JAKARTA EE

Jakarta Persistence

Jakarta Persistence Team, https://projects.eclipse.org/projects/ee4j.jpa

3.0-SNAPSHOT, March 06, 2020

Table of Contents

```
Specification: Jakarta Persistence

Version: 3.0-SNAPSHOT

Status: DRAFT

Release: March 06, 2020
```


Eclipse Foundation Specification License

By using and/or copying this document, or the Eclipse Foundation document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the Eclipse Foundation document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the document, or portions thereof, that you use:

- link or URL to the original Eclipse Foundation document.
- All existing copyright notices, or if one does not exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright (c) [$date-of-document] Eclipse Foundation, Inc. [url to this license]"

Inclusion of the full text of this NOTICE must be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of Eclipse Foundation documents is granted pursuant to this license, except anyone may prepare and distribute derivative works and portions of this document in software that implements the specification, in supporting materials accompanying such software, and in documentation of such software, PROVIDED that all such works include the notice below. HOWEVER, the publication of derivative works of this document for use as a technical specification is expressly prohibited.

The notice is:

"Copyright (c) 2018 Eclipse Foundation. This software or document includes material copied from or derived from [title and URI of the Eclipse Foundation specification document]."

Disclaimers

THIS DOCUMENT IS PROVIDED "AS IS," AND THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the copyright holders or the Eclipse Foundation may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.

Scope

Jakarta Persistence defines a standard for management of persistence and object/relational mapping in Java® environments.

Chapter 1. Introduction

This document is the specification of the Jakarta API for the management of persistence and object/relational mapping with Jakarta EE and Java SE. The technical objective of this work is to provide an object/relational mapping facility for the Java application developer using a Java domain model to manage a relational database.

The Jakarta Persistence 3.0 specification is the first release after moving the project to Eclipse Foundation. All APIs are moved from *javax.** *package* to *jakarta.** package. All properties containing *javax* as part of the name are renamed the way that *javax* is replaced with *jakarta*.

The Java Persistence 2.2 specification enhances the Jakarta Persistence API with support for repeating annotations; injection into attribute converters; support for mapping of the *java.time.LocalDate*, *java.time.LocalTime*, *java.time.LocalDateTime*, *java.time.OffsetTime*, and *java.time.OffsetDateTime* types; and methods to retrieve the results of *Query* and *TypedQuery* as streams.

The Java Persistence 2.1 specification added support for schema generation, type conversion methods, use of entity graphs in queries and find operations, unsynchronized persistence contexts, stored procedure invocation, and injection into entity listener classes. It also includes enhancements to the Java Persistence query language, the Criteria API, and to the mapping of native queries.

1.1. Expert Group

This revision to the JPA specification is based on JPA 2.1, whose work was conducted as part of JSR 338 under the Java Community Process Program. This specification is the result of the collaborative work of the members of the JSR 338 Expert Group: akquinet tech@Spree: Michael Bouschen; Ericsson: Nicolas Seyvet; IBM: Kevin Sutter, Pinaki Poddar; