @EntityResult(entityClass=com.acme.Item.class)
   })
```

Scalar result types can be included in the query result by specifying the `ColumnResult` annotation in the metadata.

```
Query q = em.createNativeQuery(
      "SELECT o.id AS order_id, " +
             "o.quantity AS order_quantity, " +
             "o.item AS order_item, " +
             "i.name AS item_name, " +
         "FROM Order o, Item i " +
         "WHERE (order_quantity > 25) AND (order_item = i.id)",
      "OrderResults");

   @SqlResultSetMapping(name="OrderResults",
      entities={
         @EntityResult(entityClass=com.acme.Order.class, fields={
            @FieldResult(name="id", column="order_id"),
            @FieldResult(name="quantity", column="order_quantity"),
            @FieldResult(name="item", column="order_item")})},
      columns={
         @ColumnResult(name="item_name")}
   )
```

The use of named parameters is not defined for native queries. Only positional parameter binding for SQL queries may be used by portable applications.

Support for joins is currently limited to single-valued relationships.

Chapter 4 Query Language

The Enterprise JavaBeans query language, EJB QL, is used to define queries over entities and their persistent state. EJB QL enables the application developer to specify the semantics of queries in a portable way, independent of the particular database in use in an enterprise environment.

This specification release augments the previous version of EJB QL defined in [5] with additional operations, including bulk update and delete, JOIN operations, GROUP BY, HAVING, projection, and subqueries. It also provides for the use of EJB QL in dynamic queries.

The full range of EJB QL may be used in both static and dynamic queries. Both static and dynamic queries may be parameterized. Named parameters as well as positional parameters are supported. Named parameters, which are new to this specification release, are described in Section 4.6.4.2.

This chapter provides the full definition of the language.

4.1 Overview

EJB QL is a query specification language for dynamic queries and for static queries expressed through metadata. It applies both to the persistent entities defined by this specification, as well as to the earlier EJB 2.1 entity beans with container-managed persistence (and their finder and select methods) as defined in [1]. [10]

EJB QL can be compiled to a target language, such as SQL, of a database or other persistent store. This allows the execution of queries to be shifted to the native language facilities provided by the database, instead of requiring queries to be executed on the runtime representation of the entity state. As a result, query methods can be optimizable as well as portable.

The Enterprise JavaBeans query language uses the abstract persistence schemas of entities, including their relationships, for its data model, and it defines operators and expressions based on this data model. EJB QL uses a SQL-like syntax to select objects or values based on entity abstract schema types and relationships among them. It is possible to parse and validate EJB QL queries before entities are deployed because EJB QL is based on abstract schema types.

The term abstract persistence schema refers to the persistent schema abstraction (persistent entities, their state, and their relationships) over which EJB QL queries operate. EJB QL translates queries over this persistent schema abstraction into queries that are executed over the database schema to which entities are mapped. See Section 4.3.

The developer uses EJB QL to write queries based on the abstract persistence schemas and the relationships defined in the metadata annotations or XML descriptor. The abstract schema types of a set of entities can be used in a query if the entities are defined in the same persistence unit as the query. The path expressions of EJB QL allow for navigation over relationships defined in the persistence unit.

A persistence unit defines the set of all classes that are related or grouped by the application and which must be colocated in their mapping to a single database.

Compatibility Note: For EJB 2.1 and earlier entity beans, the scope of the persistence unit is defined by the ejb-jar file. It is assumed that a single deployment descriptor in an ejb-jar file constitutes a nondecomposable unit for the container responsible for implementing the abstract persistence schemas of the entity beans and the relationships defined in the deployment descriptor and the ejb-jar file. Queries can be written by utilizing navigation over the cmr-fields of related beans supplied in the same ejb-jar file.

EJB QL queries can be used in several different ways:

- as queries for selecting entity objects or values through use of methods of the `Query` API (Section 3.5.1), where the queries are expressed either in metadata or dynamically.
- as queries for selecting entity objects through finder methods defined in the home interface of EJB 2.1 container-managed entity bean components using the EJB 2.1 API.
- as queries for selecting entity objects or other values derived from an entity bean's abstract schema type through select methods defined on the entity bean class of EJB 2.1 container-managed entity bean components using the EJB 2.1 API.

Restrictions upon the use of EJB QL for the finder and select methods of EJB 2.1 container-managed persistence entity beans are described in [1].

[10] We use the term "entity" in this chapter to refer both to entities as defined by this specification document as well as to the entity beans with container-managed persistence defined by [1]. Where it is important to distinguish the latter, we refer to them as "EJB 2.1 entity beans."

4.2 EJB QL Statement Types

An EJB QL statement may be either a select statement, an update statement, or a delete statement.

> *This chapter refers to all such statements as "queries". Where it is important to distinguish among statement types, the specific statement type is referenced.*

In BNF syntax, an EJB QL statement is defined as:

EJB QL :: = select_statement | update_statement | delete_statement

Any EJB QL statement may be constructed dynamically or may be statically defined in a metadata annotation or XML descriptor element.

All EJB QL statement types may have parameters.

4.2.1 Select Statements

An EJB QL select statement is a string which consists of the following clauses:

- a SELECT clause, which determines the type of the objects or values to be selected.
- a FROM clause, which provides declarations that designate the domain to which the expressions specified in the other clauses of the query apply.
- an optional WHERE clause, which may be used to restrict the results that are returned by the query.
- an optional GROUP BY clause, which allows query results to be aggregated in terms of groups.
- an optional HAVING clause, which allows filtering over aggregated groups.
- an optional ORDER BY clause, which may be used to order the results that are returned by the query.

In BNF syntax, an EJB QL select statement is defined as:

select_statement :: = select_clause from_clause [where_clause] [groupby_clause]
[having_clause] [orderby_clause]

A select statement must always have a SELECT and a FROM clause. The square brackets [] indicate that the other clauses are optional.

4.2.2 Update and Delete Statements

Update and delete statements provide bulk operations over sets of entities.

In BNF syntax, these operations are defined as:

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.

Update and delete statements are described further in Section 4.11.

Compatibility Note: Update and delete statements are not supported for EJB 2.1 entity beans with container-managed persistence.

4.3 Abstract Schema Types and Query Domains

EJB QL is a typed language, and every expression in EJB QL has a type. The type of an expression is derived from the structure of the expression, the abstract schema types of the identification variable declarations, the types to which the persistent fields and relationships evaluate, and the types of literals.

The abstract schema type of an entity is derived from the entity class and the metadata information provided by Java language annotations or in the XML descriptor.

Informally, the abstract schema type of an entity can be characterized as follows:

- For every persistent field or get accessor method (for a persistent property) of the entity class, *there is a field ("state-field") whose abstract schema type corresponds to that of the field or the result type of the accessor method.*[11]

- *For every persistent relationship field or get accessor method (for a persistent relationship property) of the entity class, there is a field ("association-field") whose type is the abstract schema type of the related entity (or, if the relationship is a one-to-many or many-to-many, a collection of such).*[12]

Abstract schema types are specific to the EJB QL data model. The persistence provider is not required to implement or otherwise materialize an abstract schema type.

The domain of an EJB QL query consists of the abstract schema types of all entities that are defined in the same persistence unit.

[11] For EJB 2.1 entity beans with container-managed persistence, these correspond to the cmp-field elements of the deployment descriptor.

[12] For EJB 2.1 entity beans with container-managed persistence, these correspond to the cmr-field elements of the deployment descriptor.

The domain of a query may be restricted by the *navigability* of the relationships of the entity on which it is based. The association-fields of an entity's abstract schema type determine navigability. Using the association-fields and their values, a query can select related entities and use their abstract schema types in the query.

4.3.1 Naming

Entities are designated in EJB QL query strings by their abstract schema names. The developer assigns unique abstract schema names to entities as part of the development process so that they can be used within queries. These unique names are scoped within the persistence unit.

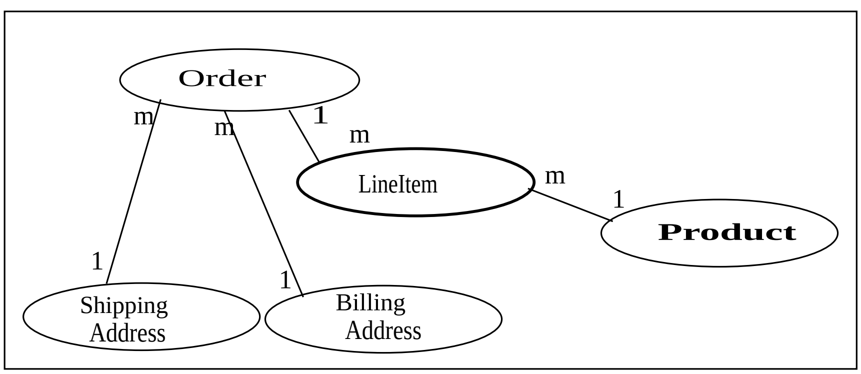
The abstract schema name is defined by the `name` element of the `Entity` annotation (or the `entity-name` XML descriptor element), and defaults to the unqualified name of the entity class.

> *Compatibility Note: For EJB 2.1 entities, abstract schema names are specified by the* `abstract-schema-name` *elements in the deployment descriptor, and there is a one-to-one mapping between entity bean abstract schema types and entity bean homes.*

4.3.2 Example

This example assumes that the application developer provides several entity classes, representing orders, products, line items, shipping addresses, and billing addresses. The abstract schema types for these entities are `Order`, `Product`, `LineItem`, `ShippingAddress`, and `BillingAddress` respectively. These entities are logically in the same persistence unit, as shown in Figure 1.

Figure 1 Several Entities with Abstract Persistence Schemas Defined in the Same Persistence Unit.



The entities `ShippingAddress` and `BillingAddress` each have one-to-many relationships with `Order`. There is also a one-to-many relationship between `Order` and `Lineitem`. The entity `LineItem` is related to `Product` in a many-to-one relationship.

Queries to select orders can be defined by navigating over the association-fields and state-fields defined by `Order` and `LineItem`. A query to find all orders with pending line items might be written as follows:

```
SELECT DISTINCT o
FROM Order AS o JOIN o.lineItems AS l
WHERE l.shipped = FALSE
```

This query navigates over the association-field `lineItems` of the abstract schema type `Order` to find line items, and uses the state-field `shipped` of `LineItem` to select those orders that have at least one line item that has not yet shipped. (Note that this query does not select orders that have no line items.)

Although predefined reserved identifiers, such as DISTINCT, FROM, AS, JOIN, WHERE, and FALSE appear in upper case in this example, predefined reserved identifiers are case insensitive.

The SELECT clause of this example designates the return type of this query to be of type `Order`.

Because the same persistence unit defines the abstract persistence schemas of the related entities, the developer can also specify a query over orders that utilizes the abstract schema type for products, and hence the state-fields and association-fields of both the abstract schema types `Order` and `Product`. For example, if the abstract schema type `Product` has a state-field named `productType`, a query over orders can be specified using this state-field. Such a query might be to find all orders for products with product type office supplies. An EJB QL query string for this might be as follows.

```
SELECT DISTINCT o
FROM Order o JOIN o.lineItems l JOIN l.product p
WHERE p.productType = ‘office_supplies’
```

Because `Order` is related to `Product` by means of the relationships between `Order` and `LineItem` and between `LineItem` and `Product`, navigation using the association-fields `lineItems` and `product` is used to express the query. This query is specified by using the abstract schema name `Order`, which designates the abstract schema type over which the query ranges. The basis for the navigation is provided by the association-fields `lineItems` and `product` of the abstract schema types `Order` and `LineItem` respectively.

4.4 The FROM Clause and Navigational Declarations

The FROM clause of an EJB QL query defines the domain of the query by declaring identification variables. An identification variable is an identifier declared in the FROM clause of a query. The domain of the query may be constrained by path expressions.

Identification variables designate instances of a particular entity abstract schema type. The FROM clause can contain multiple identification variable declarations separated by a comma (`,`).

from_clause ::=
    FROM *identification_variable_declaration*
        *{, {identification_variable_declaration | collection_member_declaration}}**
*identification_variable_declaration ::= range_variable_declaration { join | fetch_join }**

*range_variable_declaration ::= abstract_schema_name [***AS***] identification_variable*
*join ::= join_spec association_path_expression [***AS***] identification_variable*
fetch_join ::= join_spec **FETCH** *association_path_expression*
association_path_expression ::=
collection_valued_path_expression | single_valued_association_path_expression
join_spec::= [**LEFT** *[***OUTER***] |* **INNER** *]* **JOIN**
collection_member_declaration ::=
IN (*collection_valued_path_expression***)** *[***AS***] identification_variable*

The following subsections discuss the constructs used in the FROM clause.

4.4.1 Identifiers

An identifier is a character sequence of unlimited length. The character sequence must begin with a Java identifier start character, and all other characters must be Java identifier part characters. An identifier start character is any character for which the method `Character.isJavaIdentifierStart` returns true. This includes the underscore (_) character and the dollar sign ($) character. An identifier part character is any character for which the method `Character.isJavaIdentifierPart` returns true. The question mark (`?`) character is reserved for use by EJB QL.

The following are the reserved identifiers in EJB QL: *SELECT*, *FROM*, *WHERE*, *UPDATE*, *DELETE*, *JOIN*, *OUTER*, *INNER*, *LEFT*, *GROUP*, *BY*, *HAVING*, *FETCH*, *DISTINCT*, *OBJECT*, *NULL*, *TRUE*, *FALSE*, *NOT*, *AND*, *OR*, *BETWEEN*, *LIKE*, *IN*, *AS*, *UNKNOWN*[13], *EMPTY*, *MEMBER*, *OF*, *IS*, *AVG*, *MAX*, *MIN*, *SUM*, *COUNT*, *ORDER*, *BY*, *ASC*, *DESC*, *MOD*, *UPPER*, *LOWER*, *TRIM*, *POSITION*, *CHARACTER_LENGTH*, *CHAR_LENGTH*, *BIT_LENGTH*, *CURRENT_TIME*, *CURRENT_DATE*, *CURRENT_TIMESTAMP*, *NEW*, *EXISTS*, *ALL*, *ANY*, *SOME*.

Reserved identifiers are case insensitive. Reserved identifiers must not be used as identification variables.

It is recommended that other SQL reserved words also not be as identification variables in EJB QL queries because they may be used as EJB QL reserved identifiers in future releases of this specification.

4.4.2 Identification Variables

An identification variable is a valid identifier declared in the FROM clause of an EJB QL query.

All identification variables must be declared in the FROM clause. Identification variables cannot be declared in other clauses.

An identification variable must not be a reserved identifier or have the same name as any of the following in the same persistence unit:

[13] Not currently used in EJB QL; reserved for future use.

- entity name (as defined by the `Entity` annotation or `entity-name` XML descriptor element)
- abstract-schema-name (as defined by the `abstract-schema-name` deployment descriptor element for EJB 2.1 entity beans)
- ejb-name (as defined by the `ejb-name` deployment descriptor element for EJB 2.1 entity beans)

Identification variables are case insensitive.

An identification variable evaluates to a value of the type of the expression used in declaring the variable. For example, consider the previous query:

```
SELECT DISTINCT o
FROM Order o JOIN o.lineItems l JOIN l.product p
WHERE p.productType = 'office_supplies'
```

In the FROM clause declaration `o.lineItems l`, the identification variable `l` evaluates to any `LineItem` value directly reachable from `Order`. The association-field `lineItems` is a collection of instances of the abstract schema type `LineItem` and the identification variable `l` refers to an element of this collection. The type of `l` is the abstract schema type of `LineItem`.

An identification variable ranges over the abstract schema type of an entity. An identification variable designates an instance of an entity abstract schema type or an element of a collection of entity abstract schema type instances. Identification variables are existentially quantified in an EJB QL query.

An identification variable always designates a reference to a single value. It is declared in one of three ways: in a range variable declaration, in a join clause, or in a collection member declaration. The identification variable declarations are evaluated from left to right in the FROM clause, and an identification variable declaration can use the result of a preceding identification variable declaration of the query string.

4.4.3 Range Variable Declarations

The EJB QL syntax for declaring an identification variable as a range variable is similar to that of SQL; optionally, it uses the AS keyword.

*range_variable_declaration ::= abstract_schema_name [***AS***] identification_variable*

Range variable declarations allow the developer to designate a "root" for objects which may not be reachable by navigation.

In order to select values by comparing more than one instance of an entity abstract schema type, more than one identification variable ranging over the abstract schema type is needed in the FROM clause.

The following query returns orders whose quantity is greater than the order quantity for John Smith. This example illustrates the use of two different identification variables in the FROM clause, both of the abstract schema type `Order`. The SELECT clause of this query determines that it is the orders with quantities larger than John Smith's that are returned.

```
SELECT DISTINCT o1
FROM Order o1, Order o2
WHERE o1.quantity > o2.quantity AND
         o2.customer.lastname = 'Smith' AND
         o2.customer.firstname= 'John'
```

4.4.4 Path Expressions

An identification variable followed by the navigation operator (`.`) and a state-field or association-field is a path expression. The type of the path expression is the type computed as the result of navigation; that is, the type of the state-field or association-field to which the expression navigates.

Depending on navigability, a path expression that leads to a association-field may be further composed. Path expressions can be composed from other path expressions if the original path expression evaluates to a single-valued type (not a collection) corresponding to a association-field. Note that a state field may correspond to an embedded class. A path expression that ends in a *simple* state-field, rather than an embedded class, is terminal and cannot be further composed.

Path expression navigability is composed using "inner join" semantics. That is, if the value of a non-terminal association-field in the path expression is null, the path is considered to have no value, and does not participate in the determination of the result.

The syntax for single-valued path expressions and collection valued path expressions is as follows:

single_valued_path_expression ::=
    state_field_path_expression | single_valued_association_path_expression
state_field_path_expression ::=
    {identification_variable | single_valued_association_path_expression}.state_field
single_valued_association_path_expression ::=
*identification_variable.{single_valued_association_field.}*single_valued_association_field*
collection_valued_path_expression ::=
*identification_variable.{single_valued_association_field.}*collection_valued_association_field*
*state_field ::= {embedded_class_state_field.}*simple_state_field*

A *single_valued_association_field* is designated by the name of an association-field in a one-to-one or many-to-one relationship. The type of a *single_valued_association_field* and thus a *single_valued_association_path_expression* is the abstract schema type of the related entity.

A *collection_valued_association_field* is designated by the name of an association-field in a one-to-many or a many-to-many relationship. The type of a *collection_valued_association_field* is a collection of values of the abstract schema type of the related entity.

Navigation to a related entity results in a value of the related entity's abstract schema type.

The evaluation of a path expression terminating in a state-field results in the abstract schema type corresponding to the Java type designated by the state-field.

It is syntactically illegal to compose a path expression from a path expression that evaluates to a collection. For example, if `o` designates `Order`, the path expression `o.lineItems.product` is illegal since navigation to `lineItems` results in a collection. This case should produce an error when the EJB QL query string is verified. To handle such a navigation, an identification variable must be declared in the FROM clause to range over the elements of the `lineItems` collection. Another path expression must be used to navigate over each such element in the WHERE clause of the query, as in the following:

```
SELECT DISTINCT l.product
FROM Order AS o, IN(o.lineItems) l
```

4.4.5 Joins

An inner join may be implicitly specified by the use of a cartesian product in the FROM clause and a join condition in the WHERE clause. In the absence of a join condition, this reduces to the cartesian product.

The main use case for this generalized style of join is when a join condition does not involve a foreign key relationship that is mapped to an entity relationship.

Example:

```
select c from Customer c, Employee e where c.hatsize = e.shoesize
```

In general, use of this style of inner join (also referred to as theta-join) is less typical than explicitly defined joins over entity relationships.

The syntax for explicit join operations is as follows:

*join ::= join_spec association_path_expression [***AS***] identification_variable*
fetch_join ::= join_spec **FETCH** *association_path_expression*
association_path_expression ::=
    collection_valued_path_expression | single_valued_association_path_expression
join_spec::= [**LEFT** *[***OUTER***] |* **INNER** *]* **JOIN**

The following inner and outer join operation types are supported.

4.4.5.1 Inner Joins (Relationship Joins)

A join over an entity relationship is a typical use case for EJB QL. The IN operator in the FROM clause, described in Section 4.4.6, was introduced by EJB 2.0 for this purpose. This release adds explicit use of the JOIN operator to provide a more natural SQL-like syntax and to allow a wider range of operations.

The syntax for the inner join operation is

[**INNER** *]* **JOIN** *association_path_expression [***AS***] identification_variable*

For example, the query below joins over the relationship between customers and orders. This type of join typically equates to a join over a foreign key relationship in the database.

```
SELECT c FROM Customer c JOIN c.orders o WHERE c.status = 1
```

The keyword INNER may optionally be used:

```
SELECT c FROM Customer c INNER JOIN c.orders o WHERE c.status = 1
```

This is equivalent to the following query using the earlier IN construct, defined in [5]. It selects those customers of status 1 for which at least one order exists:

```
SELECT OBJECT(c) FROM Customer c, IN(c.orders) o WHERE c.status = 1
```

4.4.5.2 Left Outer Joins

LEFT JOIN and LEFT OUTER JOIN are synonymous. They enable the retrieval of a set of entities where matching values in the join condition may be absent.

The syntax for a left outer join is

LEFT *[***OUTER***]* **JOIN** *association_path_expression [***AS***] identification_variable*

For example:

```
SELECT c FROM Customer c LEFT JOIN c.orders o WHERE c.status = 1
```

The keyword OUTER may optionally be used:

```
SELECT c FROM Customer c LEFT OUTER JOIN c.orders o WHERE c.status = 1
```

4.4.5.3 Fetch Joins

An important use case for LEFT JOIN is in enabling the prefetching of related data items as a side effect of a query. This is accomplished by specifying the LEFT JOIN as a FETCH JOIN.

A FETCH JOIN enables the fetching of an association as a side effect of the execution of a query. A FETCH JOIN is specified over an entity and its related entities.

The syntax for a fetch join is

fetch_join ::= [**LEFT** *[***OUTER***]* **|** **INNER** *]* **JOIN FETCH** *association_path_expression*

The association referenced by the right side of the FETCH JOIN clause must be an association that belongs to an entity that is returned as a result of the query. It is not permitted to specify an identification variable for the entities referenced by the right side of the FETCH JOIN clause, and hence references to the implicitly fetched entities cannot appear elsewhere in the query.

The following query returns a set of customers. As a side effect, the associated orders for those customers are also retrieved, even though they are not part of the explicit query result. The persistent fields or properties of the orders that are eagerly fetched are fully initialized. The initialization of the relationship properties of the orders that are retrieved is determined by the metadata for the Order entity class.

```
SELECT DISTINCT c
FROM Customer c LEFT JOIN FETCH c.orders
WHERE c.address.state = 'CA'
```

4.4.6 Collection Member Declarations

An identification variable declared by a *collection_member_declaration* ranges over values of a collection obtained by navigation using a path expression. Such a path expression represents a navigation involving the association-fields of an entity abstract schema type. Because a path expression can be based on another path expression, the navigation can use the association-fields of related entities.

An identification variable of a collection member declaration is declared using a special operator, the reserved identifier IN. The argument to the IN operator is a collection-valued path expression. The path expression evaluates to a collection type specified as a result of navigation to a collection-valued association-field of an entity abstract schema type.

The syntax for declaring a collection member identification variable is as follows:

collection_member_declaration ::=
    IN (*collection_valued_path_expression***)** *[***AS***] identification_variable*

For example, the query

```
SELECT DISTINCT o
FROM Order o JOIN o.lineItems l JOIN l.product p
WHERE p.productType = 'office_supplies'
```

may equivalently be expressed as follows, using the IN operator:

```
SELECT DISTINCT o
FROM Order o, IN(o.lineItems) l
WHERE l.product.productType = 'office_supplies'
```

In this example, `lineItems` is the name of an association-field whose value is a collection of instances of the abstract schema type `LineItem`. The identification variable `l` designates a member of this collection, a *single* `LineItem` abstract schema type instance. In this example, `o` is an identification variable of the abstract schema type `Order`.

4.4.7 EJB QL and SQL

EJB QL treats the FROM clause similarly to SQL in that the declared identification variables affect the results of the query even if they are not used in the WHERE clause. Application developers should use caution in defining identification variables because the domain of the query can depend on whether there are any values of the declared type.

For example, the FROM clause below defines a query over all orders that have line items and existing products. If there are no `Product` instances in the database, the domain of the query is empty and no order is selected.

```
SELECT o
FROM Order AS o, IN(o.lineItems) l, Product p
```

4.4.8 Polymorphism

EJB QL queries are automatically polymorphic. The FROM clause of a query designates not only instances of the specific entity class(es) to which explicitly refers but of subclasses as well. The instances returned by a query include instances of the subclasses that satisfy the query criteria.[14]

4.5 WHERE Clause

The WHERE clause of a query consists of a conditional expression used to select objects or values that satisfy the expression. The WHERE clause restricts the result of a select statement or the scope of an update or delete operation.

A WHERE clause is defined as follows:

where_clause ::= **WHERE** *conditional_expression*

The GROUP BY construct enables the aggregation of values according to the properties of an entity class. The HAVING construct enables conditions to be specified that further restrict the query result as restrictions upon the groups.

The syntax of the HAVING clause is as follows:

having_clause ::= **HAVING** *conditional_expression*

The GROUP BY and HAVING constructs are further discussed in Section 4.7.

4.6 Conditional Expressions

The following sections describe the language constructs that can be used in a conditional expression of the WHERE clause or HAVING clause.

> *Note that state-fields that are mapped in serialized form or as lobs may not be portably used in conditional expressions[15].*

[14] Such query polymorphism does not apply to EJB 2.1 entity beans, since they do not support inheritance. We plan to consider constructs to enable restriction of the polymorphism of queries in a future release.

[15] The implementation is not expected to perform such query operations involving such fields in memory rather than in the database.

4.6.1 Literals

A string literal is enclosed in single quotes—for example: ‘literal’. A string literal that includes a single quote is represented by two single quotes—for example: ‘literal’’s’. EJB QL string literals, like Java `String` literals, use unicode character encoding.

An exact numeric literal is a numeric value without a decimal point, such as 57, -957, +62. Exact numeric literals support numbers in the range of Java `long`. Exact numeric literals use the Java integer literal syntax.

An approximate numeric literal is a numeric value in scientific notation, such as 7E3, -57.9E2, or a numeric value with a decimal, such as 7., -95.7, +6.2. Approximate numeric literals support numbers in the range of Java `double`. Approximate literals use the Java floating point literal syntax.

Appropriate suffixes may be used to indicate the specific type of a numeric literal in accordance with the Java Language Specification.

The boolean literals are `TRUE` and `FALSE`.

Although predefined reserved literals appear in upper case, they are case insensitive.

4.6.2 Identification Variables

All identification variables used in the WHERE or HAVING clause of an EJB QL SELECT or DELETE statement must be declared in the FROM clause, as described in Section 4.4.2. The identification variables used in the WHERE clause of an UPDATE statement must be declared in the UPDATE clause.

Identification variables are existentially quantified in the WHERE and HAVING clause. This means that an identification variable represents a member of a collection or an instance of an entity’s abstract schema type. An identification variable never designates a collection in its entirety.

4.6.3 Path Expressions

It is illegal to use a *collection_valued_path_expression* within a WHERE or HAVING clause as part of a conditional expression except in an *empty_collection_comparison_expression or collection_member_expression*, or as an argument to the SIZE operator.

4.6.4 Input Parameters

Either positional or named parameters may be used. Positional and named parameters may not be mixed in a single query.

Input parameters can only be used in the WHERE clause or HAVING clause of a query.

Note that if an input parameter value is null, comparison operations or arithmetic operations involving the input parameter will return an unknown value. See Section 4.12.

4.6.4.1 Positional Parameters

The following rules apply to positional parameters.

- Input parameters are designated by the question mark (`?`) prefix followed by an integer. For example: `?1`.
- Input parameters are numbered starting from 1.
- If the query is associated with a finder or select method, the number of distinct input parameters must not exceed the number of input parameters for the finder or select method. It is not required that the EJB QL query use all of the input parameters for the finder or select method. An input parameter evaluates to the abstract schema type of the corresponding parameter defined in the signature of the finder or select method with which the query is associated. It is the responsibility of the container to map the input parameter to the appropriate abstract schema type value.

4.6.4.2 Named Parameters

A named parameter is an identifier that is prefixed by the ":" symbol. It follows the rules for identifiers defined in Section 4.4.1.

Example:

```
SELECT c
FROM Customer c
WHERE c.status = :stat
```

Section 3.5.1 describes the API for the binding of named query parameters.

Named parameters are not supported for EJB 2.1 finder and select methods.

4.6.5 Conditional Expression Composition

Conditional expressions are composed of other conditional expressions, comparison operations, logical operations, path expressions that evaluate to boolean values, and boolean literals.

Arithmetic expressions can be used in comparison expressions. Arithmetic expressions are composed of other arithmetic expressions, arithmetic operations, path expressions that evaluate to numeric values, and numeric literals.

Arithmetic operations use numeric promotion.

Standard bracketing `()` for ordering expression evaluation is supported.

Conditional expressions are defined as follows:

*conditional_expression ::= conditional_term | conditional_expression **OR** conditional_term*
*conditional_term ::= conditional_factor | conditional_term **AND** conditional_factor*
*conditional_factor ::= [**NOT**] conditional_primary*
conditional_primary ::= simple_cond_expression | (conditional_expression)
simple_cond_expression ::=
    comparison_expression |
    between_expression |
    like_expression |
    in_expression |
    null_comparison_expression |
    empty_collection_comparison_expression |
    collection_member_expression |
    exists_expression

4.6.6 Operators and Operator Precedence

The operators are listed below in order of decreasing precedence.

- Navigation operator (`.`)
- Arithmetic operators:
  +, - unary
  *, / multiplication and division
  +, - addition and subtraction
- Comparison operators : =, >, >=, <, <=, <> (not equal), `[NOT] BETWEEN`, `[NOT] LIKE`, `[NOT] IN`, `IS [NOT] NULL`, `IS [NOT] EMPTY`, `[NOT] MEMBER [OF]`
- Logical operators:
  `NOT`
  `AND`
  `OR`

The following sections describe other operators used in specific expressions.

4.6.7 Between Expressions

The syntax for the use of the comparison operator [NOT] BETWEEN in a conditional expression is as follows:

*arithmetic_expression [**NOT**] **BETWEEN** arithmetic-expression **AND** arithmetic-expression |*
*string_expression [**NOT**] **BETWEEN** string-expression **AND** string-expression |*
*datetime_expression [**NOT**] **BETWEEN** datetime-expression **AND** datetime-expression*

The BETWEEN expression

`x BETWEEN y AND z`

is semantically equivalent to:

`y <= x AND x <= z`

The rules for unknown and NULL values in comparison operations apply. See Section 4.12.

Examples are:

`p.age BETWEEN 15 and 19` is equivalent to `p.age >= 15 AND p.age <= 19`

`p.age NOT BETWEEN 15 and 19` is equivalent to `p.age < 15 OR p.age > 19`

4.6.8 In Expressions

The syntax for the use of the comparison operator [NOT] IN in a conditional expression is as follows:

in_expression ::=
    *state_field_path_expression [***NOT***]* **IN (** *in_item {, in_item}* | subquery***)**
in_item ::= literal | input_parameter

The *state_field_path_expression* must have a string or numeric value.

The literal and/or input_parameter values must be *like* the same abstract schema type of the *state_field_path_expression* in type. (See Section 4.13).

The results of the subquery must be like the same abstract schema type of the *state_field_path_expression* in type. Subqueries are discussed in Section 4.6.15, “Subqueries”.

Examples are:

`o.country IN (’UK’, ’US’, ’France’)` is true for `UK` and false for `Peru`, and is equivalent to the expression `(o.country = ’UK’) OR (o.country = ’US’) OR (o.country = ’France’)`.

`o.country NOT IN (’UK’, ’US’, ’France’)` is false for `UK` and true for `Peru`, and is equivalent to the expression `NOT ((o.country = ’UK’) OR (o.country = ’US’) OR (o.country = ’France’))`.

There must be at least one element in the comma separated list that defines the set of values for the `IN` expression.

If the value of a *state_field_path_expression* in an IN or NOT IN expression is `NULL` or unknown, the value of the expression is unknown.

4.6.9 Like Expressions

The syntax for the use of the comparison operator [NOT] LIKE in a conditional expression is as follows:

*string_expression [**NOT**] **LIKE** pattern_value [**ESCAPE** escape_character]*

The *string_expression* must have a string value. The *pattern_value* is a string literal or a string-valued input parameter in which an underscore (_) stands for any single character, a percent (%) character stands for any sequence of characters (including the empty sequence), and all other characters stand for themselves. The optional *escape_character* is a single-character string literal or a character-valued input parameter (i.e., `char` or `Character`) and is used to escape the special meaning of the underscore and percent characters in *pattern_value*.[16]

Examples are:

- *address.phone LIKE '12%3'* is true for '123' '12993' and false for '1234'
- *asentence.word LIKE 'l_se'* is true for 'lose' and false for 'loose'
- *aword.underscored LIKE '_%' ESCAPE* '\' is true for '_foo' and false for 'bar'
- *address.phone NOT LIKE '12%3'* is false for '123' and '12993' and true for '1234'

If *the value of the string_expression* or *pattern_value* is `NULL` or unknown, the value of the LIKE expression is unknown. If the *escape_character* is specified and is `NULL`, the value of the LIKE expression is unknown.

4.6.10 Null Comparison Expressions

The syntax for the use of the comparison operator IS NULL in a conditional expression is as follows:

{single_valued_path_expression | input_parameter } **IS** *[***NOT***]* **NULL**

A null comparison expression tests whether or not the single-valued path expression or input parameter is a `NULL` value.

4.6.11 Empty Collection Comparison Expressions

The syntax for the use of the comparison operator IS EMPTY in an *empty_collection_comparison_expression* is as follows:

collection_valued_path_expression **IS** *[***NOT***]* **EMPTY**

[16] Refer to [4] for a more precise characterization of these rules.

This expression tests whether or not the collection designated by the collection-valued path expression is empty (i.e, has no elements).

Example:

```
SELECT o
FROM Order o
WHERE o.lineItems IS EMPTY
```

If the value of the collection-valued path expression in an empty collection comparison expression is unknown, the value of the empty comparison expression is unknown.

4.6.12 Collection Member Expressions

The syntax for the use of the comparison operator MEMBER OF[17] in an *collection_member_expression* is as follows:

entity_expression **[NOT] MEMBER [OF]** *collection_valued_path_expression*
entity_expression ::=
single_valued_association_path_expression |
identification_variable |
input_parameter

This expression tests whether the designated value is a member of the collection specified by the collection-valued path expression.

If the collection valued path expression designates an empty collection, the value of the MEMBER OF expression is FALSE and the value of the NOT MEMBER OF expression is TRUE. Otherwise, if the value of the collection-valued path expression or single-valued association-field path expression in the collection member expression is `NULL` or unknown, the value of the collection member expression is unknown.

4.6.13 Exists Expressions

An EXISTS expression is a predicate that is true only if the result of the subquery consists of one or more values and that is false otherwise.

The syntax of an exists expression is

*exists_expression::= [***NOT***]* **EXISTS** ***(****subquery****)***

[17] The use of the reserved word OF is optional in this expression.

Example:

```
SELECT DISTINCT emp
FROM Employee emp
WHERE EXISTS (
  SELECT spouseEmp
  FROM Employee spouseEmp
  WHERE spouseEmp = emp.spouse)
```

The result of this query consists of all employees whose spouses are also employees.

4.6.14 All or Any Expressions

An ALL conditional expression is a predicate that is true if the comparison operation is true for all values in the result of the subquery, and that is otherwise false. If the result of the subquery is empty, the ALL condition is true.

An ANY conditional expression is a predicate that is true if the comparison operation is true for some value in the result of the subquery, and that is otherwise false. The keyword SOME is synonymous with ANY. If the result of the subquery is empty, the ANY or SOME condition is false.

The comparison operators used with ALL or ANY conditional expressions are =, <, <=, >, >=, <>. The result of the subquery must be like that of the other argument to the comparison operator in type. See Section 4.13.

The syntax of an ALL or ANY expression is specified as follows:

all_or_any_expression ::= { **ALL** | **ANY** | **SOME***}* ***(****subquery****)***

Example:

```
SELECT emp
FROM Employee emp
WHERE emp.salary > ALL (
     SELECT m.salary
     FROM Manager m
     WHERE m.department = emp.department)
```

4.6.15 Subqueries

Subqueries may be used in the WHERE or HAVING clause.[18]

The syntax for subqueries is as follows:

subquery ::= simple_select_clause subquery_from_clause [where_clause]
[groupby_clause] [having_clause]
simple_select_clause ::= **SELECT** *[***DISTINCT***] simple_select_expression*

[18] Subqueries are restricted to the WHERE and HAVING clauses in this release. Support for subqueries in the FROM clause will be considered in a later release of this specification.

subquery_from_clause ::=
    FROM *subselect_identification_variable_declaration*
        *{, subselect_identification_variable_declaration}**
subselect_identification_variable_declaration ::=
    identification_variable_declaration |
    *association_path_expression [***AS***] identification_variable |*
    collection_member_declaration
simple_select_expression::=
    single_valued_path_expression |
    aggregate_select_expression |
    identification_variable

Examples:

```
SELECT DISTINCT emp
FROM Employee emp
WHERE EXISTS (
  SELECT spouseEmp
  FROM Employee spouseEmp
  WHERE spouseEmp = emp.spouse)

SELECT c
FROM Customer c
WHERE (SELECT COUNT(o) FROM c.orders o) > 10
```

Note that some contexts in which a subquery can be used require that the subquery be a scalar subquery (i.e., produce a single result). This is illustrated in the following example involving a numeric comparison operation.

```
SELECT goodCustomer
FROM Customer goodCustomer
WHERE goodCustomer.balanceOwed < (
  SELECT avg(c.balanceOwed) FROM Customer c)
```

4.6.16 Functional Expressions

EJB QL includes the following built-in functions, which may be used in the WHERE or HAVING clause of a query.

If the value of any argument to a functional expression is null or unknown, the value of the functional expression is unknown.

4.6.16.1 String Functions

functions_returning_strings ::=
    CONCAT(*string_primary***,** *string_primary***)** *|*
    SUBSTRING(*string_primary***,**
        *simple_arithmetic_expression***,** *simple_arithmetic_expression***) |**
    TRIM(*[[trim_specification] [trim_character]* **FROM***] string_primary***)** *|*
    LOWER(*string_primary***)** *|*
    UPPER(*string_primary***)**

trim_specification ::= **LEADING | TRAILING | BOTH**

functions_returning_numerics::=
    LENGTH(*string_primary***)** *|*
    LOCATE(*string_primary***,** *string_primary*[**,** *simple_arithmetic_expression*]**) |**

The CONCAT function returns a string that is a concatenation of its arguments.

The second and third arguments of the SUBSTRING function denote the starting position and length of the substring to be returned. These arguments are integers. The first position of a string is denoted by 1. The SUBSTRING function returns a string.

The TRIM function trims the specified character from a string. If the character to be trimmed is not specified, it is assumed to be space (or blank). The optional *trim_character* is a single-character string literal or a character-valued input parameter (i.e., `char` or `Character`)[19]. The TRIM function returns the trimmed string.

The LOWER and UPPER functions convert a string to lower and upper case, respectively. They return a string.

The LOCATE function returns the position of a given string within a string, starting the search at a specified position. It returns the first position at which the string was found as an integer. The first argument is the string to be located; the second argument is the string to be searched; the optional third argument is an integer that represents the string position at which the search is started (by default, the beginning of the string to be searched). The first position in a string is denoted by 1. If the string is not found, 0 is returned.[20]

The LENGTH function returns the length of the string in characters as an integer.

4.6.16.2 Arithmetic Functions

functions_returning_numerics::=
    ABS(*simple_arithmetic_expression***)** *|*
    SQRT(*simple_arithmetic_expression***)** *|*
    MOD(*simple_arithmetic_expression, simple_arithmetic_expression***) |**
    SIZE(*collection_valued_path_expression***)**

The ABS function takes a numeric argument and returns a number (integer, float, or double) of the same type as the argument to the function.

The SQRT function takes a numeric argument and returns a double.

The MOD function takes two integer arguments and returns an integer.

[19] Note that not all databases support the use of a trim character other than the space character; use of this argument may result in queries that are not portable.

[20] Note that not all databases support the use of the third argument to LOCATE; use of this argument may result in queries that are not portable.

The SIZE function returns an integer value, the number of elements of the collection. If the collection is empty, the SIZE function evaluates to zero.

Numeric arguments to these functions may correspond to the numeric Java object types as well as the primitive numeric types.

4.7 GROUP BY, HAVING

The GROUP BY construct enables the aggregation of values according to a set of properties. The HAVING construct enables conditions to be specified that further restrict the query result. Such conditions are restrictions upon the groups.

The syntax of the GROUP BY and HAVING clauses is as follows:

groupby_clause ::= **GROUP BY** *groupby_item {, groupby_item}**
groupby_item ::= state_field_path_expression
having_clause ::= **HAVING** *conditional_expression*

If the query contains both a WHERE clause and a GROUP BY clause, the effect is that of first applying the where clause, and then forming the groups and filtering them according to the HAVING clause. The HAVING clause causes those groups to be retained that satisfy the condition of the HAVING clause.

If there is no GROUP BY clause and the HAVING clause is used, the effect is that of treating the result of the query as a single group.

The requirements for the SELECT clause when GROUP BY is used follow those of SQL: namely, any property that appears in the SELECT clause (other than as an argument to an aggregate function) must also appear in the GROUP BY clause. In forming the groups, null values are treated as the same for grouping purposes.

For example:

```
SELECT c.status, avg(c.filledOrderCount), count(c)
FROM Customer c
GROUP BY c.status
HAVING c.status IN (1, 2)
```

4.8 SELECT Clause

The SELECT clause denotes the query result. More than one value may be returned from the SELECT clause of a query.

The SELECT clause may contain one or more of the following elements: a single range variable or identification variable that ranges over an entity abstract schema type, a single-valued path expression, an aggregate select expression, a constructor expression.

In the case of an EJB 2.1 select method, the SELECT clause is restricted to contain one of the above elements. In the case of a finder method, the SELECT clause is restricted to contain either a single range variable or a single-valued path expression that evaluates to the abstract schema type of the entity bean for which the finder method is defined.

The SELECT clause has the following syntax:

select_clause ::= **SELECT** *[***DISTINCT***] select_expression {, select_expression}**
select_expression ::=
    single_valued_path_expression |
    aggregate_select_expression |
    identification_variable |
    OBJECT(*identification_variable***)** *|*
    constructor_expression
constructor_expression ::=
    NEW *constructor_name* **(** *constructor_item {, constructor_item}** **)**
constructor_item ::= single_valued_path_expression | aggregate_select_expression
aggregate_select_expression ::=
    { **AVG** *|* **MAX** *|* **MIN** *|* **SUM** *}* **(***[***DISTINCT***] state_field_path_expression***)** *|*
    COUNT (*[***DISTINCT***] identification_variable | state_field_path_expression |*
        *single_valued_association_path_expression***)**

All standalone identification variables in the SELECT clause may optionally be qualified by the OBJECT operator. The SELECT clause must not use the OBJECT operator to qualify path expressions.

For example:

```
SELECT c.id, c.status
FROM Customer c JOIN c.orders o
WHERE o.count > 100
```

Note that the SELECT clause must be specified to return only single-valued expressions. The query below is therefore not valid:

```
SELECT o.lineItems FROM Order AS o
```

The DISTINCT keyword is used to specify that duplicate values must be eliminated from the query result.

If DISTINCT is not specified, duplicate values are not eliminated unless the query is specified for a finder or select method whose result type is `java.util.Set.` *If a query is specified for a finder or select method whose result type is* `java.util.Set`*, but does not specify DISTINCT, the container must interpret the query as if SELECT DISTINCT had been specified. In general, however, the application developer should specify the DISTINCT keyword when writing queries for methods that return* `java.util.Set`.

4.8.1 Constructor Expressions in the SELECT Clause

A constructor may be used in the SELECT list to return a collection of Java instances. The specified class is not required to be an entity or to be mapped to the database. The constructor name must be fully qualified.

If an entity class name is specified in the SELECT NEW clause, the resulting entity instances are in the new state.

```
SELECT NEW com.acme.example.CustomerDetails(c.id, c.status, o.count)
FROM Customer c JOIN c.orders o
WHERE o.count > 100
```

4.8.2 Null Values in the Query Result

If the result of an EJB QL query corresponds to a association-field or state-field whose value is null, that null value is returned in the result of the query method. The IS NOT NULL construct can be used to eliminate such null values from the result set of the query.

> *In the case of queries that are associated with finder or select methods for EJB 2.1 entity beans, if the finder or select method is a single-object finder or select method, and the result set of the query consists of a single null value, the container must return the null value as the result of the method. If the result set of a query for a single-object finder or select method contains more than one value (whether non-null, null, or a combination), the container must throw the FinderException.*

Note, however, that state-field types defined in terms of Java numeric primitive types cannot produce NULL values in the query result. An EJB QL query that returns such a state-field type as a result type must not return a null value.

4.8.3 Aggregate Functions in the SELECT Clause

The result of an EJB QL query may be the result of an aggregate function applied to a path expression.

The following aggregate functions can be used in the SELECT clause of an EJB QL query: AVG, COUNT, MAX, MIN, SUM.

For all aggregate functions except COUNT, the path expression that is the argument to the aggregate function must terminate in a state-field. The path expression argument to COUNT may terminate in either a state-field or a association-field, or the argument to COUNT may be an identification variable.

Arguments to the functions SUM and AVG must be numeric. Arguments to the functions MAX and MIN must correspond to orderable state-field types (i.e., numeric types, string types, character types, or date types).

The Java type that is contained in the result of a query using an aggregate function is as follows[21]:

[21] The rules for finder and select method result types are defined in Section 4.10.1.

- COUNT returns Long.
- MAX, MIN return the type of the state-field to which they are applied.
- AVG returns Double.
- SUM returns Long when applied to state-fields of integral types (other than BigInteger); Double when applied to state-fields of floating point types; BigInteger when applied to state-fields of type BigInteger; and BigDecimal when applied to state-fields of type BigDecimal.

If SUM, AVG, MAX, or MIN is used, and there are no values to which the aggregate function can be applied, the result of the aggregate function is NULL.

If COUNT is used, and there are no values to which COUNT can be applied, the result of the aggregate function is 0.

The argument to an aggregate function may be preceded by the keyword DISTINCT to specify that duplicate values are to be eliminated before the aggregate function is applied.[22]

Null values are eliminated before the aggregate function is applied, regardless of whether the keyword DISTINCT is specified.

4.8.4 Examples

The following example returns all line items related to some order:

```
SELECT l
FROM Order o JOIN o.lineItems l
```

The following query returns all line items regardless of whether a line item is related to any order or product:

```
SELECT l FROM LineItems AS l
```

The following query returns the average order quantity:

```
SELECT AVG(o.quantity) FROM Order o
```

The following query returns the total cost of the items that John Smith has ordered.

```
SELECT SUM(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = ‘Smith’ AND c.firstname = ‘John’
```

[22] It is legal to specify DISTINCT with MAX or MIN, but it does not affect the result.

The following query returns the total number of orders.

```
SELECT COUNT(o)
FROM Order o
```

The following query counts the number of items in John Smith's order for which prices have been specified.

```
SELECT COUNT(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Smith' AND c.firstname = 'John'
```

Note that this is equivalent to:

```
SELECT COUNT(l)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Smith' AND c.firstname = 'John'
     AND l.price IS NOT NULL
```

4.9 ORDER BY Clause

The ORDER BY clause allows the objects or values that are returned by the query to be ordered.

The syntax of the ORDER BY clause is

orderby_clause ::= **ORDER BY** *orderby_item {, orderby_item}**
orderby_item ::= state_field_path_expression [ASC | DESC]

When the ORDER BY clause is used in an EJB QL query, each element of the SELECT clause of the query must be one of the following:

1. an identification variable x, optionally denoted as OBJECT(x)

2. a *single_valued_association_path_expression*

3. a *state_field_path_expression*

In the first two cases, each *orderby_item* must be an orderable state-field of the entity abstract schema type value returned by the SELECT clause. In the third case, the *orderby_item* must evaluate to the same state-field of the same entity abstract schema type as the *state_field_path_expression* in the SELECT clause.

For example, the first two queries below are legal, but the third and fourth are not.

```
SELECT o
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = ‘CA’
ORDER BY o.quantity, o.totalcost

SELECT o.quantity, a.zipcode
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = ‘CA’
ORDER BY o.quantity, a.zipcode
```

The following two queries are not legal because the *orderby_item* is not reflected in the SELECT clause of the query.

```
SELECT p.product_name
FROM Order o JOIN o.lineItems l JOIN l.product p JOIN o.customer c
WHERE c.lastname = ‘Smith’ AND c.firstname = ‘John’
ORDER BY p.price

SELECT p.product_name
FROM Order o, IN(o.lineItems) l JOIN o.customer c
WHERE c.lastname = ‘Smith’ AND c.firstname = ‘John’
ORDER BY o.quantity
```

If more than one *orderby_item* is specified, the left-to-right sequence of the *orderby_item* elements determines the precedence, whereby the leftmost *orderby_item* has highest precedence.

The keyword ASC specifies that ascending ordering be used; the keyword DESC specifies that descending ordering be used. Ascending ordering is the default.

SQL rules for the ordering of null values apply: that is, all null values must appear before all non-null values in the ordering or all null values must appear after all non-null values in the ordering, but it is not specified which.

The ordering of the query result is preserved in the result of the query method if the ORDER BY clause is used.

4.10 Return Value Types

The type of the query result specified by the SELECT clause of a query is an entity abstract schema type, a state-field type, the result of an aggregate function, the result of a construction operation, or some sequence of these.

4.10.1 Result types for Finder and Select methods of 2.1 Entity Beans

The following rules apply to EJB 2.x finder and select methods:

How the result type of a query is mapped depends on whether the query is defined for a finder method on the remote home interface, for a finder method on the local home interface, or for a select method.

- The result type of a query for a finder method must be the entity bean abstract schema type that corresponds to the entity bean type of the entity bean on whose home interface the finder method is defined. If the query is used for a finder method defined on the remote home interface of the bean, the result of the finder method is the entity bean's remote interface (or a collection of objects implementing the entity bean's remote interface). If the finder method is defined on the local home interface, the result is the entity bean's local interface (or a collection of objects implementing the entity bean's local interface).

- If the result type of a query for a select method is an entity bean abstract schema type, the return values for the query method are instances of the entity bean's local interface or instances of the entity bean's remote interface, depending on whether the value of the `result-type-mapping` deployment descriptor element contained in the `query` element for the select method is `Local` or `Remote`. The default value for `result-type-mapping` is `Local`.

- If the result type of a query used for a select method is an abstract schema type corresponding to a cmp-field type (excluding queries whose SELECT clause uses one of the aggregate functions AVG, COUNT, MAX, MIN, SUM), the result type of the select method is as follows:
  - If the Java type of the cmp-field is an object type and the select method is a single-object select method, the result of the select method is an instance of that object type. If the select method is a multi-object select method, the result is a collection of instances of that type.
  - If the Java type of the cmp-field is a primitive Java type (e.g., int), and the select method is a single-object select method, the result of the select method is that primitive type.
  - If the Java type of the cmp-field is a primitive Java type (e.g., int), and the select method is a multi-object select method, the result of the select method is a collection of values of the corresponding wrappered type (e.g., Integer).

- If the select method query is an aggregate query, the select method must be a single-object select method.
  - The result type of the select method must be a primitive type, a wrappered type, or an object type that is compatible with the standard JDBC conversion mappings for the type of the cmp-field [6].
  - If the aggregate query uses the SUM, AVG, MAX, or MIN operator, and the result type of the select method is an object type and there are no values to which the aggregate function can be applied, the select method returns null.
  - If the aggregate query uses the SUM, AVG, MAX, or MIN operator, and the result type of the select method is a primitive type and there are no values to which the aggregate function can be applied, the container must throw the `ObjectNotFoundException`.
  - If the aggregate query uses the COUNT operator, the result of the select method should be an exact numeric type. If there are no values to which the COUNT method can be applied, the result of the select method is 0.

The result of a finder or select method may contain a null value if a cmp-field or cmr-field in the query result is null.

4.11 Bulk Update and Delete Operations

Bulk update and delete operations apply to entities of a single entity class (together with its subclasses, if any). Only one entity abstract schema type may be specified in the FROM or UPDATE clause.

A delete operation only applies to entities of the specified class and its subclasses. It does not cascade to related entities.

The *new_value* specified for an update operation must be compatible in type with the state-field to which it is assigned.

The syntax of these operations is as follows:

```
update_statement ::= update_clause [where_clause]
update_clause ::= UPDATE abstract_schema_name [[AS] identification_variable]
                        SET update_item {, update_item}*
update_item ::= [identification_variable.]state_field = new_value
new_value ::=
        simple_arithmetic_expression |
        string_primary |
        datetime_primary |
        boolean_primary


delete_statement ::= delete_clause [where_clause]
delete_clause ::= DELETE FROM abstract_schema_name [[AS] identification_variable]
```

The syntax of the WHERE clause is described in Section 4.5.

Caution should be used when executing bulk update or delete operations because they may result in inconsistencies between the database and the entities in the active persistence context. In general, bulk update and delete operations should only be performed within a separate transaction or at the beginning of a transaction (before entities have been accessed whose state might be affected by such operations).

Examples:

```
DELETE
FROM Customer c
WHERE c.status = ‘inactive’

DELETE
FROM Customer c
WHERE c.status = ‘inactive’
  AND c.orders IS EMPTY

UPDATE customer c
SET c.status = ‘outstanding’
WHERE c.balance < 10000
  AND 1000 > (SELECT COUNT(o)
                FROM customer cust JOIN cust.order o)
```

4.12 Null Values

When the target of a reference does not exist in the database, its value is regarded as `NULL`. SQL 92 `NULL` semantics [4] defines the evaluation of conditional expressions containing `NULL` values.

The following is a brief description of these semantics:

- Comparison or arithmetic operations with a NULL value always yield an unknown value.
- Two NULL values are not considered to be equal, the comparison yields an unknown value.
- Comparison or arithmetic operations with an unknown value always yield an unknown value.
- The IS NULL and IS NOT NULL operators convert a `NULL` state-field or single-valued association-field value into the respective TRUE or FALSE value.
- Boolean operators use three valued logic, defined by Table 1, Table 2, and Table 3.

Table 1 Definition of the AND Operator

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

Table 2 Definition of the OR Operator

OR	**T**	**F**	**U**
T	T	T	T
F	T	F	U
U	T	U	U

Table 3 Definition of the NOT Operator

NOT	
T	F
F	T
U	U

Note: EJB QL defines the empty string, ‘’, as a string with 0 length, which is not equal to a NULL value. However, NULL values and empty strings may not always be distinguished when queries are mapped to some databases. Application developers should therefore not rely on the semantics of EJB QL comparisons involving the empty string and NULL value.

4.13 Equality and Comparison Semantics

EJB QL only permits the values of *like* types to be compared. A type is *like* another type if they correspond to the same Java language type, or if one is a primitive Java language type and the other is the wrappered Java class type equivalent (e.g., `int` and `Integer` are like types in this sense). There is one exception to this rule: it is valid to compare numeric values for which the rules of numeric promotion apply. Conditional expressions attempting to compare non-like type values are disallowed except for this numeric case.

> *Note that EJB QL permits the arithmetic operators and comparison operators to be applied to state-fields and input parameters of the wrappered Java class equivalents to the primitive numeric Java types.*

Two entities of the same abstract schema type are equal if and only if they have the same primary key value.

4.14 Restrictions

Although SQL requires support for fixed decimal comparison in arithmetic expressions, EJB QL does not. For this reason EJB QL restricts exact numeric literals to those without a decimal point (and numerics with a decimal point as an alternate representation for approximate numeric values).

`Boolean` comparison is restricted to = and <>.

EJB QL does not support the use of comments.

> *EJB 2.1 entity objects of different types cannot be compared. EJB QL queries that contain such comparisons are invalid.*

4.15 Examples

The following examples illustrate the syntax and semantics of EJB QL. These examples are based on the example presented in Section 4.3.2.

4.15.1 Simple Queries

Find all orders:

```
SELECT o
FROM Order o
```

Find all orders that need to be shipped to California:

```
SELECT o
FROM Order o
WHERE o.shippingAddress.state = ‘CA’

Find all states for which there are orders:

SELECT DISTINCT o.shippingAddress.state
FROM Order o
```

4.15.2 Queries with Relationships

Find all orders that have line items:

```
SELECT DISTINCT o
FROM Order o, IN(o.lineItems) l
```

Note that the result of this query does not include orders with no associated line items. This query can also be written as:

```
SELECT o
FROM Order o
WHERE o.lineItems IS NOT EMPTY
```

Find all orders that have no line items:

```
SELECT o
FROM Order o
WHERE o.lineItems IS EMPTY
```

Find all pending orders:

```
SELECT DISTINCT o
FROM Order o JOIN o.lineItems l
WHERE l.shipped = FALSE
```

Find all orders in which the shipping address differs from the billing address. This example assumes that the application developer uses two distinct entity types to designate shipping and billing addresses, as in Figure 1.

```
SELECT o
FROM Order o
WHERE
NOT (o.shippingAddress.state = o.billingAddress.state AND
      o.shippingAddress.city = o.billingAddress.city AND
      o.shippingAddress.street = o.billingAddress.street)
```

If the application developer uses a single entity in two different relationships for both the shipping address and the billing address, the above expression can be simplified based on the equality rules defined in Section 4.13. The query can then be written as:

```
SELECT o
FROM Order o
WHERE o.shippingAddress <> o.billingAddress
```

The query checks whether the same entity abstract schema type instance (identified by its primary key) is related to an order through two distinct relationships.

Find all orders for a book titled ‘Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform’:

```
SELECT DISTINCT o
FROM Order o JOIN o.lineItems l
WHERE l.product.type = ‘book’ AND
     l.product.name = ‘Applying Enterprise JavaBeans:
     Component-Based Development for the J2EE Platform’
```

4.15.3 Queries Using Input Parameters

The following query finds the orders for a product whose name is designated by an input parameter:

```
SELECT DISTINCT o
FROM Order o, IN(o.lineItems) l
WHERE l.product.name = ?1
```

For this query, the input parameter must be of the type of the state-field name, i.e., a string.

4.16 EJB QL BNF

EJB QL BNF notation summary:

- { ... } grouping
- [...] optional constructs
- **boldface** keywords
- * zero or more
- | alternates

The following is the BNF for EJB QL. This is a superset of EJB QL as defined in [5].

EJB QL ::= select_statement | update_statement | delete_statement
select_statement ::= select_clause from_clause [where_clause] [groupby_clause]
    [having_clause] [orderby_clause]
update_statement ::= update_clause [where_clause]
delete_statement ::= delete_clause [where_clause]
from_clause ::=
    FROM *identification_variable_declaration*
        *{**,** {identification_variable_declaration | collection_member_declaration}}**
*identification_variable_declaration ::= range_variable_declaration { join | fetch_join }**
*range_variable_declaration ::= abstract_schema_name [***AS***] identification_variable*
*join ::= join_spec association_path_expression [***AS***] identification_variable*
fetch_join ::= join_spec **FETCH** *association_path_expression*
association_path_expression ::=
    collection_valued_path_expression | single_valued_association_path_expression
join_spec::= [**LEFT** *[***OUTER***] |* **INNER** *]* **JOIN**
collection_member_declaration ::=
    IN (*collection_valued_path_expression***)** *[***AS***] identification_variable*
single_valued_path_expression ::=
    state_field_path_expression | single_valued_association_path_expression
state_field_path_expression ::=

{identification_variable | single_valued_association_path_expression}.state_field
single_valued_association_path_expression ::=
identification_variable.{single_valued_association_field.} single_valued_association_field*
collection_valued_path_expression ::=
*identification_variable***.***{single_valued_association_field.}*collection_valued_association_field*
*state_field ::= {embedded_class_state_field.}*simple_state_field*
update_clause ::= **UPDATE** *abstract_schema_name [[***AS***] identification_variable]*
SET *update_item {, update_item}**
update_item ::= [identification_variable.]state_field **=** *new_value*
new_value ::=
simple_arithmetic_expression |
string_primary |
datetime_primary |
boolean_primary
delete_clause ::= **DELETE FROM** *abstract_schema_name [[***AS***] identification_variable]*
select_clause ::= **SELECT** *[***DISTINCT***] select_expression {, select_expression}**
select_expression ::=
single_valued_path_expression |
aggregate_select_expression |
identification_variable |
OBJECT(*identification_variable***)** *|*
constructor_expression
constructor_expression ::=
NEW *constructor_name* **(** *constructor_item {, constructor_item}** **)**
constructor_item ::= single_valued_path_expression | aggregate_select_expression
aggregate_select_expression ::=
{ **AVG** *|* **MAX** *|* **MIN** *|* **SUM** *}* **(***[***DISTINCT***] state_field_path_expression***)** *|*
COUNT (*[***DISTINCT***] identification_variable | state_field_path_expression |*
*single_valued_association_path_expression***)**
where_clause ::= **WHERE** *conditional_expression*
groupby_clause ::= **GROUP BY** *groupby_item {, groupby_item}**
groupby_item ::= state_field_path_expression
having_clause ::= **HAVING** *conditional_expression*
orderby_clause ::= **ORDER BY** *orderby_item {***,** *orderby_item}**
orderby_item ::= state_field_path_expression [**ASC** *|* **DESC** *]*
subquery ::= simple_select_clause subquery_from_clause [where_clause]
[groupby_clause] [having_clause]
subquery_from_clause ::=
FROM *subselect_identification_variable_declaration*
*{, subselect_identification_variable_declaration}**
subselect_identification_variable_declaration ::=
identification_variable_declaration |
*association_path_expression [***AS***] identification_variable |*
collection_member_declaration
simple_select_clause ::= **SELECT** *[***DISTINCT***] simple_select_expression*
simple_select_expression::=
single_valued_path_expression |
aggregate_select_expression |
identification_variable
conditional_expression ::= conditional_term | conditional_expression **OR** *conditional_term*

```
conditional_term ::= conditional_factor | conditional_term AND conditional_factor
conditional_factor ::= [ NOT ] conditional_primary
conditional_primary ::= simple_cond_expression | (conditional_expression)
simple_cond_expression ::=
        comparison_expression |
        between_expression |
        like_expression |
        in_expression |
        null_comparison_expression |
        empty_collection_comparison_expression |
        collection_member_expression |
        exists_expression
between_expression ::=
        arithmetic_expression [NOT] BETWEEN
                        arithmetic_expression AND arithmetic_expression |
        string_expression [NOT] BETWEEN string_expression AND string_expression |
        datetime_expression [NOT] BETWEEN
                datetime_expression AND datetime_expression
in_expression ::=
        state_field_path_expression [NOT] IN ( in_item {, in_item}* | subquery)
in_item ::= literal | input_parameter
like_expression ::=
        string_expression [NOT] LIKE pattern_value [ESCAPE escape_character]
null_comparison_expression ::=
        {single_valued_path_expression | input_parameter} IS [NOT] NULL
empty_collection_comparison_expression ::=
        collection_valued_path_expression IS [NOT] EMPTY
collection_member_expression ::= entity_expression
                [NOT] MEMBER [OF] collection_valued_path_expression
exists_expression::= [NOT] EXISTS (subquery)
all_or_any_expression ::= { ALL | ANY | SOME} (subquery)
comparison_expression ::=
        string_expression comparison_operator {string_expression | all_or_any_expression} |
        boolean_expression { =|<>} {boolean_expression | all_or_any_expression} |
        datetime_expression comparison_operator
                {datetime_expression | all_or_any_expression} |
        entity_expression { = | <> } {entity_expression | all_or_any_expression} |
        arithmetic_expression comparison_operator
                {arithmetic_expression | all_or_any_expression}
comparison_operator ::= = | > | >= | < | <= | <>
arithmetic_expression ::= simple_arithmetic_expression | (subquery)
simple_arithmetic_expression ::=
        arithmetic_term | simple_arithmetic_expression { + | - } arithmetic_term
arithmetic_term ::= arithmetic_factor | arithmetic_term { * | / } arithmetic_factor
arithmetic_factor ::= [{ + | - }] arithmetic_primary
arithmetic_primary ::=
        state_field_path_expression |
        numeric_literal |
        (simple_arithmetic_expression) |
        input_parameter |
```

functions_returning_numerics |
string_expression ::= string_primary | **(***subquery***)**
string_primary ::=
state_field_path_expression |
string_literal |
input_parameter |
functions_returning_strings |
datetime_expression ::= datetime_primary | **(***subquery***)**
datetime_primary ::=
state_field_path_expression |
input_parameter |
functions_returning_datetime |
boolean_expression ::= boolean_primary | **(***subquery***)**
boolean_primary ::=
state_field_path_expression |
boolean_literal |
input_parameter |
entity_expression ::=
single_valued_association_path_expression |
identification_variable |
input_parameter
functions_returning_numerics::=
LENGTH(*string_primary***)** *|*
LOCATE(*string_primary***,** *string_primary[***,** *simple_arithmetic_expression]***) |**
ABS(*simple_arithmetic_expression***)** *|*
SQRT(*simple_arithmetic_expression***) |**
MOD(*simple_arithmetic_expression, simple_arithmetic_expression***) |**
SIZE(*collection_valued_path_expression***)**
functions_returning_datetime ::=
CURRENT_DATE*|*
CURRENT_TIME *|*
CURRENT_TIMESTAMP
functions_returning_strings ::=
CONCAT(*string_primary***,** *string_primary***)** *|*
SUBSTRING(*string_primary***,**
*simple_arithmetic_expression***,** *simple_arithmetic_expression***)|**
TRIM(*[[trim_specification] [trim_character]* **FROM***] string_primary***)** *|*
LOWER(*string_primary***)** *|*
UPPER(*string_primary***)**
trim_specification ::= **LEADING | TRAILING | BOTH**

Chapter 5 EntityManager

The lifecycle of an entity manager may be managed by the J2EE container or by the application. The application may manage the lifecycle of an entity manager in both J2EE and J2SE environments.

5.1 Entity Managers

A container-managed entity manager is an entity manager whose lifecycle is managed by the J2EE container.

An application-managed entity manager is an entity manager whose lifecycle is managed by the application.

Both container-managed and application-managed entity managers are required to be supported in J2EE web containers and EJB containers. Within an EJB environment, a container-managed entity manager is typically used.

In J2SE environments, only application-managed entity managers are supported.

5.2 Obtaining an EntityManager

How an entity manager is obtained depends on whether it is container-managed or application-managed.

When multiple persistence archives are present in the application, the application must designate which persistence unit to use.

5.2.1 Obtaining a Container-managed Entity Manager

A container-managed entity manager is obtained by the application through dependency injection or through JNDI lookup, or by calling `EntityManagerFactory.getEntityManager()`. The container manages the creation of the entity manager and handles the closing of the entity manager transparently to the application.

Entity managers can be injected using the `PersistenceContext` annotation. If multiple persistence units exist, the `unitName` element must be specified. The `type` element specifies whether a transaction-scoped or extended persistence context is to be used.

For example,

```
@PersistenceContext(unitName="order")
EntityManager em;

//here only one persistence unit exists
@PersistenceContext(type=PersistenceContextType.EXTENDED)
EntityManager orderEM;
```

The JNDI lookup of an entity manager is illustrated below:

```
@Stateless
@PersistenceContext(name="OrderEM", unitName="Order")
public class MySessionBean implements MyInterface {
    @Resource SessionContext ctx;

    public void doSomething() {
        EntityManager em = (EntityManager)
                ctx.lookup("OrderEM");
        ...
    }
}
```

5.2.2 Obtaining an Application-managed Entity Manager

An application-managed entity manager is obtained by the application from an entity manager factory.

The `EntityManagerFactory` interface is used to create an entity manager and manage its lifecycle.

An entity manager factory provides entity manager instances that are all configured in the same manner (e.g., configured to connect to the same database, use the same initial settings as defined by the implementation, etc.).

More than one entity manager factory instance may be available simultaneously in the JVM.[23]

5.2.2.1 Obtaining an Entity Manager Factory in a J2EE Container

Within a J2EE environment, an entity manager factory may be injected using the `PersistenceUnit` annotation or obtained through JNDI lookup.

For example

```
@PersistenceUnit
EntityManagerFactory emf;
```

5.2.2.2 Obtaining an Entity Manager Factory in a J2SE Environment

Outside a J2EE container environment, the `javax.persistence.Persistence` class is the bootstrap class that provides access to an entity manager factory. The application creates an entity manager factory by calling the `createEntityManagerFactory` method of the `javax.persistence.Persistence` class.

No name needs to be specified in the case where only one persistence unit exists in the application. If a name is not passed, but multiple persistence units exist, a PersistenceException is thrown.

For example,

```
EntityManagerFactory emf =
javax.persistence.Persistence.createEntityManagerFactory("Order");
    EntityManager em = emf.createEntityManager();
```

[23] This may be the case when using multiple databases, since in a typical configuration a single entity manager only communicates with a single database.

5.2.2.3 The EntityManagerFactory Interface

The `EntityManagerFactory` interface is the interface used by the application to obtain entity managers. When the application has finished using the entity manager factory, and/or at application shutdown, the application should close the entity manager factory.

```
public interface javax.persistence.EntityManagerFactory {

     /**
      * Create a new EntityManager of PersistenceContextType.TRANSAC-
TION
      *
      * The isOpen method will return true on the returned instance.
      *
      * This method returns a new EntityManager instance (with a new
      * persistence context) every time it is invoked.
      */
     EntityManager createEntityManager();

     /**
      * Create a new EntityManager of the specified
      * PersistenceContextType.
      * The isOpen method will return true on the returned instance.
      * This method returns a new EntityManager instance (with a new
      * persistence context) every time it is invoked.
      */
     EntityManager createEntityManager(PersistenceContextType type);

     /**
      * Get the container-managed EntityManager bound to the
      * current JTA transaction.
      * If there is no persistence context bound to the current
      * JTA transaction, a new persistence context is created and
      * associated with the transaction.
      * If there is an existing persistence context bound to
      * the current JTA transaction, it is returned.
      * If no JTA transaction is in progress, an EntityManager
      * instance is created that will be bound to subsequent
      * JTA transactions.
      * Throws IllegalStateException if called on an
      * EntityManagerFactory that does not provide JTA EntityManagers.
      */
     EntityManager getEntityManager();

     /**
      * Close this factory, releasing any resources that might be
      * held by this factory. After invoking this method, all methods
      * on the EntityManagerFactory instance will throw an
      * IllegalStateException, except for isOpen, which will return
      * false.
      */
     void close();

     /**
      * Indicates whether or not this factory is open. Returns true
      *  until a call to close has been made.
      */
     public boolean isOpen();
}
```

The following example illustrates the creation of an EntityManagerFactory, and its use in creating and using a resource-local EntityManager.[24]

```
import javax.persistence.*;

public class PasswordChanger {
     public static void main (String[] args) {

          EntityManagerFactory emf =
               Persistence.createEntityManagerFactory();
          EntityManager em = emf.createEntityManager();

          em.getTransaction().begin();
          user = em.createQuery
             ("SELECT u FROM User u WHERE u.name=:name AND
u.pass=:pass")
             .setParameter("name", args[0])
             .setParameter("pass", args[1])
             .getSingleResult();

          if (user!=null)
               user.setPassword(args[2]);

          em.getTransaction().commit();

          em.close();
          emf.close ();
    }
}
```

Configuration information needed for the creation of an EntityManagerFactory is described in Chapter 6, “Entity Packaging”.

5.2.2.4 Control of the Application-Managed EntityManager Lifecycle.

The EntityManager methods `close` and `isOpen` are used to manage the lifecycle of an application-managed entity manager.

The `EntityManager.close` method closes an entity manager to release its resources. The `close` method must only be invoked when a transaction is not active. The close method must not be invoked on a container-managed entity manager or on an entity manager that has been closed.

The `EntityManager.isOpen` method indicates whether the entity manager is open. The `isOpen` method will return true unless the entity manager has been closed.

[24] Resource-local entity managers are described in Section 5.3.2.

5.3 Controlling Transactions

Transactions involving EntityManager operations may controlled either through JTA or through use of a resource-local `EntityTransaction` API, which is mapped to a resource transaction over the resource that underlies the entities managed by the entity manager.

An entity manager is defined to be of a given transactional type—either JTA or resource-local—at the time its underlying entity manager factory is configured and created.

Both JTA entity managers and resource-local entity managers are required to be supported in J2EE web containers and EJB containers. Within an EJB environment, a JTA entity manager is typically used. In J2SE environments, only resource-local entity managers are supported.

5.3.1 JTA EntityManagers

An entity manager whose transactions are controlled through JTA is termed a JTA entity manager. A JTA entity manager participates in the current JTA transaction, which is begun and committed external to the entity manager and propagated to the underlying resource manager.

Container-managed entity managers can only be JTA entity managers. JTA entity managers are only specified for use in J2EE containers.

5.3.2 Resource-local EntityManagers

An entity manager whose transactions are controlled by the application through the `EntityTransaction` API is termed a resource-local entity manager. A resource-local entity manager transaction is mapped to a resource transaction over the resource by the persistence provider. Resource-local entity managers may use server or local resources to connect to the database and are unaware of the presence of JTA transactions that may or may not be active.

Application-managed entity managers may be either JTA entity managers or resource-local entity managers.

5.3.2.1 The EntityTransaction Interface

The `EntityTransaction` interface is used to control resource transactions on resource-local entity managers. The `EntityManager` `getTransaction` method returns the `EntityTransaction` interface.

```
public interface EntityTransaction {
    /**
     * Start a resource transaction.
     * Throws IllegalStateException if isActive() is true.
     */
    public void begin();

    /**
     * Commit the current transaction, writing any unflushed
     * changes to the database.
     * @throws IllegalStateException if isActive() is false.
     * @throws PersistenceException if the commit fails.
     */
    public void commit();

    /**
     * Roll back the current transaction.
     * @throws IllegalStateException if isActive() is false.
     */
    public void rollback();

    /**
     * Check to see if a transaction is in progress.
     */
    public boolean isActive();
}
```

5.4 Persistence Contexts

As described in chapter 3, a persistence context is a set of entity instances in which for any persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their lifecycle are managed by the entity manager.

A persistence context may be either a transaction-scoped persistence context or an extended persistence context.

A persistence context is either container-managed or application-managed, as described further below.

Examples of persistence context use are given in Section 5.5.

5.4.1 Container-managed Persistence Contexts

A container-managed persistence context is always associated with a container-managed entity manager. The lifecycle of the persistence context is managed automatically by the container.

5.4.1.1 Container-managed Transaction-scoped Persistence Context

A new persistence context begins when a container-manager entity manager is invoked in the scope of an active JTA transaction, and there is no current persistence context already associated with the JTA transaction. The persistence context is created and then associated with the current JTA transaction. The persistence context ends when the associated JTA transaction completes, and all entities that were managed by the EntityManager become detached.

If the entity manager is invoked outside the scope of a transaction, a persistence context is created and destroyed to service the method call only, and any entities loaded from the database will immediately become detached at the end of the method call.

5.4.1.2 Container-managed Extended Persistence Context

An extended persistence context exists from the point at which the container-managed entity manager has been obtained by dependency injection or through JNDI lookup until it is closed by the container. Such an extended persistence context can only be used within the scope of a stateful session bean and is closed by the container when the `@Remove` method of the stateful session bean completes (or the stateful session bean instance is otherwise destroyed).

When an extended persistence context is used, the entities managed by the EntityManager remain managed after a JTA transaction commits. They do not become detached until the persistence context ends.

5.4.2 Application-managed Persistence Contexts

An application-managed persistence context is always associated with an application-managed entity manager. When the persistence context is application managed, the application interacts directly with the persistence provider's entity manager factory to obtain and destroy persistence contexts by means of the `EntityManagerFactory.createEntityManager()` and `EntityManager.close() operations`, and transaction APIs.

5.4.2.1 Application-managed Transaction-scoped Persistence Context

For a JTA entity manager with transaction-scoped persistence context, a new persistence context begins when the entity manager is invoked in the scope of an active JTA transaction, and there is no current persistence context already associated with the entity manager. This persistence context is associated with the entity manager instance. The persistence context ends when the associated JTA transaction completes, and all entities that were managed by the EntityManager become detached. If the entity manager is invoked outside the scope of a transaction, a persistence context is created and destroyed to service the method call only, and any entities loaded from the database will immediately become detached at the end of the method call.

For a resource-local entity manager, a new persistence context begins whenever a new resource transaction is started via `EntityManagerTransaction.begin()`. The persistence context ends when the resource transaction ends, and all entities that were managed by the EntityManager become detached. If the entity manager is invoked outside the scope of a transaction, a persistence context is created and destroyed to service the method call only, and any entities loaded from the database will immediately become detached at the end of the method call.

5.4.2.2 Application-managed Extended Persistence Context

In the case of an application-managed entity manager with extended persistence context (whether a JTA or resource-local entity manager), the extended persistence context exists from the point at which the entity manager has been created until the entity manager is closed, using the `EntityManagerFactory.createEntityManager()` and `EntityManager.close()` APIs for the management of the entity manager lifecycle.

When an extended persistence context is used, the entities managed by the EntityManager remain managed after the JTA transaction or resource-local transaction commits. They do not become detached until the persistence context ends.

5.4.3 Persistence Context Propagation

For container-managed persistence contexts, a single persistence context may correspond to one or more JTA entity manager instances.

Persistence context propagation does not apply to application-managed persistence contexts.

In the case of container-managed persistence contexts of type `PersistenceContextType.TRANSACTION`, the propagation of a JTA transaction causes the propagation of the managed persistence context across the entity managers that are accessed in the same transactional context.[25]

Entity managers in different JTA transactions do not share the same persistence context.

Entity managers obtained from different entity manager factories never share the same persistence context.

5.4.3.1 Persistence Context Propagation for Transaction-scoped Persistence Contexts

The application may obtain a container-managed JTA entity manager with persistence context of type `PersistenceContextType.TRANSACTION` bound to the JTA transaction by injection or JNDI lookup, or by calling `getEntityManager()` on a JTA entity manager factory.

In either case, the returned entity manager accesses a persistence context bound to the JTA transaction:

- If the entity manager is called when no JTA transaction is in progress, a persistence context is created and destroyed to service the method call only, and any entities loaded from the database will immediately become detached at the end of the method call.

- If the entity manager is called and there is no persistence context associated with the current JTA transaction, a new persistence context will be created and bound to the JTA transaction, and the call will take place in that context.

- If the entity manager is called and there is an existing persistence context bound to the current JTA transaction, the call takes place in that context.

[25] Note that these entity managers are associated with the same entity manager factory. See Section 5.6.

5.4.3.2 Persistence Context Propagation Rules for Extended Persistence Contexts

The application may obtain a container-managed JTA entity manager with persistence context of type `PersistenceContextType.EXTENDED` bound to a stateful session bean instance by injection or JNDI lookup.

The following rules apply when the persistence context type of a container-managed entity manager is `EXTENDED`:

- If a component with a transaction-scoped persistence context calls a stateful session bean with an extended persistence context in the same JTA transaction, an IllegalStateException is thrown.
- If a stateful session bean with an extended persistence context calls a stateless session bean or a stateful session bean with a transaction-scoped persistence context in the same JTA transaction, the persistence context is propagated.
- If a stateful session bean with an extended persistence context calls a stateless or stateful session bean in a different JTA transaction context, the persistence context is not propagated.
- If a stateful session bean with an extended persistence context instantiates another stateful session bean with an extended persistence context, the extended persistence context is inherited by the second stateful session bean. If the second stateful session bean is called with a different transaction context than the first, an IllegalStateException is thrown.
- If a stateful session bean with an extended persistence context calls a stateful session bean with a different extended persistence context in the same transaction, an IllegalStateException is thrown.

In general, an exception is thrown if there are two different extended persistence contexts for the same EntityManagerFactory in the same transaction.

5.5 Examples

5.5.1 Container-managed Transaction-scoped Persistence Context

```
@Stateless
public class ShoppingCartImpl implements ShoppingCart {

     @PersistenceContext EntityManager em;

     public Order getOrder(Long id) {
          return em.find(Order.class, id);
     }

     public Product getProduct(String name) {
          return (Product) em.createQuery("select p from Product p
where p.name = :name")
               .setParameter("name", name)
               .getSingleResult();
   }

    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;
   }

}
```

5.5.2 Container-managed Extended Persistence Context

```
@Stateful
@Transaction(REQUIRES_NEW)
public class ShoppingCartImpl implements ShoppingCart {

     @PersistenceContext(type=EXTENDED)
     EntityManager em;

     private Order order;
     private Product prod;

     public void initOrder(Long id) {
          order = em.find(Order.class, id);
     }

     public void initProduct(String name) {
          product = (Product) em.createQuery("select p from Product p
where p.name = :name")
               .setParameter("name", name)
               .getSingleResult();
     }

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

}
```

5.5.3 Application-managed Transaction-scoped Persistence Context (JTA)

```
@Stateless
public class ShoppingCartImpl implements ShoppingCart {

     @PersistenceUnit
     private EntityManagerFactory emf;

     private EntityManager em;

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

     public Order getOrder(Long id) {
          return em.find(Order.class, id);
     }

     public Product getProduct() {
          return (Product) em.createQuery("select p from Product p
where p.name = :name")
               .setParameter("name", name)
               .getSingleResult();
     }

     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;
     }

     @PreDestroy
     public void destroy() {
          em.close();
     }

}
```

5.5.4 Application-managed Extended Persistence Context(JTA)

```
@Stateful
public class ShoppingCartImpl implements ShoppingCart {

     @PersistenceUnit
     private EntityManagerFactory emf;

     private EntityManager em;

     private Order order;
     private Product prod;

     @PostConstruct
     public void init() {
          em = emf.createEntityManager(PersistenceContext-
Type.EXTENDED);
     }

     public void initOrder(Long id) {
          order = em.find(Order.class, id);
     }

     public void initProduct(String name) {
          product = (Product) em.createQuery("select p from Product p
where p.name = :name")
               .setParameter("name", name)
               .getSingleResult();
     }

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

     @Remove
     public void destroy() {
          em.close();
     }

}
```

5.5.5 Application-managed Transaction-scoped Persistence Context (Resource Transaction)

```
public class ShoppingCart {

     private EntityManager em;
     private EntityManagerFactory emf;

     public ShoppingCart() {
          emf = Persistence.createEntityManagerFactory();
          em = emf.createEntityManager();
     }

     public Order getOrder(Long id) {
          return em.find(Order.class, id);
     }

     public Product getProduct() {
          return (Product) em.createQuery("select p from Product p
where p.name = :name")
               .setParameter("name", name)
               .getSingleResult();
     }

     public LineItem createLineItem(Order order, Product product, int
quantity) {
          em.getTransaction().begin();

          LineItem li = new LineItem(order, product, quantity);
          order.getLineItems().add(li);
          em.persist(li);

          em.getTransaction().commit();

          return li;
     }

     public void destroy() {
          em.close();
          emf.close();
     }

}
```

5.5.6 Application-managed Extended Persistence Context (Resource Transaction)

```
public class ShoppingCart {

     private EntityManager em;
     private EntityManagerFactory emf;

     public ShoppingCart() {
          emf = Persistence.createEntityManagerFactory();
          em = emf.createEntityManager(PersistenceContext-
Type.EXTENDED);
     }


     private Order order;
     private Product prod;

     public void initOrder(Long id) {
          order = em.find(Order.class, id);
     }

     public void initProduct(String name) {
          product = (Product) em.createQuery("select p from Product p
where p.name = :name")
               .setParameter("name", name)
               .getSingleResult();
     }

     public LineItem createLineItem(int quantity) {
          em.getTransaction().begin();

          LineItem li = new LineItem(order, product, quantity);
          order.getLineItems().add(li);

          em.getTransaction().commit();

          return li;
     }

     public void destroy() {
          em.close();
          emf.close();
     }

}
```

5.6 Requirements on the Container

5.6.1 Persistence Context Management

For application managed persistence contexts, the application interacts directly with the persistence provider and uses the `EntityManagerFactory` and `EntityManager` APIs to create and destroy persistence contexts. For container managed persistence contexts, the container might use these same APIs or might use its own internal APIs; however, the container is required to support third-party persistence providers. The APIs for the support of third-party persistence providers are described further in Chapter 7.

Persistence contexts are always associated with an entity manager factory. In the following, everywhere that "the persistence context" appears, it should be understood to mean "the persistence context associated with a particular entity manager factory".

Outside the container environment, the application creates an entity manager factory explicitly by calling `Persistence.createEntityManagerFactory()`. Inside the container environment, the container instantiates the entity manager factory and exposes it to the application via JNDI. The container might use internal APIs to create the entity manager factory, or it might use `PersistenceProvider.createContainerEntityManagerFactory()`. However, the container is required to support third-party persistence providers, and in this case, the container must use the `PersistenceProvider.createContainerEntityManagerFactory()` call to create the entity manager factory and must call `EntityManagerFactory.close()` to destroy the entity manager factory prior to shutdown.

5.6.2 Container Managed Persistence Contexts

When operating in a container environment, the container is responsible for managing the lifecycle of persistence contexts, and injecting `EntityManager` references into web components and session bean and message-driven bean components.

The container:

- Begins a new persistence context of type `PersistenceContextType.TRANSACTION` whenever invocation of a business method of a component using an entity manager with `PersistenceContextType.TRANSACTION`, results in the beginning of a new JTA transaction
- Associates that persistence context with the JTA transaction, so that subsequent local business methods which occur in the same JTA transaction also propagate the persistence context
- Ends the persistence context when the JTA transaction completes

The container also:

- Begins a new persistence context of type `PersistenceContextType.EXTENDED` whenever a stateful session bean using an entity manager with `PersistenceContextType.TRANSACTION` is created outside the scope of a JTA transaction and associates that persistence context with the stateful session bean instance

- Associates the persistence context with the current JTA transaction whenever a business method of the stateful bean is invoked, so that
    - subsequent local business methods which occur in the same JTA transaction also propagate the persistence context
    - instantiations of stateful session beans with entity managers with `PersistenceContextType.EXTENDED` associate the persistence context with the new instance of the stateful bean
- Ends the persistence context when the bean is removed

The container is responsible for associating any `EntityManager` references injected into components with the managed persistence context before invoking a business method of the component. The container must also make the managed persistence context available via JNDI lookup .

The rules above can result in "persistence context duplication", where a persistence context associated with the JTA transaction is not the same as the persistence context associated with a stateful bean which is being invoked in the context of that transaction. (See Section 5.4.3 above). For example, this could happen if a business method annotated `Transaction(REQUIRED)` of a stateful session bean using a persistence context of type `PersistenceContextType.EXTENDED` was called from a stateless session bean. The container must detect persistence context duplication and throw the IllegalStateException.

When operating with a third-party persistence provider, the container uses the `EntityManagerFactory/EntityManager` contract defined above to create and destroy persistence contexts. It is undefined whether a new entity manager instance is created for every persistence context, or whether entity manager instances are sometimes reused. Exactly how the container maintains the association between persistence context and JTA transaction is not defined. The container may maintain this association internally, or it may delegate this concern to the persistence provider by using `getEntityManager()` to obtain the provider's current entity manager.

Chapter 6 Entity Packaging

This chapter describes the packaging of persistence units.

The persistence archive file is used to package a persistence unit in J2EE.

In J2SE environments, the metadata mapping files, jar files, and classes described in this chapter are used. The persistence archive file may be used, or the metadata mapping files, jar files, and classes may be packaged in accordance with requirements imposed by the persistence provider.

6.1 Persistence Unit

A persistence unit is a logical grouping that includes:

- A named entity manager together with its provider and configuration information
- The set of managed classes included in the persistence unit for the specified entity manager
- Mapping metadata (in the form of metadata annotations and/or XML) that specifies the mapping of the classes to the database

6.2 Persistence Archive

Within J2EE, the persistence archive, or `.par` file, is the packaging artifact or deployment vehicle for persistence units. Each persistence archive houses a single persistence unit.

Persistence archives may be referenced by J2EE application metadata. If specified in the `application.xml`, persistence archives are listed as follows:

```
<application>
    <module>
        <persistence>orderEntities.par</persistence>
    </module>
</application>
```

Any number of persistence archives may be deployed within a J2EE application (EAR). All persistence archives used by the application must be accessible to all other J2EE components in the application—i.e. loaded by the application classloader—such that if the same entity class is referenced by two different J2EE components (which may be using different persistence units), the referenced class is the same identical class.

6.2.1 persistence.xml file

The configuration information for the entity manager and its entity manager factory, the managed classes included in the persistence unit, and the object/relational mapping information for a persistence unit are defined in a `persistence.xml` file located in the `META-INF` directory of the persistence archive. This information may be defined by containment or by reference, as described below.

The object/relational mapping information may take the form of annotations on the classes included in the persistence archive, one or more XML files contained in the persistence archive, one or more XML files outside the persistence archive on the classpath and referenced from the persistence archive, or a combination of these.

The classes may either be contained within the persistence archive; or they may be specified by reference—i.e., by naming the classes, class archives, or mapping XML files (which in turn reference classes) that are accessible on the application classpath; or they may be specified by some combination of these means.

The `entity-manager` element consists of the following sub-elements: `name`, `provider`, `jta-data-source`, `non-jta-data-source`, `mapping-file`, `jar-file`, `class`, `properties`.

The semantics of the elements are as described below.

For example:

```
<entity-manager>
    <name>em1</name>
    <provider>com.acme.persistence</provider>
    <jta-data-source>jdbc/MyDB</jta-data-source>
    <mapping-file>ormap.xml</mapping-file>
    <jar-file>MyApp.jar</jar-file>
    <class>com.widgets.Order</class>
    <class>com.widgets.Customer</class>
    <properties>
       <property name="sql-logging" value="on"/>
    </properties>
</entity-manager>
```

6.2.1.1 name

Every entity manager must have a name. If no name is specified for an entity manager, the name of its containing persistence archive is used (without the `.par` extension).

For example, if the `persistence.xml` file is contained in `orderEntities.par` and no name is specified for the entity manager, the entity manager's name will be `orderEntities`.

6.2.1.2 provider

The `provider` element specifies the name of the persistence provider's `javax.persistence.spi.PersistenceProvider` class. The `provider` element must be specified if a third-party persistence provider implementation is used.

6.2.1.3 jta-data-source, non-jta-data-source

The `jta-data-source` and `non-jta-data-source` elements are used to specify the global JNDI name of the JTA and/or non-JTA data sources respectively. These elements name the data sources in the local environment; the format of these names and the ability to specify the names are product specific (e.g., they might be provided by other means).

6.2.1.4 mapping-file, jar-file, class

The set of classes that are managed by a persistence unit is defined by using one or more of the following:[26]

- One or more object/relational mapping XML files
- One or more jar files that will be searched for classes
- An explicit list of the classes
- The classes contained in the persistence archive

[26] Note that an individual class may be used in more than one persistence unit.

An object/relational mapping XML file contains the mapping information for the classes listed in it. An `entity-mappings.xml` file may be specified in the `META-INF` directory in the persistence archive or one or more mapping files may be referenced by the `mapping-file` elements of the `entity-manager` element. If a mapping file is specified, the classes and mapping information specified in the mapping file will be used. If multiple mapping files are specified (possibly including an `entity-mappings.xml` file), the resulting mappings are obtained by combining the mappings from all of the files. The result is undefined if multiple mapping files referenced within a single persistence unit contain overlapping mapping information for any given class. The object/relational mapping information contained in any mapping file referenced within the persistence unit must be disjoint at the class-level from object/relational mapping information contained in any other such mapping file.

One or more JAR files may be specified instead of, or in addition to the mapping files. If specified, these JAR files will be searched for entity and embedded classes, and any mapping metadata annotations found on them will be processed, or they will be mapped using the mapping annotation defaults defined by this specification.

A list of named entity and embedded classes may also be specified instead of, or in addition to, the JAR files and mapping files. Any mapping metadata annotations found on these classes will be processed, or they will be mapped using the mapping annotation defaults.

All classes contained in the persistence archive itself are also searched for entity and embedded classes and any mapping metadata annotations found on them will be processed, or they will be mapped using the mapping annotation defaults.

The resulting set of entities managed by the persistence unit is the union of these four sources, with the mapping metadata annotations (or annotation defaults) for any given class being overridden by the XML mapping information file if there are both annotations as well as XML mappings for that class. The level of overriding is at the level of the class: if a class is mapped using XML, all of its mapping information must be specified using XML or none of it must be.

All classes must be on the application classpath to ensure that entity managers from different persistence units that map the same class will be accessing the same identical class.

6.2.1.5 properties

The `properties` element is used to specify vendor-specific properties that apply to the persistence unit and its entity manager factory configuration.

Entries that make use of the namespace `javax.persistence` and its subnamespaces must not be used for vendor-specific information. All names containing `javax.persistence` are reserved for future use by this specification.

6.2.1.6 Examples

The following are sample contents of a `persistence.xml` file. Assume that this file is located in the `META-INF` directory of an `orderEntities.par` persistence archive.

Example 1:

```
<entity-manager>
</entity-manager>
```

A persistence unit is created for entity managers named `orderEntities`.

If a `META-INF/entity-mapping.xml` file exists, any classes referenced by it and mapping information contained in it are used as specified above. Any annotated entity and embedded classes found in the `orderEntities.par` archive are also added to the list of managed classes.

Example 2:

```
<entity-manager>
     <name>EM-2<name>
     <mapping-file>mappings.xml</mapping-file>
</entity-manager>
```

A persistence unit is created for entity managers named `EM-2`. The `mappings.xml` resource exists on the classpath and any classes and mapping information contained in it are used. Any annotated entity and embedded classes found in the `orderEntities.par` archive are also added to the list of managed classes. If a `META-INF/entity-mappings.xml` file exists, any classes and mapping information contained in it are used as well.

Example 3:

```
<entity-manager>
     <name>EM-3</name>
     <jar-file>order.jar</jar-file>
     <jar-file>order-supplemental.jar</jar-file>
</entity-manager>
```

A persistence unit is created for entity managers named `EM-3`. If a `META-INF/entity-mappings.xml` file exists then any classes and mapping information contained in it are used. The `order.jar` and `order-supplemental.jar` are searched for entity and embedded classes and any annotated classes found are added. Any annotated entity and embedded classes found in the `orderEntities.par` archive are also added to the list of managed classes.

Example 4:

```
<entity-manager>
     <name>EM-4</name>
     <mapping-file>order-mappings.xml</mapping-file>
     <class>com.acme.Order</class>
     <class>com.acme.Customer</class>
     <class>com.acme.Item</class>
</entity-manager>
```

A persistence unit is created for entity managers named `EM-4.` The `order-mappings.xml` is read as a resource and any classes referenced by it and mapping information contained in it are used. The annotated `Order`, `Customer` and `Item` classes are loaded and are added. (Note that explicitly enumerated classes must also be annotated). Any annotated entity and embedded classes found in the `orderEntities.par` archive are also added to the list of managed classes.

Example 5:

```
<entity-manager>
     <name>EM-5</name>
     <mapping-file>order1.xml</mapping-file>
     <mapping-file>order2.xml</mapping-file>
     <jar-file>order.par</jar-file>
     <jar-file>order-supplemental.jar</jar-file>
</entity-manager>
```

A persistence unit is created for entity managers named `EM-5`. The `order1.xml` and `order2.xml` files are read as resources and any classes referenced by them and mapping information contained in them are used. The `order.par` is another persistence archive on the classpath, while `order-supplemental.jar` is just a library of classes. Both of these are searched for entity classes and any annotated classes found are added to the list of managed classes. Any annotated entity classes found in the `orderEntities.par` archive are also added.

6.2.2 Default EntityManager

Any persistence archive that exists in the application but does not contain a `persistence.xml` file will have a persistence unit configured using the default values. This is equivalent to specifying an empty `entity-manager` element in a `persistence.xml` file in the archive. This means that it will use the default name (persistence archive name minus the extension); the `entity-mappings.xml` mapping file, if any, that is contained in the persistence archive; and the annotated entity and embedded classes contained in the persistence archive.

6.3 Deployment

The persistence archive is either specified in `application.xml` or discovered through J2EE EAR processing. When the container finds a `.par` file it looks for `META-INF/persistence.xml` file and processes the persistence unit definition that it contains. If no `META-INF/persistence.xml` is found, a default persistence unit configuration is created as specified above.

Chapter 7

Container and Provider Contracts for Deployment and Bootstrapping

This chapter defines requirements on the J2EE container and on the persistence provider for deployment and bootstrapping.

7.1 J2EE Container Deployment

Persistence archives are deployed into the container in the form of persistence archive files, or `.par` files. Each persistence archive file may contain zero or one `persistence.xml` file, any number of mapping files and any number of class files.

7.1.1 Responsibilities of the Container

At deployment time the container is responsible for discovering the persistence archives and processing any persistence.xml files in them. The container must also apply any defaults including:

- EntityManager name
- `entity-mapping.xml` mapping file

- set of managed classes

The defaults for these are described in Chapter 6.

The container may optionally add its own container-defaulted rules and values for such properties as the persistence provider, the data source, or any container-specific properties.

Once the container has read the persistence metadata, it determines the provider's `javax.persistence.spi.PersistenceProvider` implementation class for each deployed named EntityManager. It creates an instance of this implementation class and invokes the `createContainerEntityManagerFactory` method on this instance. The metadata is passed into the persistence provider as part of this call. This occurs once for each named EntityManager configuration.The factory obtained will be used by the container to create container-managed EntityManagers. Only one EntityManagerFactory may be created for each named EntityManager configuration. Any number of EntityManager instances may be created from a given factory.

When a persistence archive is redeployed then the container must call the `createContainerEntityManagerFactory` method again, with the required metadata, to indicate the deployment.

7.1.2 Responsibilities of the Persistence Provider

The persistence provider must implement the `PersistenceProvider` SPI and be able to process the metadata that is passed to it at the time `createContainerEntityManagerFactory` method is called. An instance of `EntityManagerFactory` is created and the metadata for the named EntityManager is associated with the factory. The factory is then returned to the container. The factory instance must implement `javax.naming.Referenceable`.

7.1.3 javax.persistence.spi.PersistenceProvider

The interface `javax.persistence.spi.PersistenceProvider` is implemented by the persistence provider, and is specified in the `persistence.xml` file in the persistence archive. It is invoked by the container when it needs to create an EntityManagerFactory, or by the `javax.persistence.Persistence` class when running outside the container.

```
package javax.persistence.spi;

/**
 * Interface implemented by a persistence provider.
 * The implementation of this interface that is to
 * be used for a given EntityManager is specified in
 * persistence.xml file in the persistence archive.
 * This interface is invoked by the Container when it
 * needs to create an EntityManagerFactory, or by the
 * Persistence class when running outside the Container.
 */
public interface PersistenceProvider {

     /**
      * Called by Persistence class when an EntityManagerFactory
      * is to be created.
      *
      * @param emName The name of the EntityManager configuration
      * for the factory
      * @param map A Map of properties that may be used by the
      * persistence provider
      * @return EntityManagerFactory for the named EntityManager,
      * or null if the provider is not the right provider
      */
     public EntityManagerFactory createEntityManagerFactory(String
emName, Map map);

     /**
      * Called by the container when an EntityManagerFactory
      * is to be created.
      *
      * @param info Metadata needed by the provider
      * @return EntityManagerFactory for the named EntityManager
      */
      public EntityManagerFactory createContainerEntityManagerFac-
tory(PersistenceInfo info);
}
```

7.1.4 javax.persistence.spi.PersistenceInfo Interface

```
package javax.persistence.spi;

import javax.sql.DataSource;

/**
 * Interface implemented and used by the Container to pass
 * persistence metadata to the persistence provider as part of
 * the createContainerEntityManagerFactory() call. The provider
 * will use this metadata to obtain the mappings and initialize
 * its structures.
 */
public interface PersistenceInfo {

/**
 * @return The name of the EntityManager that is being created.
 * Corresponds to the <name> element in persistence.xml
 */
public String getEntityManagerName();

/**
 * @returns The name of the persistence provider implementation
 * class.
 * Corresponds to the <provider> element in persistence.xml
 */
public String getPersistenceProviderClassName();

/**
 * @return the JTA-enabled data source to be used by the
 * persistence provider.
 * The data source corresponds to the named <jta-data-source>
 * element in persistence.xml
 */
public DataSource getJtaDataSource();

/**
 * @return The non-JTA-enabled data source to be used by the
 * persistence provider when outside the container, or inside
 * the container when accessing data outside the global
 * transaction.
 * The data source corresponds to the named <non-jta-data-source>
 * element in persistence.xml
 */
public DataSource getNonJtaDataSource();

/**
 * @return The list of mapping file names that the persistence
 * provider must load to determine the mappings for the entity
 * classes. The mapping files must be in the standard XML
 * mapping format, be uniquely named and be resource-loadable
 * from the application classpath. This list will not include
 * the entity-mappings.xml file if one was specified.
 * Each mapping file name corresponds to a <mapping-file>
 * element in persistence.xml
 */
public List<String> getMappingFileNames();

/**
 * @return The list of JAR file URLs that the persistence
```

```
     * provider must look in to find the entity classes that must
     * be managed by EntityManagers of this name. The persistence
     * archive jar itself will always be the last entry in the
     * list. Each jar file URL corresponds to a named <jar-file>
     * element in persistence.xml
     */
    public List<URL> getJarFiles();

    /**
     * @return The list of class names that the persistence
     * provider must inspect to see if it should add it to its
     * set of managed entity classes that must be managed by
     * EntityManagers of this name.
     * Each class name corresponds to a named <class> element
     * in persistence.xml
     */
    public List<String> getEntityClassNames();

    /**
     * @return Properties object that may contain vendor-specific
     * properties contained in the persistence.xml file.
     * Each property corresponds to a <property> element in
     *persistence.xml
     */
    public Properties getProperties();

    /**
     * @return ClassLoader that the provider may use to load any
     * classes, resources, or open URLs.
     */
    public ClassLoader getClassLoader();

    /**
     * @return URL object that points to the persistence.xml
     * file; useful for providers that may need to re-read the
     * persistence.xml file. If no persistence.xml
     * file is present in the persistence archive, null is
     * returned.
     */
    public URL getPersistenceXmlFileUrl();

    /**
     * @return URL object that points to the entity-mappings.xml
     * file.
     * If no entity-mappings.xml file was present in the persistence
     * archive,null is returned.
     */
    public URL getEntityMappingsXmlFileUrl();
}
```

7.2 Bootstrapping in J2SE Environments

In J2SE environments (outside the J2EE container), the `Persistence.createEntityManagerFactory` call is used by the application to create an entity manager factory. To find the provider for the named EntityManager configuration then `Persistence` class does the following:

- Looks up all of the persistence provider services that exist on the context classpath.
- Instantiates each of the provider classes and invokes `createEntityManagerFactory` on the providers until one of the calls returns an EntityManagerFactory instance.
- Returns the factory or an error if none was able to be obtained from the known providers.

Persistence providers may require that all persistence archives contain persistence.xml files. The persistence provider is responsible for discovering all of the `persistence.xml` files in the `.par` files and processes them in order to discover the providers for all of the archives. Persistence providers may also require that the set of entity classes and classes that are to be managed must be fully enumerated in each of the persistence.xml files.

Chapter 8 Metadata Annotations

This chapter and chapter 9 define the metadata annotations introduced by this specification.

The XML schema defined in chapter 10 provides an alternative to the use of metadata annotatations.

These annotations are in the package `javax.persistence`.

8.1 Entity

The `Entity` annotation specifies that the class is an entity. This annotation is applied to the entity class.

The `name` annotation element defaults to the unqualified name of the entity class. This name is used to refer to the entity in queries. The name must not be a reserved literal in EJB QL.

```
@Target(TYPE) @Retention(RUNTIME)
public @interface Entity {
   String name() default "";
   AccessType access() default PROPERTY;
}
```

The enum `AccessType` is used to specify whether the persistence provider runtime uses properties or fields to access the entity state. The `AccessType` for an entity class determines whether its object/relational mapping annotations are applied to its property methods or to its instance variables.

```
public enum AccessType {
  PROPERTY,
  FIELD
}
```

8.2 Callback Annotations

The EntityListener annotation specifies the callback listener class to be used for an entity.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface EntityListener {
  Class value();
}
```

The following annotations are used to specify callback methods for the corresponding lifefcycle events. These annotations may be applied to methods on the entity class or methods of the EntityListener class.

```
@Target({METHOD}) @Retention(RUNTIME)
public @interface PrePersist {}

@Target({METHOD}) @Retention(RUNTIME)
public @interface PostPersist {}

@Target({METHOD}) @Retention(RUNTIME)
public @interface PreRemove {}

@Target({METHOD}) @Retention(RUNTIME)
public @interface PostRemove {}

@Target({METHOD}) @Retention(RUNTIME)
public @interface PreUpdate {}

@Target({METHOD}) @Retention(RUNTIME)
public @interface PostUpdate {}

@Target({METHOD}) @Retention(RUNTIME)
public @interface PostLoad {}
```

8.3 Annotations for Queries

8.3.1 Flush Mode Annotation

[Note to readers] *The semantics and applicability points of the* `FlushMode` *annotation are still currently undergoing review.*

The `FlushMode` annotation is used on a client component to designate whether entities should be flushed to the database as part of a query or a method's behavior. For example, the `FlushMode` annotation can be used to control whether or not queries return entities that have been made persistent or removed in the current transaction.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface FlushMode {
  FlushModeType value();
}

public enum FlushModeType {
  COMMIT,
  AUTO,
  NEVER
}
```

`FlushMode(AUTO)` will cause flushes to occur at commit and before query execution. `FlushMode(COMMIT)` will cause flush to occur only at transaction commit; the persistence provider runtime is permitted to flush before query execution.

`FlushMode(NEVER)` will cause changes not to be written to the database unless the `flush()` method is called.

8.3.2 NamedQuery Annotation

The `NamedQuery` annotation is used to specify a named EJB QL query. The `name` element is used to refer to the query when using the EntityManager methods that create query objects.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface NamedQuery {
   String name();
   String queryString();
}
@Target({TYPE}) @Retention(RUNTIME)
public @interface NamedQueries {
   NamedQuery [] value ();
}
```

[Note to readers] *Means for handling application-level metadata is still under discussion. We would like to be able to use annotations for named queries that are logically scoped to a persistence unit rather than to a specific class or method. This applies also to the NamedNativeQuery annotations.*

8.3.3 NamedNativeQuery Annotation

The `NamedNativeQuery` annotation is used to specify a native SQL named query. The name element is used to refer to the query when using the EntityManager methods that create query objects. The `resultClass` element refers to the class of the result; the value of the `resultSetMapping` element is the name of a `SQLResultSetMapping`, as defined in metadata.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface NamedNativeQuery {
   String name();
   String queryString();
   Class resultClass() default void.class;
   String() resultSetMapping() default ""; // name of SQLResultSetMap-
ping
}

@Target({TYPE}) @Retention(RUNTIME)
public @interface NamedNativeQueries {
   NamedNativeQuery [] value ();
}
```

8.3.4 Annotations for SQL Query Result Set Mappings

The `SqlResultSetMapping` annotation is used to specify the mapping of the result of a native SQL query.

```
@Target({TYPE, METHOD}) @Retention(RUNTIME)
public @interface SqlResultSetMapping {
     String name();
     EntityResult[] entities() default {};
     ColumnResult[] columns() default {};
}
```

The `name` element is the name given to the result set mapping, and used to refer to it in the methods of the Query API. The `entities` and `columns` elements are used to specify the mapping to entities and to scalar values respectively.

```
@Target({}) @Retention(RUNTIME)
public @interface EntityResult {
     Class entityClass();
     FieldResult[] fields() default {};
     String discriminatorColumn() default "";
}
```

The `entityClass` element specifies the class of the result.

The `discriminatorColumn` element is used to specify the column name (or alias) of the column in the SELECT list that is used to determine the type of the entity instance.

The `fields` element is used to map the columns specified in the SELECT list of the query to the properties or fields of the entity class.

```
@Target({}) @Retention(RUNTIME)
public @interface FieldResult {
     String name();
     String column();
}
```

The `name` element is the name of the persistent field or property of the class.

The `column` element is the column name (or alias) as specified in the SELECT list.

```
@Target({}) @Retention(RUNTIME)
public @interface ColumnResult {
     String name();
}
```

8.4 References to EntityManager and EntityManagerFactory

These annotations are used to express dependencies on entity managers and entity manager factories.

[Note to readers] *The names of these annotations are currently undergoing review, and are subject to change.*

8.4.1 PersistenceContext Annotation

The `PersistenceContext` annotation is used to express a dependency on an EntityManager persistence context.

The `name` element refers to the name by which the EntityManager and its persistence unit are to be known in the environment referencing context, and is not needed when dependency injection is used.

The `unitName` element refers to the name of the persistence unit. It must be specified if there is more than one persistence unit.

The `type` element specifies whether a transaction-scoped or extended persistence context is to be used.

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface PersistenceContext{
    String name() default "";
    String unitName() default "";
    PersistenceContextType type default TRANSACTION;
}
public enum PersistenceContextType {
  TRANSACTION,
  EXTENDED
}

@Target(TYPE) @Retention(RUNTIME)
public @interface PersistenceContexts{
  PersistenceContexts[] value();
}
```

8.4.2 PersistenceUnit Annotation

The `PersistenceUnit` annotation is used to express a dependency on an EntityManagerFactory.

The `name` element refers to the name by which the EntityManagerFactory is to be known in the environment referencing context, and is not needed when dependency injection is used.

The `unitName` element refers to the name of the persistence unit. It must be specified if there is more than one persistence unit.

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface PersistenceUnit{
    String name() default "";
    String unitName() default "";
}


@Target(TYPE) @Retention(RUNTIME)
public @interface PersistenceUnits{
  PersistenceUnit[] value();
}
```

Chapter 9 Metadata for Object/Relational Mapping

The object/relational mapping metadata expressed by an application is part of the application domain model contract.

The object/relational mapping metadata expresses requirements and expectations on the part of the application as to the mapping of the entities and relationships of the application domain to a database. Queries (and, in particular, SQL queries) written against the database schema that corresponds to the application domain model are dependent upon the mappings expressed by means of the object/relational mapping metadata.

The implementation of this specification must assume the application logic to be dependent upon the object/relational mapping expressed in metadata.

It is permitted, but not required, that DDL generation be supported by an implementation of this specification. The annotation elements that specify such DDL are intended as hints to the implementation for DDL generation. Use of such hints is not portable.

9.1 Annotations for Object/Relational Mapping

These annotations are in the package `javax.persistence`.

[Note to readers] *We are currently examining how we might use metadata annotations at the persistence unit level to allow defaults to be set for an entire persistence unit. These defaults would include settings for access type, cascade mode, and flush mode.*

9.1.1 Table Annotation

The Table `annotation` specifies the primary table for the annotated entity. Additional tables may be specified using `SecondaryTable` or `SecondaryTables` annotation. If no `Table` annotation is specified for an entity class, all of the default values defined by the `Table` annotation will apply.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Table {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    UniqueConstraint[] uniqueConstraints() default {};
    boolean specified() default true; // For internal use only[27]
}
```

Table 4 lists the annotation elements that may be specified for a `Table` annotation.

Table 4 Table Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the table.	Unqualified class name of the entity
String	catalog	(Optional) The catalog of the table.	Default catalog
String	schema	(Optional) The schema of the table.	Default schema for user
UniqueConstraint[]	uniqueConstraints	(Optional) Unique constraints that should be placed on the table. These are only used if table generation is in effect. These constraints apply in addition to any constraints specified by the Column and JoinColumn annotations, or entailed by primary key mappings.	No constraints

Example:

```
@Entity
@Table(name="CUST", schema="RECORDS")
public class Customer { ... }
```

[27] Note to the reader: use of this element, where specified=FALSE, allows this annotation to be treated as an optional element of a containing annotation. See, e.g., JoinTable.

9.1.2 SecondaryTable Annotation

The `SecondaryTable` annotation is used to specify a secondary table for an entity class. Specifying one or more secondary tables indicates that the entity data is stored across multiple tables.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface SecondaryTable {
    String name();
    String catalog() default "";
    String schema() default "";
    PrimaryKeyJoinColumn[] pkJoin() default {};
    UniqueConstraint[] uniqueConstraints() default {};
}
```

Table 5 lists the annotation elements that may be specified for a `SecondaryTable` annotation.

Table 5 SecondaryTable Annotation Elements

Type	Name	Description	Default
String	name	(Required) The name of the table.	
String	catalog	(Optional) The catalog of the table.	Default catalog
String	schema	(Optional) The schema of the table.	Default schema for user
PrimaryKeyJoin-Column[]	pkJoin	(Optional) The columns that should be used to join with the primary table.	Column(s) of the same name as the primary key column(s) in the primary table
UniqueConstraint[]	uniqueConstraints	(Optional) Unique constraints that should be placed on the table. These are typically only used if table generation is in effect. These constraints apply in addition to any constraints specified by the Column and Join-Column annotations, or entailed by primary key mappings.	No constraints

If no `SecondaryTable` annotation is specified, it is assumed that all properties of the entity are mapped to the primary table. If no primary key join column is specified, the join columns are assumed to reference the primary key columns of the primary table, and have the same names as the referenced columns.

Example: Single secondary table with a single primary key column

```
@Entity
@Table(name="CUSTOMER")
@SecondaryTable(name="CUST_DETAIL",
  pkJoin=@PrimaryKeyJoinColumn(name="CUST_ID"))
public class Customer { ... }
```

Example: Single secondary table with multiple primary key columns

```
@Entity
@Table(name="CUSTOMER")
@SecondaryTable(name="CUST_DETAIL",
  pkJoin=@PrimaryKeyJoinColumns({
    @PrimaryKeyJoinColumn(name="CUST_ID"),
    @PrimaryKeyJoinColumn(name="CUST_TYPE")}))
public class Customer { ... }
```

9.1.3 SecondaryTables Annotation

An entity may have multiple secondary tables. In this case they must be enclosed within a `SecondaryTables` annotation. A `SecondaryTables` annotation takes an array of `SecondaryTable` annotations as its single annotation element.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface SecondaryTables {
    SecondaryTable[] value();
}
```

Example: Multiple secondary tables assuming primary key columns are named the same in all tables

```
@Entity
@Table(name="EMPLOYEE")
@SecondaryTables({
    @SecondaryTable(name="EMP_DETAIL"),
    @SecondaryTable(name="EMP_HIST")
})
public class Employee { ... }
```

Example: Multiple secondary tables with differently named primary key columns

```
@Entity
@Table(name="EMPLOYEE")
@SecondaryTables({
    @SecondaryTable(name="EMP_DETAIL",
        pkJoin=@PrimaryKeyJoinColumn(name="EMPL_ID")),
    @SecondaryTable(name="EMP_HIST",
        pkJoin=@PrimaryKeyJoinColumn(name="EMPLOYEE_ID"))
})
public class Employee { ... }
```

9.1.4 UniqueConstraint Annotation

The `UniqueConstraint` annotation is used to specify that a unique constraint should be included in the generated DDL for a primary or secondary table.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface UniqueConstraint {
    String[] columnNames();
}
```

Table 6 lists the annotation elements that may be specified for a `UniqueConstraint` annotation.

Table 6 UniqueConstraint Annotation Elements

Type	Name	Description	Default
String[]	columnNames	(Required) An array of the column names that make up the constraint.	

Example:

```
@Entity
@Table(
    name="EMPLOYEE",
    uniqueConstraints=
        {@UniqueConstraint(columnNames={"EMP_ID", "EMP_NAME"})}
)
public class Employee { ... }
```

9.1.5 Column Annotation

The `Column` annotation is used to specify a mapped column for a persistent property or field. If a `Column` annotation is not specified, or if the name annotation element is missing, the column name defaults to the persistent property or field name.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Column {
    String name() default "";
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String secondaryTable() default "";
    int length() default 255;
    int precision() default 0; // decimal precision
    int scale() default 0; // decimal scale
}
```

Table 7 lists the annotation elements that may be specified for a `Column` annotation.

Table 7 Column Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the column.	The property or field name

Type	Name	Description	Default
boolean	unique	(Optional) Whether the property is a unique key. This is a shortcut for the UniqueConstraint annotation at the table level and is useful for when the unique key constraint is only a single field. This constraints applies in addition to any constraint entailed by primary key mapping.	false
boolean	nullable	(Optional) Whether the database column is nullable.	true
boolean	insertable	(Optional) Whether the column should be included in SQL INSERT statements generated by the persistence provider.	true
boolean	updatable	(Optional) Whether the column should be included in SQL UPDATE statements generated by the persistence provider.	true
String	columnDefinition	(Optional) The SQL fragment that is used when generating the DDL for the column.	Generated SQL to create a column of the inferred type.
String	secondaryTable	(Optional) The name of the secondary table that contains the column. If absent the column is assumed to be in the primary table.	Column is in primary table.
int	length	(Optional) The column length.	255
int	precision	(Optional) The precision for a decimal column.	0
int	scale	(Optional) The scale for a decimal column.	0

Examples:

```
@Column(name="DESC", nullable=false, length=512)
public String getDescription() { return description; }

@Column(name="DESC",
    columnDefinition="CLOB NOT NULL",
    secondaryTable="EMP_DETAIL")
public String getDescription() { return description; }

@Column(name="ORDER_COST", updatable=false, precision=12, scale=2)
public BigDecimal getCost() { return cost; }
```

9.1.6 JoinColumn Annotation

The `JoinColumn` annotation is used to specify a mapped column for joining an entity association. The `name` annotation element defines the name of the foreign key column. The remaining annotation elements (other than `referencedColumnName`) refer to this column and have the same semantics as for the `Column` annotation.

If the `referencedColumnName` element is missing, the foreign key is assumed to refer to the primary key of the referenced table.

If the `name` annotation element is missing, or if no `JoinColumn` annotation is specified, the join columns are assumed to have the same names as the primary key columns of the referenced table.

If no `JoinColumn` annotation is specified, a single join column is assumed. The defaults for the join column are as described below.

If there is a single join column, then

- If the `name` annotation member is missing, the join column name is formed as the concatenation of the following: the name of the referencing relationship property or field of the referencing entity; "_"; the name of the referenced primary key column.

- If the `referencedColumnName` element is missing, the foreign key is assumed to refer to the primary key of the referenced table.

If there is more than one join column, a `JoinColumn` annotation must be specified for each join column. Both the `name` and the `referencedColumnName` elements must be specified in each such `JoinColumn` annotation.

```
@Target({TYPE, METHOD, FIELD})  @Retention(RUNTIME)
public @interface JoinColumn {
     String name() default "";
     String referencedColumnName() default "";
     boolean unique() default false;
     boolean nullable() default true;
     boolean insertable() default true;
     boolean updatable() default true;
     String columnDefinition() default "";
     String secondaryTable() default "";
}
```

Support for referenced columns that are not the primary key of the referenced table is optional in this release, but will be required in the next.

Table 8 lists the annotation elements that may be specified for a `JoinColumn` annotation.

Table 8 JoinColumn Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the foreign key column. The table in which it is found depends upon the context. If the join is for a OneToOne or ManyToOne mapping or for a secondary table joined to a primary table, then the foreign key column is in the table of the source entity. If the join is for a ManyToMany then the foreign key is in a join table.	(Only applies if single join column is being used.) The concatenation of the following: the name of the referencing relationship property or field of the referencing entity; "_"; the name of the referenced primary key column.

Type	Name	Description	Default
String	referencedColumnName	(Optional) The name of the column referenced by this foreign key column. When used with mappings, the referenced column is in the table of the target entity. When used inside a JoinTable annotation, the referenced key column is in the entity table of the owning entity, or inverse entity if the join is part of the inverse join definition.	(Only applies if single join column is being used.) The same name as the primary key column of the referenced table.
boolean	unique	(Optional) Whether the property is a unique key. This is a shortcut for the UniqueConstraint annotation at the table level and is useful for when the unique key constraint is only a single field. It is not necessary to explicitly specify this for a join column that corresponds to a primary key that is part of a foreign key.	false
boolean	nullable	(Optional) Whether the foreign key column is nullable.	true
boolean	insertable	(Optional) Whether the column should be included in SQL INSERT statements generated by the persistence provider.	true
boolean	updatable	(Optional) Whether the column should be included in SQL UPDATE statements generated by the persistence provider.	true
String	columnDefinition	(Optional) The SQL fragment that is used when generating the DDL for the column.	Generated SQL to create a column of the inferred type.
String	secondaryTable	(Optional) The name of the secondary table that contains the column. If absent the column is assumed to be in the primary table of the applicable entity.	Not set, column is in primary table.

Examples:

```
@ManyToOne
@JoinColumn(name="ADDR_ID")
public Address getAddress() { return address; }
```

9.1.7 JoinColumns Annotation

Composite keys are supported via the `JoinColumns` annotation. This allows grouping a number of `JoinColumn` specifications for the same relationship or table association.

```
@Target({TYPE, METHOD, FIELD})  @Retention(RUNTIME)
public @interface JoinColumns {
    JoinColumn[] value();
}
```

Example:

```
@ManyToOne
@JoinColumns({
    @JoinColumn(name="ADDR_ID", referencedColumnName="ID"),
    @JoinColumn(name="ADDR_ZIP", referencedColumnName="ZIP")
})
public Address getAddress() { return address; }
```

9.1.8 Id Annotation

The `Id` annotation selects the identifier property of an entity root class. By default, the mapped columns of this property are assumed to form the primary key of the primary table. If no `Column` annotation is specified, the primary key column name is assumed to be the name of the identifier property or field.

Primary key generation strategies may also be specified in the `Id` annotation. The types of id generation are defined by the `GeneratorType` enum:

```
public enum GeneratorType { TABLE, SEQUENCE, IDENTITY, AUTO, NONE };
```

The TABLE strategy indicates that the persistence provider should assign identifiers using an underlying database table to ensure uniqueness. The SEQUENCE and IDENTITY strategies specify the use of a database sequence or identity column, respectively. AUTO indicates that the persistence provider should pick an appropriate strategy for the particular database. Specifying NONE indicates that no primary key generation by the persistence provider should occur, and that the application will be responsible for assigning the primary key. This specification does not define the exact behavior of these strategies.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Id {
    GeneratorType generate() default NONE;
    String generator() default "";
}
```

Table 9 lists the annotation elements that may be specified for an `Id` annotation.

Table 9 Id Annotation Elements

Type	Name	Description	Default
GeneratorType	generate	(Optional) The type of primary key generation that the persistence provider should use to generate the annotated entity primary key.	GeneratorType.NONE (no primary key generation)
String	generator	(Optional) The generator annotation element selects a specific primary key generator that is defined by an annotation.	Default id generator supplied by persistence provider.

Examples:

```
@Id
public Long getId() { return id; }

@Id(generate=SEQUENCE, generator="CUST_SEQ")
@Column(name="CUST_ID")
public Long getId() { return id; }

@Id(generate=TABLE, generator="CUST_GEN")
@Column(name="CUST_ID")
Long id;
```

9.1.9 AttributeOverride Annotation

The `AttributeOverride` annotation is used to override mappings of properties or fields. Columns in the overrides apply to the current primary table for the class that contains the annotation. The `AttributeOverride` (or `AttributeOverrides`) annotation may be used on an entity that extends an embeddable superclass or on an embedded field or property. If `AttributeOverride` is not specified, the column is mapped the same as in the original mapping.

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface AttributeOverride {
    String name();
    Column column();
}
```

Table 10 lists the annotation elements that may be specified for a `AttributeOverride` annotation.

Table 10 AttributeOverride Annotation Elements

Type	Name	Description	Default
String	name	(Required) The name of the property that is being mapped if access is set to PROPERTY, or the name of the field if access is set to FIELD in the embedded object.	
Column	column	(Required) The column that is being mapped to the persistent attribute. The mapping type will remain the same as is defined in the embeddable class.	

9.1.10 AttributeOverrides Annotation

The mappings of multiple properties or fields may be overridden. In this case, the overriding information must be enclosed with an `AttributeOverrides` annotation.

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface AttributeOverrides {
  AttributeOverride[] value();
}
```

9.1.11 EmbeddedId Annotation

The `EmbeddedId` annotation is used to denote a composite primary key that is an embeddable class. It may be applied to a persistent field or property of the entity class. There should only be one `EmbeddedId` annotation and no `Id` annotations when the `EmbeddedId` annotation is used.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface EmbeddedId {}
```

Example:

```
@EmbeddedId
protected EmployeePK empPK;
```

9.1.12 IdClass Annotation

The `IdClass` annotation is used to denote a composite primary key. It is applied to the entity class. The composite primary key class corresponds to multiple fields or properties of the entity class, and the names of primary key fields or properties in the primary key class and those of the entity class must correspond and their types must be the same. See Section 2.1.4, “Primary Keys and Entity Identity”. The `Id` annotation may also be applied to such fields or properties, however this is not required.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface IdClass {
    Class value();
}
```

Example:

```
@IdClass(com.acme.EmployeePK.class)
@Entity(access=FIELD)
public class Employee {
  @Id String empName;
  @Id Date birthDay;
...
}
```

9.1.13 Transient Annotation

The `Transient` annotation is used to annotate a property or field of the entity class. It specifies that the property or field is not persistent.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Transient {}
```

9.1.14 Version Annotation

The `Version` annotation specifies the version property (optimistic lock value) of an entity class. This is used to ensure integrity when reattaching and for overall optimistic concurrency control. Only a single `Version` property/field should be used per class; applications that use more than one are not expected to be portable. The `Version` property should be mapped to the primary table for the entity class; applications that map the `Version` property to a table other than the primary table are not portable.

Fields or properties that are specified with the `Version` annotation should not be updated by the application.

The following types are supported for version properties: `int`, `Integer`, `short`, `Short`, `long`, `Long`, `Timestamp`.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Version {}
```

Example:

```
@Version
@Column("OPTLOCK")
protected int getVersionNum() { return versionNum; }
```

9.1.15 Basic Annotation

The `Basic` annotation is the simplest type of mapping to a database column. It can optionally be applied to any persistent property or instance variable of the following type: Java primitive types, wrappers of the primitive types, `java.lang.String`, `java.math.BigInteger`, `java.math.BigDecimal`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`, `byte[]`, `Byte[]`, `char[]`, `Character[]`, enums, and any other type that implements Serializable.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Basic {
    FetchType fetch() default EAGER;
    TemporalType temporalType() default NONE;
    boolean optional() default true;
}
```

The `FetchType` enum defines strategies for fetching data from the database:

```
public enum FetchType { LAZY, EAGER };
```

The EAGER strategy is a requirement on the persistence provider runtime that data should be eagerly fetched. The LAZY strategy is a *hint* to the persistence provider runtime that data should be fetched lazily when it is first accessed. The implementation is permitted to eagerly fetch data for which the LAZY strategy hint has been specified. For `Basic` properties, lazy fetching might only be available for properties which are always accessed via the get/set pair.

The `TemporalType` enum defines the mapping for temporal types.

```
public enum TemporalType {
    DATE, //java.sql.Date
    TIME, //java.sql.Time
    TIMESTAMP, //java.sql.Timestamp
    NONE
}
```

The `optional` element can be used a a hint as to whether the value of the field or property may be null. It is disregarded for primitive types, which are considered non-optional.

Table 11 lists the annotation elements that may be specified for a `Basic` annotation.

Table 11 Basic Annotation Elements

Type	**Name**	**Description**	**Default**
FetchType	fetch	(Optional) Whether the value of the field or property should be lazy loaded or eagerly fetched.	EAGER
TemporalType	tempo-ralType	(Optional) The type used in mapping a tempo-ral type.	NONE
boolean	optional	(Optional) Whether the value of the field or property may be null. This is a hint and is dis-regarded for primitive types; it may be used in schema generation.	true

Examples:

```
@Basic
protected String name;

@Basic(fetch=LAZY)
protected String getName() { return name; }

@Basic(fetch=LAZY)
@Column(name="EMP_PIC")
protected byte[] pic;
```

An enum can be mapped to either a string or an integer, and depending upon the column type either the ordinal value or string value of the enum will be stored.

Example:

```
public enum EmployeeStatus { FULL_TIME, PART_TIME, SEASONAL, CONTRACT
}
public enum SalaryRate { PAROLE, JUNIOR, INTERMEDIATE, SENIOR, MAN-
AGER, EXECUTIVE }

@Entity public class Employee {
    ...
    public EmployeeStatus getStatus() {...}
    public SalaryRate getPayScale() {...}
    ...
}
```

If the status property were mapped to a column of type integer, and the payscale property to a column of varchar type, an instance that had a status of `PART_TIME` and a pay rate of `JUNIOR` would have a row stored in the table with `STATUS` set to 1 and `PAYSCALE` set to `"JUNIOR"`.

9.1.16 Lob Annotation

A `Lob` annotation specifies that a persistent property or field should be persisted as a large object to a database-supported large object type. A Lob may be either a binary or character type, as defined by the `LobType` enum.

```
public enum LobType { BLOB, CLOB };

@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Lob {
     FetchType fetch() default LAZY;
     LobType type() default BLOB;
     boolean optional() default true;
}
```

Blob fields may be defined to be of type `Byte[]` or a Serializable type.

Clob fields may be defined to be of type `char[]`, `Character[]` or `String`.

The LAZY strategy for Blob and Clob fields is a `hint` to the persistence provider runtime that data should be fetched lazily when it is first accessed. The EAGER strategy is a requirement on the persistence provider runtime that data should be eagerly fetched.

Examples:

```
@Lob
@Column(name="PHOTO" columnDefinition="BLOB NOT NULL")
protected JPEGImage picture;

@Lob(fetch=EAGER, type=CLOB)
@Column(name="REPORT")
protected String report;
```

Table 12 LobAnnotation Elements

Type	Name	Description	Default
FetchType	fetch	(Optional) Whether the lob should be lazy loaded or eagerly fetched.	LAZY
LobType	type	(Optional) The type of the lob.	BLOB
boolean	optional	(Optional) Whether the value of the field or property may be null. This is a hint; it may be used in schema generation.	true

9.1.17 ManyToOne Annotation

The `ManyToOne` annotation defines a single-valued association to another entity class that has many-to-one multiplicity. It is not normally necessary to specify the target entity explicitly since it can usually be inferred from the type of the object being referenced.

The `cascade` set will cause the specified cascadable operations to be propagated to the associated entity. The operations that are cascadable are defined by the `CascadeType` enum:

```
public enum CascadeType { ALL, PERSIST, MERGE, REMOVE, REFRESH};
```

Multiple operations may be included in the set. The value `cascade=ALL` is equivalent to `cascade={PERSIST, MERGE, REMOVE, REFRESH}`.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface ManyToOne {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
}
```

The EAGER strategy is a requirement on the persistence provider runtime that the associated entity should be eagerly fetched. The LAZY strategy is a *hint* to the persistence provider runtime that the associated entity should be fetched lazily when it is first accessed. The implementation is permitted to eagerly fetch associations for which the LAZY strategy hint has been specified.

Table 13 lists the annotation elements that may be specified for a `ManyToOne` annotation.

Table 13 ManyToOne Annotation Elements

Type	Name	Description	Default
Class	targetEntity	(Optional) The entity class that is the target of the association.	The type of the field or property that stores the association.
CascadeType[]	cascade	(Optional) The operations that should be cascaded to the target of the association.	No operations are cascaded.
FetchType	fetch	(Optional) Hint to the implementation as to whether the association should be lazy loaded or eagerly fetched. The EAGER strategy is a requirement on the persistence provider runtime that the associated entity should be eagerly fetched.	EAGER
boolean	optional	(Optional) Whether the association is optional. If set to false then a non-null relationship must always exist.	true

Example:

```
@ManyToOne(optional=false)
@JoinColumn(name="CUST_ID", nullable=false, updatable=false)
public Customer getCustomer() { return customer; }
```

9.1.18 OneToOne Annotation

The `OneToOne` annotation defines a single-valued association to another entity that has one-to-one multiplicity. It is not normally necessary to specify the associated target entity explicitly since it can usually be inferred from the type of the object being referenced.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OneToOne {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
    String mappedBy() default "";
}
```

Table 14 lists the annotation elements that may be specified for a `OneToOne` annotation.

Table 14 OneToOne Annotation Elements

Type	Name	Description	Default
Class	targetEntity	(Optional) The entity class that is the target of the association.	The type of the property that stores the association.

Type	Name	Description	Default
CascadeType[]	cascade	(Optional) The operations that should be cascaded to the target of the association.	No operations are cascaded.
FetchType	fetch	(Optional) Hint to the implementation as to whether the association should be lazy loaded or eagerly fetched. The EAGER strategy is a requirement on the persistence provider runtime that the associated entity should be eagerly fetched.	EAGER
boolean	optional	(Optional) Whether the association is optional. If set to false then a non-null relationship must always exist.	true
String	mappedBy	(Optional) The field that owns the relationship. The mappedBy element is only specified on the inverse (non-owning) side of the association.	

Example: One-to-one association that maps a foreign key column.

On Customer class:

```
@OneToOne(optional=false)
@JoinColumn(
    name="CUSTREC_ID", unique=true, nullable=false, updatable=false)
public CustomerRecord getCustomerRecord() { return customerRecord; }
```

On CustomerRecord class:

```
@OneToOne(optional=false, mappedBy="customerRecord")
public Customer getCustomer() { return customer; }
```

Example: One-to-one association that assumes both the source and target share the same primary key values.

On Employee class:

```
@Entity(access=FIELD)
public class Employee {
    @Id Integer id;

    @OneToOne @PrimaryKeyJoinColumn
    EmployeeInfo info;
    ...
}
```

On EmployeeInfo class:

```
@Entity(access=FIELD)
public class EmployeeInfo {
    @Id Integer id;
    ...
}
```

9.1.19 OneToMany Annotation

A `OneToMany` annotation defines a many-valued association with one-to-many multiplicity.

If the Collection is defined using generics to specify the element type then the associated target entity type need not be specified; otherwise the target entity class must be specified.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OneToMany {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default LAZY;
    String mappedBy() default "";
}
```

Table 15 lists the annotation elements that may be specified for a `OneToMany` annotation.

Table 15 OneToMany Annotation Elements

Type	Name	Description	Default
Class	targetEntity	(Optional) The entity class that is the target of the association. Optional only if the Collection property is defined using Java generics. Must be specified otherwise.	The parameter type of the Collection when defined using generics.
CascadeType[]	cascade	(Optional) The operations that should be cascaded to the target of the association.	No operations are cascaded.
FetchType	fetch	(Optional) Whether the association should be lazy loaded or eagerly fetched. The EAGER strategy is a requirement on the persistence provider runtime that the associated entities should be eagerly fetched.	LAZY
String	mappedBy	The field that owns the relationship. Required unless the relationship is unidirectional.	

The default schema-level mapping for unidirectional one-to-many relationships uses a join table, as described in Section 2.1.8.5. Unidirectional one-to-many relationships may be implemented using one-to-many foreign key mappings, however, such support is not required in this release. Applications that want to use a foreign key mapping strategy for one-to-many relationships should make these relationships bidirectional to ensure portability.

Example 1 : One-to-Many association using generics

In Customer class:

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

In Order class:

```
@ManyToOne
@JoinColumn(name="CUST_ID", nullable=false)
public Customer getCustomer() { return customer; }
```

Example 2: One-to-Many association without using generics

In Customer class:

```
@OneToMany(targetEntity=com.acme.Order.class, cascade=ALL,
mappedBy=”customer”)
public Set getOrders() { return orders; }
```

In Order class:

```
@ManyToOne
@JoinColumn(name="CUST_ID", nullable=false)
public Customer getCustomer() { return customer; }
```

9.1.20 JoinTable Annotation

A `JoinTable` annotation is specified on the owning side of a many-to-many association. If the `JoinTable` annotation is missing, the default values of the annotation elements apply.

The name of the join table is assumed to be the table names of the associated primary tables concatenated together (owning side first) using an underscore.

```
@Target({METHOD, FIELD})
public @interface JoinTable {
    Table table() default @Table(specified=false);
    JoinColumn[] joinColumns() default {};
    JoinColumn[] inverseJoinColumns() default {};
}
```

Table 16 lists the annotation elements that may be specified for a `JoinTable` annotation.

Table 16 JoinTable Annotation Elements

Type	Name	Description	Default
Table	table	(Optional) The table definition for the join table.	A default Table definition, having as its name the concatenated names of the two associated entity primary tables, separated by an underscore.

Type	Name	Description	Default
JoinColumn[]	joinColumns	(Optional) Define the foreign key columns of the join table which reference the primary table of the entity owning the association (i.e. the owning side of the association).	The primary key columns of the entity and the foreign key columns in the join table are assumed to have the same names.
JoinColumn[]	inverseJoinColumns	(Optional) Define the foreign key columns of the join table which reference the primary table of the entity that does not own the association (i.e. the inverse side of the association).	The primary key columns of the entity and the foreign key columns in the join table are assumed to have the same names.

Example:

```
@JoinTable(
    table=@Table(name=CUST_PHONE),
    joinColumns=
         @JoinColumn(name="CUST_ID", referencedColumnName="ID"),
    inverseJoinColumns=
         @JoinColumn(name="PHONE_ID", referencedColumnName="ID")
)
```

9.1.21 ManyToMany Annotation

A `ManyToMany` annotation defines a many-valued association with many-to-many multiplicity. If the Collection is defined using generics to specify the element type, the associated target entity class does not need to be specified; otherwise it must be specified.

Every many-to-many association has two sides, the owning side and the non-owning or inverse side. The join table is specified on the owning side. If the association is bidirectional, either side may be designated as the owning side.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface ManyToMany {
     Class targetEntity() default void.class;
     CascadeType[] cascade() default {};
     FetchType fetch() default LAZY;
     String mappedBy() default "";
}
```

The annotation elements listed in Table 15 apply for a `ManyToMany` annotation.

Example 1:

In Customer class:

```
@ManyToMany(cascade=PERSIST)
@JoinTable(table=@Table(name="CUST_PHONES"))
public Set<PhoneNumber> getPhones() { return phones; }
```

In PhoneNumber class:

```
@ManyToMany(cascade=PERSIST, mappedBy="phones")
public Set<Customer> getCustomers() { return customers; }
```

Example 2:

In Customer class:

```
@ManyToMany(targetEntity=com.acme.PhoneNumber.class, cascade=PERSIST)
public Set getPhones() { return phones; }
```

In PhoneNumber class:

```
@ManyToMany(targetEntity=com.acme.Customer.class, cascade=PERSIST,
mappedBy="phones")
public Set getCustomers() { return customers; }
```

Example 3:

In Customer class:

```
@ManyToMany(cascade=PERSIST)
@JoinTable(
    table=@Table(name=CUST_PHONE),
    joinColumns=
         @JoinColumn(name="CUST_ID", referencedColumnName="ID"),
    inverseJoinColumns=
         @JoinColumn(name="PHONE_ID", referencedColumnName="ID")
)
public Set<PhoneNumber> getPhones() { return phones; }
```

In PhoneNumberClass:

```
@ManyToMany(cascade=PERSIST, mappedBy="phones")
public Set<Customer> getCustomers() { return customers; }
```

9.1.22 MapKey Annotation

The MapKey annotation is used to specify the map key for associations of type `java.util.Map`.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface MapKey {
  String name() default "";
}
```

The `name` element designates the name of the persistent field or property of the associated entity that is used as the map key. If `name` is not specified, by default the primary key is used as the map key.

If the primary key is a composite primary key and is mapped as `IdClass`, an instance of the primary key class is used as the key.

Example 1:

```
@Entity
public class Department {
       ...
       @OneToMany(mappedBy="department")
       @MapKey(name="empId")
       public Map<Integer, Employee> getEmployees() {... }
       ...
}

@Entity
public class Employee {
       private empId;
       @Id Integer getEmpid() { return empId; }

       @ManyToOne
       @JoinColumn(name="dept_id")
       public Department getDepartment() { ... }
       ...
}
```

Example 2:

```
@Entity
public class Department {
     ...
     @OneToMany(mappedBy="department")
     @MapKey(name="empPK")
     public Map<EmployeePK, Employee> getEmployees() {... }
     ...
 }
@Entity(access=FIELD)
public class Employee {
     @EmbeddedId EmployeePK empPK;
     ...
     @ManyToOne
     @JoinColumn(name="dept_id")
     public Department getDepartment() { ... }
     ...

 }

@Embeddable(access=FIELD)
public class EmployeePK {
    String name;
    Date bday;
}
```

9.1.23 OrderBy Annotation

The `OrderBy` annotation specifies the ordering of the elements of a collection valued association at the point when the association is retrieved.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OrderBy {
  String value() default "";
}
```

The syntax of the ordering element is an *orderby_list*, as follows:

*orderby_list::= orderby_item [,orderby_item]**
orderby_item::= property_or_field_name [ASC | DESC]

If `ASC` or `DESC` is not specified, `ASC` (ascending order) is assumed.

If the ordering element is not specified, ordering by the primary key is assumed.

The property or field name must correspond to that of a persistent property or field of the associated class. The properties or fields used in the ordering must correspond to columns for which comparison operators are supported.

Example:

```
@Entity public class Course {
 ...
 @ManyToMany
 @OrderBy("lastname ASC")
 public List<Student> getStudents() {...};
 ...
}

@Entity public class Student {
  ...
  @ManyToMany(mappedBy="students")
  @OrderBy // PK is assumed
  public Set<Course> getCourses() {...};
  ...
}
```

9.1.24 Inheritance Annotation

The Inheritance annotation defines the inheritance strategy to be used for an entity class hierarchy.

The three inheritance mapping strategies are the single table per class hierarchy, table per class, and joined subclass strategies. See Section 2.1.10 for a more detailed discussion of inheritance strategies. The inheritance strategy options are defined by the `InheritanceType` enum:

```
public enum InheritanceType
    { SINGLE_TABLE, TABLE_PER_CLASS, JOINED };
```

Support for the TABLE_PER_CLASS and JOINED mapping strategies is optional in this release, but will be required in the next.

For the SINGLE_TABLE mapping strategy, and potentially also for the JOINED strategy, the persistence provider will use a type discriminator column. The supported discriminator types are defined by the `DiscriminatorType` enum:

```
public enum DiscriminatorType { STRING, CHAR, INTEGER };
```

The strategy and the `discriminatorType` are only specified once per class hierarchy (in the root class), while the `discriminatorValue` should be specified for each class in the hierarchy.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Inheritance {
    InheritanceType strategy() default SINGLE_TABLE;
    DiscriminatorType discriminatorType() default STRING;
    String discriminatorValue() default "";
}
```

If no inheritance type is specified for a class hierarchy, the single table per class hierarchy strategy is used.

Table 17 lists the annotation elements that may be specified for a `Inheritance` annotation.

Table 17 Inheritance Annotation Elements

Type	Name	Description	Default
InheritanceType	strategy	(Optional) The table strategy to use to store the entity inheritance hierarchy.	InheritanceType.SINGLE_TABLE
DiscriminatorType	discriminatorType	(Optional) The type of object/column to use as a class discriminator.	DiscriminatorType.STRING
String	discriminatorValue	(Optional) The value that indicates that the row is an entity of the annotated entity type.	Provider-specific function to generate a String representing the entity class

Example:

```
@Entity
@Table(name="CUST")
@Inheritance(strategy=SINGLE_TABLE,
     discriminatorType=STRING,
     discriminatorValue="CUST")
public class Customer { ... }

@Entity
@Inheritance(discriminatorValue="VCUST")
public class ValuedCustomer extends Customer { ... }
```

9.1.25 PrimaryKeyJoinColumn Annotation

The `PrimaryKeyJoinColumn` annotation specifies the primary key columns that are used as a foreign key to join to another table. The `PrimaryKeyJoinColumn` annotation is used to join the primary table of an entity subclass in the JOINED mapping strategy to the primary table of its superclass; together with a `SecondaryTable` annotation to join a secondary table to a primary table; or in a `OneToOne` mapping in which the primary key of the referencing entity is used as a foreign key to the referenced entity.

If no `PrimaryKeyJoinColumn` annotation is specified for a subclass in the JOINED mapping strategy, the foreign key columns are assumed to have the same names as the primary key columns of the primary table of the superclass.

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface PrimaryKeyJoinColumn {
    String name() default "";
    String referencedColumnName() default "";
    String columnDefinition() default "";
}
```

Table 18 lists the annotation elements that may be specified for a `PrimaryKeyJoinColumn` annotation.

Table 18 PrimaryKeyJoinColumn Annotation Elements

Type	Name	Description	Default
String	name	The name of the primary key column of the current table.	The same name as the primary key column of the table for the referencing entity.
String	referencedColumnName	(Optional) The name of the primary key column of the table being joined to.	The same name as the primary key column of the table for the referenced entity.
String	columnDefinition	(Optional) The SQL fragment that is used when generating the DDL for the column. This should not be specified for a OneToOne primary key association.	Generated SQL to create a column of the inferred type.

Example: Customer and ValuedCustomer subclass

```
@Entity
@Table(name="CUST")
@Inheritance(strategy=JOINED,
    discriminatorType=STRING,
    discriminatorValue="CUST")
public class Customer { ... }

@Entity
@Table(name="VCUST")
@Inheritance(discriminatorValue="VCUST")
@PrimaryKeyJoinColumn(name="CUST_ID")
public class ValuedCustomer extends Customer { ... }
```

9.1.26 PrimaryKeyJoinColumns Annotation

Composite keys are supported via the `PrimaryKeyJoinColumns` annotation.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface PrimaryKeyJoinColumns {
PrimaryKeyJoinColumn[] value();
}
```

Example: ValuedCustomer subclass

```
@Entity
@Table(name="VCUST")
@Inheritance(discriminatorValue="VCUST")
@PrimaryKeyJoinColumns({
    @PrimaryKeyJoinColumn(name="CUST_ID",
        referencedColumnName="ID"),
    @PrimaryKeyJoinColumn(name="CUST_TYPE",
        referencedColumnName="TYPE")
})
public class ValuedCustomer extends Customer { ... }
```

Example: OneToOne relationship between Employee and EmployeeInfo classes

```
public class EmpPK {
    public Integer id;
    public String name;
}

@Entity(access=FIELD)
@IdClass(com.acme.EmpPK.class)
public class Employee {

    @Id Integer id;
    @Id String name;

    @OneToOne
    @PrimaryKeyJoinColumns({
        @PrimaryKeyJoinColumn(name="ID", referencedColumn-
Name="EMP_ID"),
        @PrimaryKeyJoinColumn(name="NAME", referencedColumn-
Name="EMP_NAME")})
    EmployeeInfo info;

    ...
}

@Entity(access=FIELD)
@IdClass(com.acme.EmpPK.class)
public class EmployeeInfo {

    @Id @Column(name="EMP_ID")
    Integer id;
    @Id @Column(name="EMP_NAME")
    String name;

    ...
}
```

9.1.27 DiscriminatorColumn Annotation

The `DiscriminatorColumn` annotation is used to define the discriminator column for SINGLE_TABLE and JOINED mapping strategies.

```
@Target({TYPE}) @Retention(RUNTIME)
    public @interface DiscriminatorColumn {
    String name() default "";
    String columnDefinition() default "";
    int length() default 10;
}
```

If the `DiscriminatorColumn` annotation is missing, and a discriminator column is required, the name of the discriminator column defaults to "TYPE".

Table 19 lists the annotation elements that may be specified for a `DiscriminatorColumn` annotation.

Table 19 DiscriminatorColumn Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of column to be used for the discriminator.	“TYPE”
String	columnDefinition	(Optional) The SQL fragment that is used when generating the DDL for the discriminator column.	Provider-generated SQL to create a column of the specified discriminator type.
String	length	(Optional) The column length for String-based discriminator types. Ignored for other discriminator types.	10

Example:

```
@Entity
@Table(name="CUST")
@Inheritance(strategy=SINGLE_TABLE,
    discriminatorType=STRING,
    discriminatorValue="CUSTOMER")
@DiscriminatorColumn(name="DISC", length=20)
public class Customer { ... }
```

9.1.28 Embeddable Annotation

The `Embeddable` annotation is used to mark an object that is stored as an intrinsic part of an owning entity and shares the identity of that entity. Each of the persistent properties or fields of the embedded object is mapped to the database table. Only `Basic`, `Column`, and `Lob` mapping annotations may be used to map embedded objects.

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Embeddable {
    AccessType access() default PROPERTY;
}
```

Table 20 lists the annotation elements that may be specified for an `Embeddable` annotation.

Table 20 Embeddable Annotation Elements

Type	Name	Description	Default
AccessType	access	(Optional) Specifies how the persistence provider runtime accesses the persistent attributes of the embedded object, either through its properties or its fields.	PROPERTY

Example:

```
@Embeddable(access=FIELD)
public class EmploymentPeriod {
    java.util.Date startDate;
    java.util.Date endDate;
    ...
}
```

9.1.29 Embedded Annotation

The `Embedded` annotation may be used in an entity class when it is using a shared embeddable class. The entity may override the column mappings declared within the embeddable class to apply to its own entity table.

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Embedded {}
}
```

Example:

```
@Embedded
@AttributeOverrides({
    @AttributeOverride(name="startDate", column=@Col-
umn("EMP_START")),
    @AttributeOverride(name="endDate", column=@Column("EMP_END"))
})
public EmploymentPeriod getEmploymentPeriod() { ... }
```

9.1.30 EmbeddableSuperclass Annotation

The `EmbeddableSuperclass` annotation designates an embedded superclass.

A class designated as an embeddable superclass has no separate table defined for it. Its mapping information is applied to the entities that inherit from it.

A class designated as `EmbeddableSuperclass` can be mapped in the same way as an entity except that the mappings will apply only to its subclasses since no table exists for the embeddable superclass. When applied to the subclasses the inherited mappings will apply in the context of the subclass tables. Mapping information may be overridden in such subclasses by using the `AttributeOverride` annotation.

```
@Target(TYPE) @Retention(RUNTIME)
public @interface EmbeddableSuperclass {
  AccessType access() default PROPERTY;
 }
```

The semantics of the `access` element are the same as for the Entity annotation.

9.1.31 SequenceGenerator Annotation

The `SequenceGenerator` annotation defines a primary key or id generator which may be referenced by name when annotating the id attribute (see `@Id` annotation). A generator may be defined at either the class, method, or field level. The level at which it is defined will depend upon the desired visibility and sharing of the generator. No scoping or visibility rules are actually enforced. However, it is good practice to define the generator at the level for which it will be used.

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface SequenceGenerator {
    String name();
    String sequenceName() default "";
    int initialValue() default 0;
    int allocationSize() default 50;
}
```

Table 21 lists the annotation elements that may be specified for a `SequenceGenerator` annotation.

Table 21 SequenceGenerator Annotation Elements

Type	Name	Description	Default
String	name	(Required) A unique name for the generator that can be referenced by one or more classes to be the generator for ids.	
String	sequenceName	(Optional) The name of the database sequence object to obtain ids from.	A provider-chosen value
int	initialValue	(Optional) The value to set the sequence object to start generating from once it has been created.	0
int	allocationSize	(Optional) The amount to increment by when allocating sequence numbers from the sequence.	50

Example:

```
@SequenceGenerator(name="EMP_SEQ", allocationSize=25)
```

9.1.32 TableGenerator Annotation

The `TableGenerator` annotation defines a primary key or id generator which may be referenced by name when annotating the id attribute (see `@Id` annotation). A generator may be defined at either the class, method, or field level. The level at which it is defined will depend upon the desired visibility and sharing of the generator. No scoping or visibility rules are actually enforced. However, it is good practice to define the generator at the level for which it will be used.

The table is used by the persistence provider to store generated id values for entities. An entity type will typically use its own row in the table to generate the id values for that entity class. The id values are normally positive integers.

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface TableGenerator {
     String name();
     Table table() default @Table(specified=false);
     String pkColumnName() default "";
     String valueColumnName() default "";
     String pkColumnValue() default "";
     int initialValue() default 0;
     int allocationSize() default 50;
}
```

Table 22 lists the annotation elements that may be specified for a `TableGenerator` annotation.

Table 22 TableGenerator Annotation Elements

Type	Name	Description	Default
String	name	(Required) A unique name for the generator that can be referenced by one or more classes to be the generator for ids.	
Table	table	(Optional) Table that stores the generated ids. Full table annotation that may be used when a table definition is required.	Name is chosen by persistence provider
String	pkColumnName	(Optional) Name of the primary key column in the table.	A provider-chosen name
String	valueColumnName	(Optional) Name of the column that stores the last value generated.	A provider-chosen name
String	pkColumnValue	(Optional) The primary key value in the generator table that distinguishes this set of generated values from others that may be stored in the table.	A provider-chosen value to store in the primary key column of the generator table
int	initialValue	(Optional) The initial value to be used when allocating id numbers from the generator.	0
int	allocationSize	(Optional) The amount to increment by when allocating id numbers from the generator.	50

Examples:

```
@Entity public class Employee {
    ...
    @TableGenerator(name="empGen",
        table=@Table(name="ID_GEN"),
        pkColumnName="GEN_KEY",
        valueColumnName="GEN_VALUE",
        pkColumnValue="EMP_ID",
        allocationSize=1)
    @Id(generate=TABLE, generator="empGen")
    public int id;
    ...
}

@Entity public class Address {
    ...
    @TableGenerator(name="addressGen",
        table=@Table(name="ID_GEN"),
        pkColumnValue="ADDR_ID")
    @Id(generate=TABLE, generator="addressGen")
    public int id;
    ...
}
```

9.2 Examples of the Application of Annotations for Object/Relational Mapping

9.2.1 Examples of Simple Mappings

```
@Entity(access=FIELD)
public class Customer {

    @Id(generate=AUTO) Long id;
    @Version protected int version;
    @ManyToOne Address address;
    @Basic String description;
    @OneToMany(targetEntity=com.acme.Order.class,
               mappedBy="customer")
    Collection orders = new Vector();
    @ManyToMany(mappedBy="customers")
    Set<DeliveryService> serviceOptions = new HashSet();

    public Customer() {}

    public Long getId() { return id; }

    public Address getAddress() { return address; }
    public void setAddress(Address addr) {
        this.address = addr;
    }

    public String getDescription() { return description; }
    public void setDescription(String desc) {
        this.description = desc;
    }

    public Collection getOrders() { return orders; }

    public Set<DeliveryService> getServiceOptions() {
        return serviceOptions;
    }
}


@Entity
public class Address {

    private Long id;
    private int version;
    private String street;

    public Address() {}

    @Id(generate=AUTO)
    public Long getId() { return id; }
    protected void setId(Long id) { this.id = id; }

    @Version
    public int getVersion() { return version; }
```

```
    protected void setVersion(int version) {
        this.version = version;
    }

    public String getStreet() { return street; }
    public void setStreet(String street) {
        this.street = street;
    }
}


@Entity
public class Order {

    private Long id;
    private int version;
    private String itemName;
    private int quantity;
    private Customer cust;

    public Order() {}

    @Id(generate=AUTO)
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    @Version
    protected int getVersion() { return version; }
    protected void setVersion(int version) {
        this.version = version;
    }

    public String getItemName() { return itemName; }
    public void setItemName(String itemName) {
        this.itemName = itemName;
    }

    public int getQuantity() { return quantity; }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    @ManyToOne
    public Customer getCustomer() { return cust; }
    public setCustomer(Customer cust) {
        this.cust = cust;
    }
}


@Entity
@Table(name="DLVY_SVC")
public class DeliveryService {

    private String serviceName;
    private int priceCategory;
    private Collection customers;

    public DeliveryService() {}
```

```
    @Id
    public String getServiceName() { return serviceName; }
    public void setServiceName(String serviceName) {
        this.serviceName = serviceName;
    }

    public int getPriceCategory() { return priceCategory; }
    public void setPriceCategory(int priceCategory) {
        this.priceCategory = priceCategory;
    }

    @ManyToMany(targetEntity=com.acme.Customer.class)
    @JoinTable(table=@Table(name="CUST_DLVRY"))
    public Collection getCustomers() { return customers; }
    public setCustomers(Collection customers) {
        this.customers = customers;
    }
}
```

9.2.2 A More Complex Example

```
/***** Employee class *****/

@Entity
@Table(name="EMPL")
@SecondaryTable(name="EMP_SALARY",
    pkJoin=@PrimaryKeyJoinColumn(name="EMP_ID",
                     referencedColumnName="ID"))
public class Employee implements Serializable {

    private Long id;
    private int version;
    private String name;
    private Address address;
    private Collection phoneNumbers;
    private Collection<Project> projects;
    private Long salary;
    private EmploymentPeriod period;

    public Employee() {}

    @Id(generate=TABLE)
    public Integer getId() { return id; }
    protected void setId(Integer id) { this.id = id; }

    @Version
    @Column(name="EMP_VERSION", nullable=false)
    public int getVersion() { return version; }
    protected void setVersion(int version) {
        this.version = version;
    }

    @Column(name="EMP_NAME", length=80)
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    @ManyToOne(cascade=PERSIST, optional=false)
    @JoinColumn(name="ADDR_ID",
            referencedColumnName="ID", nullable=false)
    public Address getAddress() { return address; }
    public void setAddress(Address address) {
        this.address = address;
    }

    @OneToMany(targetEntity=com.acme.PhoneNumber.class,
           cascade=ALL, mappedBy="employee")
    public Collection getPhoneNumbers() { return phoneNumbers; }
    public void setPhoneNumbers(Collection phoneNumbers) {
        this.phoneNumbers = phoneNumbers;
    }

    @ManyToMany(mappedBy="employee", cascade=PERSIST)
    @JoinTable(table=@Table(name="EMP_PROJ"),
        joinColumns=@JoinColumn(
            name="EMP_ID", referencedColumnName="ID"),
        inverseJoinColumns=@JoinColumn(
            name="PROJ_ID", referencedColumnName="ID"))
    public Collection<Project> getProjects() { return projects; }
```

```
    public void setProjects(Collection<Project> projects) {
        this.projects = projects;
    }

    @Column(name="EMP_SAL", secondaryTable="EMP_SALARY")
    public Long getSalary() { return salary; }
    public void setSalary(Long salary) {
        this.salary = salary;
    }

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="startDate",
            column=@Column(name="EMP_START")),
        @AttributeOverride(name="endDate",
            column=@Column(name="EMP_END"))
    })
    public EmploymentPeriod getEmploymentPeriod() {
        return period;
    }
    public void setEmploymentPeriod(EmploymentPeriod period) {
        this.period = period;
    }
}


/***** Address class *****/

@Entity
public class Address implements Serializable {

    private Integer id;
    private int version;
    private String street;
    private String city;

    public Address() {}

    @Id(generate=IDENTITY)
    public Integer getId() { return id; }
    protected void setId(Integer id) { this.id = id; }

    @Version @Column("VERS", nullable=false)
    public int getVersion() { return version; }
    protected void setVersion(int version) {
        this.version = version;
    }

    @Column(name="RUE")
    public String getStreet() { return street; }
    public void setStreet(String street) {
        this.street = street;
    }

    @Column(name="VILLE")
    public String getCity() { return city; }
    public void setCity(String city) { this.city = city; }
}
```

```
/***** PhoneNumber class *****/

@Entity
@Table(name="PHONE")
public class PhoneNumber implements Serializable {

    private String number;
    private int phoneType;
    private Employee employee;

    public PhoneNumber() {}

    @Id
    public String getNumber() { return number; }
    public void setNumber(String number) {
        this.number = number;
    }

    @Column(name="PTYPE")
    public int getPhonetype() { return phonetype; }
    public void setPhoneType(int phoneType) {
        this.phoneType = phoneType;
    }

    @ManyToOne(optional=false)
    @JoinColumn(name="EMP_ID", referencedColumnName="ID")
    public Employee getEmployee() { return employee; }
    public void setEmployee(Employee employee) {
        this.employee = employee;
    }
}


/***** Project class *****/

@Entity
@Inheritance(strategy=JOINED,
    discriminatorType=STRING,
    discriminatorValue="Proj")
@DiscriminatorColumn(name="DISC")
public class Project implements Serializable {

    private Integer projId;
    private int version;
    private String name;
    private Set<Employee> employees;

    public Project() {}

    @Id(generate=TABLE)
    public Integer getId() { return projId; }
    protected void setId(Integer id) { this.projId = id; }

    @Version
    public int getVersion() { return version; }
    protected void setVersion(int version) { this.version = version; }

    @Column(name="PROJ_NAME")
    public String getName() { return name; }
```

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

    @ManyToMany(mappedBy="projects")
    public Set<Employee> getEmployees() { return employees; }
    public void setEmployees(Set<Employee> employees) {
        this.employees = employees;
    }
}


/***** GovernmentProject subclass *****/

@Entity
@Table(name="GOVT_PROJECT")
@Inheritance(discriminatorValue="GovtProj")
@PrimaryKeyJoinColumn(name="GOV_PROJ_ID",
                      referencedColumnName="ID")
public class GovernmentProject extends Project {

    private String fileInfo;

    public GovernmentProject() { super(); }

    @Column("INFO")
    public String getFileInfo() { return fileInfo; }
    public void setFileInfo(String fileInfo) {
        this.fileInfo = fileInfo;
    }
}


/***** CovertProject subclass *****/

@Entity
@Table(name="C_PROJECT")
@Inheritance(discriminatorValue="CovProj")
@PrimaryKeyJoinColumn(name="COV_PROJ_ID",
                      referencedColumnName="ID")
public class CovertProject extends Project {

    private String classified;

    public CovertProject(String classified) {
        super();
        this.classified = classified;
    }

    @Column(updatable=false)
    public String getClassified() { return classified; }
    protected void setClassified(String classified) {
        this.classified = classified;
    }
}


/***** EmploymentPeriod class *****/

@Embeddable
public class EmploymentPeriod implements Serializable {
```

```
    private Date start;
    private Date end;

    public EmploymentPeriod() {}

    @Column(nullable=false)
    public Date getStartDate() { return start; }
    public void setStartDate(Date start) {
        this.start = start;
    }

    public Date getEndDate() { return end; }
    public void setEndDate(Date end) {
       this.end = end;
    }
}
```

Chapter 10 XML Descriptor

The XML descriptor is intended to serve as both an alternative and an overriding mechanism to the use of Java language metadata annotations.

10.1 XML Schema

This section provides the XML schema for use with the persistence API.

This is currently work in progress. The contents of this chapter are under discussion in the Expert Group and are undergoing development and change. We present this information as illustrative of a XML descriptor alternative and overriding mechanism to the use of Java language metadata annotations for object/relational mapping. The intention is for this schema to parallel the use of annotations in functionality.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://java.sun.com/xml/ns/persistence_ORM"
  targetNamespace="http://java.sun.com/xml/ns/persistence_ORM"
  elementFormDefault="qualified"
  version="1.0">



  <!--
    Top-level element defines entity and embeddable mappings, named queries
    and named id generators. Defines a default package for classnames
    in this mapping.
  -->

  <xsd:element name="entity-mappings">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="package" type="xsd:string" minOccurs="0"/>
        <xsd:element name="default-access" type="access-type"
                     default="PROPERTY" minOccurs="0"/>
        <xsd:element name="default-cascade" type="cascade-type"
                     minOccurs="0"/>

        <xsd:element name="embeddable-superclass" type="embeddable"
                     maxOccurs="unbounded" minOccurs="0"/>
        <xsd:element name="entity" type="entity"
                     maxOccurs="unbounded" minOccurs="0"/>
        <xsd:element name="embeddable" type="embeddable"
                     maxOccurs="unbounded" minOccurs="0"/>

        <xsd:element name="query" type="query"
                     maxOccurs="unbounded" minOccurs="0"/>
        <xsd:element name="native-query" type="native-query"
                     maxOccurs="unbounded" minOccurs="0"/>

        <xsd:element name="sequence-generator" type="sequence-generator"
                     maxOccurs="unbounded" minOccurs="0"/>
        <xsd:element name="table-generator" type="table-generator"
                     maxOccurs="unbounded" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>



  <!--
    Defines table, inheritance, attribute and association mappings
    for an entity class.
  -->

  <xsd:complexType name="entity">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" minOccurs="0"/>
      <xsd:element name="class" type="xsd:string"/>
      <xsd:element name="access" type="access-type" default="PROPERTY"
                   minOccurs="0"/>
      <xsd:element name="inheritance-strategy" type="inheritance-type"
```

```
                    minOccurs="0"/>

      <xsd:element name="table" type="table" minOccurs="0"/>
      <xsd:element name="secondary-table" type="secondary-table"
                   maxOccurs="unbounded" minOccurs="0"/>

      <xsd:choice>
        <xsd:sequence>
          <xsd:element name="discriminator-column" type="discriminator-col-
umn"
                       minOccurs="0"/>
          <xsd:element name="discriminator-type" type="discriminator-type"
                       minOccurs="0"/>
          <xsd:element name="discriminator-value" type="xsd:string"
                       minOccurs="0"/>
        </xsd:sequence>
        <xsd:element name="primary-key-join-column"
                     type="primary-key-join-column"
                     minOccurs="0" maxOccurs="unbounded"/>

      </xsd:choice>

      <xsd:choice>
        <xsd:element name="embedded-id" type="xsd:string"/>
        <xsd:sequence>
          <xsd:element name="id-class" type="xsd:string" minOccurs="0"/>
          <xsd:element name="id" type="id"
                       maxOccurs="unbounded" minOccurs="0"/>
        </xsd:sequence>
      </xsd:choice>

      <xsd:element name="version" type="version" minOccurs="0"/>

      <xsd:element name="basic" type="basic"
                   maxOccurs="unbounded" minOccurs="0"/>
      <xsd:element name="lob" type="lob"
                   maxOccurs="unbounded" minOccurs="0"/>

      <xsd:element name="embedded" type="embedded"
                   maxOccurs="unbounded" minOccurs="0"/>

      <xsd:element name="one-to-one" type="many-to-one"
                   maxOccurs="unbounded" minOccurs="0"/>
      <xsd:element name="many-to-one" type="many-to-one"
                   maxOccurs="unbounded" minOccurs="0"/>
      <xsd:element name="one-to-many" type="one-to-many"
                   maxOccurs="unbounded" minOccurs="0"/>
      <xsd:element name="many-to-many" type="many-to-many"
                   maxOccurs="unbounded" minOccurs="0"/>

      <xsd:element name="transient" type="xsd:string"
                   maxOccurs="unbounded" minOccurs="0"/>

    </xsd:sequence>
  </xsd:complexType>



  <!--
    Declares a secondary table for an entity, and defines attribute
    mappings to columns of that secondary table.
  -->

  <xsd:complexType name="secondary-table">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
```

```
    <xsd:element name="schema" type="xsd:string" minOccurs="0"/>
    <xsd:element name="catalog" type="xsd:string" minOccurs="0"/>

    <xsd:element name="primary-key-join-column"
                 type="primary-key-join-column"
                 minOccurs="0" maxOccurs="unbounded"/>


    <xsd:element name="unique-constraint"
                 minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>



<!--
  Defines default attribute mappings for an embeddable superclass.
-->

<xsd:complexType name="embeddable-superclass">
  <xsd:sequence>
    <xsd:element name="class" type="xsd:string"/>
    <xsd:element name="access" type="access-type" default="PROPERTY"
                 minOccurs="0"/>

    <xsd:choice>
      <xsd:element name="embedded-id" type="xsd:string"/>
      <xsd:sequence>
        <xsd:element name="id-class" type="xsd:string" minOccurs="0"/>
        <xsd:element name="id" type="id"
                     maxOccurs="unbounded" minOccurs="0"/>
      </xsd:sequence>
    </xsd:choice>

    <xsd:element name="version" type="version" minOccurs="0"/>

    <xsd:element name="basic" type="basic"
                 maxOccurs="unbounded" minOccurs="0"/>
    <xsd:element name="lob" type="lob"
                 maxOccurs="unbounded" minOccurs="0"/>

    <xsd:element name="embedded" type="embedded"
                 maxOccurs="unbounded" minOccurs="0"/>

    <xsd:element name="one-to-one" type="one-to-one"
                 maxOccurs="unbounded" minOccurs="0"/>
    <xsd:element name="many-to-one" type="many-to-one"
                 maxOccurs="unbounded" minOccurs="0"/>
    <xsd:element name="one-to-many" type="one-to-many"
                 maxOccurs="unbounded" minOccurs="0"/>
    <xsd:element name="many-to-many" type="many-to-many"
                 maxOccurs="unbounded" minOccurs="0"/>

    <xsd:element name="transient" type="xsd:string"
                 maxOccurs="unbounded" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>



<!--
  Defines default attribute mappings for an embeddable class.
-->

<xsd:complexType name="embeddable">
  <xsd:sequence>
```

```
      <xsd:element name="class" type="xsd:string"/>
      <xsd:element name="access" type="access-type" default="PROPERTY"
                   minOccurs="0"/>

      <xsd:element name="basic" type="basic"
                   maxOccurs="unbounded" minOccurs="0"/>
      <xsd:element name="lob" type="lob"
                   maxOccurs="unbounded" minOccurs="0"/>

      <xsd:element name="transient" type="xsd:string"
                   maxOccurs="unbounded" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>



  <!--
    Declares a primary key attribute and, optionally, a generation strategy.
  -->

  <xsd:complexType name="id">
    <xsd:sequence>
      <xsd:element name="attribute" type="xsd:string"/>
      <xsd:element name="generate" type="generator-type" minOccurs="0"/>
      <xsd:element name="generator" type="xsd:string" minOccurs="0" />
      <xsd:element name="column" type="column" minOccurs="0" />
    </xsd:sequence>
  </xsd:complexType>



  <!--
    Declares a version attribute.
  -->

  <xsd:complexType name="version">
    <xsd:sequence>
      <xsd:element name="attribute" type="xsd:string"/>
      <xsd:element name="column" type="column" minOccurs="0" />
    </xsd:sequence>
  </xsd:complexType>



  <!--
    Declares a mapping for a basic attribute.
  -->

  <xsd:complexType name="basic">
    <xsd:sequence>
      <xsd:element name="attribute" type="xsd:string"/>
      <xsd:element name="fetch" type="fetch-type" default="EAGER"
                   minOccurs="0" />
      <xsd:element name="optional" type="xsd:boolean" default="true"
                   minOccurs="0" />
     <xsd:element name="temporal-type" type="temporal-type" minOccurs="0" />
      <xsd:element name="column" type="column" minOccurs="0" />
    </xsd:sequence>
  </xsd:complexType>



  <!--
    Declares a mapping for a large object attribute.
  -->
```

```
<xsd:complexType name="lob">
  <xsd:sequence>
    <xsd:element name="attribute" type="xsd:string"/>
    <xsd:element name="fetch" type="fetch-type" default="LAZY"
                 minOccurs="0" />
    <xsd:element name="optional" type="xsd:boolean" default="true"
                 minOccurs="0" />
    <xsd:element name="lob-type" type="lob-type" default="BLOB"
                 minOccurs="0" />
    <xsd:element name="column" type="column" minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>



<!--
  Declares an attribute of an embeddable type, and allows overriding and
  addition of attribute mappings.
-->

<xsd:complexType name="embedded">
  <xsd:sequence>
    <xsd:element name="attribute" type="xsd:string"/>

    <xsd:element name="basic" type="basic"
                 maxOccurs="unbounded" minOccurs="0"/>
    <xsd:element name="lob" type="lob"
                 maxOccurs="unbounded" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>



<!--
  Declares a one-to-one association mapping to a foreign key column,
  a primary key column, or an inverse one-to-one association.
-->

<xsd:complexType name="one-to-one">
  <xsd:sequence>
    <xsd:element name="attribute" type="xsd:string"/>
    <xsd:element name="target-entity" type="xsd:string" minOccurs="0"/>
    <xsd:element name="cascade" type="cascade-type" minOccurs="0" />
    <xsd:element name="fetch" type="fetch-type" default="EAGER"
                 minOccurs="0" />
    <xsd:element name="optional" type="xsd:boolean" default="true"
                 minOccurs="0" />
    <xsd:choice>
       <xsd:element name="mapped-by" type="xsd:string" minOccurs="0" />
       <xsd:element name="join-column" type="column"
                    minOccurs="0" maxOccurs="unbounded" />
       <xsd:element name="primary-key-join-column" type="column"
                    minOccurs="0" maxOccurs="unbounded" />
   <!-- xsd:element name="join-table" type="join-table" minOccurs="0" /-->
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>



<!--
  Declares a many-to-one association mapping to a foreign key column.
-->

<xsd:complexType name="many-to-one">
  <xsd:sequence>
```

```
    <xsd:element name="attribute" type="xsd:string"/>
    <xsd:element name="target-entity" type="xsd:string" minOccurs="0"/>
    <xsd:element name="cascade" type="cascade-type" minOccurs="0" />
    <xsd:element name="fetch" type="fetch-type" default="EAGER"
                 minOccurs="0" />
    <xsd:element name="optional" type="xsd:boolean" default="true"
                 minOccurs="0" />
    <xsd:choice>
       <xsd:element name="join-column" type="column"
                    minOccurs="0" maxOccurs="unbounded" />
   <!-- xsd:element name="join-table" type="join-table" minOccurs="0" /-->
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>



<!--
 Declares a one-to-many association mapping to an association join table or
  an inverse many-to-one association.
-->

<xsd:complexType name="one-to-many">
  <xsd:sequence>
    <xsd:element name="attribute" type="xsd:string"/>
    <xsd:element name="target-entity" type="xsd:string" minOccurs="0"/>
    <xsd:element name="cascade" type="cascade-type" minOccurs="0" />
    <xsd:element name="fetch" type="fetch-type" default="LAZY"
                 minOccurs="0" />
    <xsd:element name="map-key" type="xsd:string" minOccurs="0" />
    <xsd:element name="order-by" type="xsd:string" minOccurs="0" />
    <xsd:choice>
      <xsd:element name="mapped-by" type="xsd:string" minOccurs="0" />
      <xsd:element name="join-table" type="join-table" minOccurs="0" />
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>



<!--
  Declares a many-to-many association mapping to a join table or an
  inverse many-to-many join table.
-->

<xsd:complexType name="many-to-many">
  <xsd:sequence>
    <xsd:element name="attribute" type="xsd:string"/>
    <xsd:element name="target-entity" type="xsd:string" minOccurs="0"/>
    <xsd:element name="cascade" type="cascade-type" minOccurs="0" />
    <xsd:element name="fetch" type="fetch-type" default="LAZY"
                 minOccurs="0" />
    <xsd:element name="map-key" type="xsd:string" minOccurs="0" />
    <xsd:element name="order-by" type="xsd:string" minOccurs="0" />
    <xsd:choice>
      <xsd:element name="mapped-by" type="xsd:string" minOccurs="0" />
      <xsd:element name="join-table" type="join-table" minOccurs="0" />
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>



<!--
  Defines a mapped table.
-->
```

```
  <xsd:complexType name="table">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" minOccurs="0"/>
      <xsd:element name="schema" type="xsd:string" minOccurs="0"/>
      <xsd:element name="catalog" type="xsd:string" minOccurs="0"/>
      <xsd:element name="unique-constraint"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <!--
    Defines the type discriminator column for a SINGLE_TABLE mapping strat-
egy.
  -->

  <xsd:complexType name="discriminator-column">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" minOccurs="0" />
      <xsd:element name="length" type="xsd:integer" minOccurs="0" />
      <xsd:element name="column-definition" type="xsd:string"
                   minOccurs="0" />
    </xsd:sequence>
  </xsd:complexType>



  <!--
    Defines a mapped column.
  -->

  <xsd:complexType name="column">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" minOccurs="0" />
      <xsd:element name="secondary-table" type="xsd:string"
                   minOccurs="0" />

      <xsd:element name="unique" type="xsd:boolean" default="false"
                   minOccurs="0" />
      <xsd:element name="nullable" type="xsd:boolean" default="true"
                   minOccurs="0" />

      <xsd:element name="length" type="xsd:integer" minOccurs="0" />
      <xsd:element name="precision" type="xsd:integer" minOccurs="0" />
      <xsd:element name="scale" type="xsd:integer" minOccurs="0" />

      <xsd:element name="insertable" type="xsd:boolean" default="true"
                   minOccurs="0" />
      <xsd:element name="updatable" type="xsd:boolean" default="true"
                   minOccurs="0" />

      <xsd:element name="column-definition" type="xsd:string"
                   minOccurs="0" />

    </xsd:sequence>
  </xsd:complexType>



  <!--
    Defines a join table for an association mapping, and the columns
    used to join to and from that table.
  -->

  <xsd:complexType name="join-table">
    <xsd:sequence>
```

```
    <xsd:element name="name" type="xsd:string" minOccurs="0"/>
    <xsd:element name="schema" type="xsd:string" minOccurs="0"/>
    <xsd:element name="catalog" type="xsd:string" minOccurs="0"/>

    <xsd:element name="inverse-join-column" type="join-column"
                 maxOccurs="unbounded"/>
    <xsd:element name="join-column" type="join-column"
                 maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>



<!--
  Defines a join condition from a named column of the current table
  to the referenced column of another table.
-->

<xsd:complexType name="join-column">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" minOccurs="0" />
    <xsd:element name="referenced-column-name" type="xsd:string"
                 minOccurs="0" />
    <xsd:element name="secondary-table" type="xsd:string" minOccurs="0" />

    <xsd:element name="unique" type="xsd:boolean" default="false"
                 minOccurs="0" />
    <xsd:element name="nullable" type="xsd:boolean" default="true"
                 minOccurs="0" />

    <xsd:element name="insertable" type="xsd:boolean" default="true"
                 minOccurs="0" />
    <xsd:element name="updatable" type="xsd:boolean" default="true"
                 minOccurs="0" />

    <xsd:element name="column-definition" type="xsd:string"
                 minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>



<!--
  Defines the join condition from the primary key column of the
  current table to the primary key column of another table.
-->

<xsd:complexType name="primary-key-join-column">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" minOccurs="0" />
    <xsd:element name="referenced-column-name" type="xsd:string"
                 minOccurs="0" />
    <xsd:element name="column-definition" type="xsd:string"
                 minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>



<!--
  Declares a named native EJB QL query.
-->

<xsd:complexType name="query">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" />
```

```
    <xsd:element name="query-string" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>



<!--
  Declares a named native SQL query.
-->

<xsd:complexType name="native-query">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" />
    <xsd:element name="query-string" type="xsd:string"/>
    <xsd:choice>
      <xsd:element name="result-class" type="xsd:string"/>
      <xsd:element name="result-set-mapping" type="result-set-mapping"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>



<!--
  Defines metadata for a native SQL query result set.
-->

<xsd:complexType name="result-set-mapping">
  <xsd:sequence>
    <xsd:element name="name" minOccurs="0"/>
    <xsd:element name="entity-result" type="entity-result"
                 minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="column-result" type="xsd:string"
                 minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>



<!--
  Maps column aliases of a native SQL query result set to attributes
  of an entity class.
-->

<xsd:complexType name="entity-result">
  <xsd:sequence>
    <xsd:element name="entity-class" type="xsd:string" />
    <xsd:element name="discriminator-column" type="xsd:string"
                 minOccurs="0"/>
    <xsd:element name="field-result" type="field-result"
                 minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>



<!--
  Maps a result set column alias to an attribute name.
-->

<xsd:complexType name="field-result">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" />
    <xsd:element name="column" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

```
<!--
  Specifies a unique constraint.
-->

<xsd:complexType name="unique-constraint">
  <xsd:sequence>
    <xsd:element name="column-name" type="xsd:string"
                 maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>



<!--
  Defines a named table-based id generator.
-->

<xsd:complexType name="table-generator">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="table" type="table" minOccurs="0"/>
    <xsd:element name="pk-column-name" type="xsd:string" minOccurs="0"/>
    <xsd:element name="pk-column-value" type="xsd:string" minOccurs="0"/>
    <xsd:element name="value-column-name" type="xsd:string"
                 minOccurs="0"/>
    <xsd:element name="initial-value" type="xsd:integer" default="0"
                 minOccurs="0"/>
    <xsd:element name="allocation-size" type="xsd:integer" default="50"
                 minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>



<!--
  Defines a named sequence-based id generator.
-->

<xsd:complexType name="sequence-generator">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="sequence-name" type="xsd:string" minOccurs="0"/>
    <xsd:element name="initial-value" type="xsd:integer" default="0"
                 minOccurs="0"/>
    <xsd:element name="allocation-size" type="xsd:integer" default="50"
                 minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>



<!--
  Enumeration of attribute access types.
-->

<xsd:simpleType name="access-type">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="PROPERTY"/>
    <xsd:enumeration value="FIELD"/>
  </xsd:restriction>
</xsd:simpleType>
```

```
<!--
  Enumeration of fetch types.
-->

<xsd:simpleType name="fetch-type">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="LAZY"/>
    <xsd:enumeration value="EAGER"/>
  </xsd:restriction>
</xsd:simpleType>



<!--
  Enumeration of LOB types.
-->

<xsd:simpleType name="lob-type">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="BLOB"/>
    <xsd:enumeration value="CLOB"/>
  </xsd:restriction>
</xsd:simpleType>



<!--
  Enumeration of discriminator column types.
-->

<xsd:simpleType name="discriminator-type">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="INTEGER"/>
    <xsd:enumeration value="STRING"/>
    <xsd:enumeration value="CHARACTER"/>
  </xsd:restriction>
</xsd:simpleType>



<!--
  Enumeration of temporal types.
-->

<xsd:simpleType name="temporal-type">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="DATE"/>
    <xsd:enumeration value="TIME"/>
    <xsd:enumeration value="TIMESTAMP"/>
  </xsd:restriction>
</xsd:simpleType>



<!--
  Enumeration of cascade styles, specified as, e.g.

  <cascade><persist/><merge/></cascade>

-->

<xsd:complexType name="cascade-type">
  <xsd:sequence>
    <!--a list of empty elements -->
    <xsd:element name="all" minOccurs="0"/>
```

```
      <xsd:element name="persist" minOccurs="0"/>
      <xsd:element name="merge" minOccurs="0"/>
      <xsd:element name="remove" minOccurs="0"/>
      <xsd:element name="refresh" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>



  <!--
    Enumeration of inheritance types.
  -->

  <xsd:simpleType name="inheritance-type">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="SINGLE_TABLE"/>
      <xsd:enumeration value="JOINED"/>
      <xsd:enumeration value="TABLE_PER_CLASS"/>
    </xsd:restriction>
  </xsd:simpleType>



  <!--
    Enumeration of generator types.
  -->

  <xsd:simpleType name="generator-type">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="TABLE"/>
      <xsd:enumeration value="SEQUENCE"/>
      <xsd:enumeration value="IDENTITY"/>
      <xsd:enumeration value="AUTO"/>
    </xsd:restriction>
  </xsd:simpleType>


</xsd:schema>
```

Chapter 11 Related Documents

[1] Enterprise JavaBeans, v. 3.0. EJB Core Contracts and Requirements.

[2] JSR-250: Common Annotations for the Java Platform. *http://jcp.org/en/jsr/detail?id=250*.

[3] JSR-175: A Metadata Facility for the Java Programming Language. *http://jcp.org/en/jsr/detail?id=175*.

[4] Database Language SQL. ANSI X3.135-1992 or ISO/IEC 9075:1992.

[5] Enterprise JavaBeans, v 2.1. *http://java.sun.com/products/ejb.*

[6] JDBC 3.0 Specification. *http://java.sun.com/products/jdbc*.

[7] Enterprise JavaBeans, Simplified API, v 3.0. *http://java.sun.com/products/ejb.*

Appendix A Revision History

This appendix lists the significant changes that have been made during the development of the EJB 3.0 specification.

A.1 Early Draft 1

Created document.

A.2 Early Draft 2

Split *Persistence API* from single Early Draft 1 document.

Renamed dependent classes as "embedded classes".

Added support for EJB 2.1 style composite keys for entities.

Added support for BLOBs and CLOBs

Clarified rules for defaulting of O/R mapping when OneToOne, OneToMany, ManyToOne, and ManyToMany annotations are used.

Clarified default mappings for non-relationship fields and properties.

Clarified exceptions for entity lifecycle operations and `EntityManager` and `Query` interface methods.

Clarified semantics of `contains` method.

Renaming of annotations for dependent objects to reflect "embedded" terminology.

Added EmbeddedId and IdClass annotations to support composite keys.

Added AttributeOverride annotation to support embedded objects and embedded primary keys.

Added annotations to support BLOB/CLOB mappings.

Renamed GeneratorTable annotation as GeneratedIdTable.

Added setFlushMode method to Query interface.

Added missing Transient annotation.

Rename create() method as persist() in EntityManager API, and CREATE as PERSIST in CascadeType enum.

Provided full definition of EJB QL.

Removed POSITION, CHAR_LENGTH, and CHARACTER_LENGTH as redundant.

Added support for mapping of SQL query results.

Extended EJB QL queries to apply to embedded classes.

Added XML descriptor.

Added Related Documents section.

Updated numerous examples.

A.3 Changes Since EDR 2

Clearer formatting for description of merge operation.

Removed requirements for java.sql.Blob and java.sql.Clob.

Added java.util.Date and java.sql.Date as permitted primary key types.

Added introduction to O/R mapping metadata specification.

Removed primary annotation element from UniqueConstraint, Column, and JoinColumn annotations as redundant.

Clarified that UniqueConstraint applies in addition to unique constraints entailed by primary key mappings.

Clarified that PostLoad method should be invoked after refresh.

Added caution about use of business logic in accessor methods when access=PROPERTY.

Clarified that precision and scale apply to decimal columns.

Editorial changes to remove implications that entity lifecycle operations entail implementation in terms of a “state” model.

Removed entityType and version elements of Entity annotation.

Added note about the use of EJB QL bulk update and delete operations.

Clarified that fetch=LAZY is a hint; implementations may elect to prefetch.

Clarified that only a single version property is required to be supported per class.

Allowed persistent instance variables to be private.

Removed requirement that if access=FIELD, the fields in the primary key class must be public or protected.

Extended mapping defaults for fields and properties of byte[], Byte[], char[], and Character[] to Basic mapping type.

Made TemporalType enum top-level; added NONE so that it can be used to specify Basic mapping for temporal types.

Clarified that query execution methods getResultList and getSingleResult throw IllegalStateException when called for EJB QL UPDATE or DELETE statements; executeUpdate throws IllegalStateException when called for EJB QL SELECT statement.

Clarified that constructor names in EJB QL queries must be fully qualified.

Removed requirement for support of BIT_LENGTH function from EJB QL.

The executeUpdate method throws TransactionRequiredException if there is no active transaction.

Clarified that EJB QL delete operation does not cascade.

Added support for use of EntityManager in application-managed environments, including outside of J2EE containers.

Added EntityManager bootstrapping APIs.

Added support for extended persistence contexts.

Added support for non-entity classes in the entity inheritance hierarchy.

Added supported support for abstract entity classes in the entity inheritance hierarchy.

Added EmbeddableSuperclass annotation.

Clarifications to EntityManager and Query exceptions.

Added LEFT, EXISTS, ALL, ANY, SOME to EJB QL reserved identifiers.

Renamed InheritanceJoinColumn as PrimaryKeyJoinColumn. Removed usePKasFK from the OneToOne annotation, clarifying that PrimaryKeyJoinColumn can be used instead.

Clarified result types for aggregate functions.

Clarification of TRIM function and its arguments.

In OneToOne, OneToMany, ManyToOne, ManyToMany annotations, targetEntity type is Class, note String.

Merge @Serialized annotation into @Basic.

Added discriminatorColumn element to @EntityResult

Instance variables allowed to be private, package visibility.

Removed restriction about use of identification variable for IS EMPTY in the FROM clause, since this is no longer true given outer joins.

Removed restriction that @Table must have been explicitly specified if @SecondaryTable is used—this is unnecessary, since defaults can be used.

Removed specified element for @Column: it is not needed.

Remove operation applied to removed entity is ignored.

EntityManager.find changed to return null if the entity does not exist.

EntityManager.contains doesn't require a transaction be active.

Added @OrderBy, @MapKey annotations

Clarified rules regarding the availability of detached instances.

Added SIZE function to EJB QL.

Cleaned up EJB QL grammar.

Added optional hint to Basic and Lob annotations.

Added EntityManager.getReference().

EJB QL LIKE operator allows string-expressions.

Added chapters with contracts on packaging, deployment, and bootstrapping outside a container.

Merged GeneratedIdTable into TableGenerator annotation to resolve overlap between the two.

Updated XML descriptor to match annotations.

Editorial sweep over document.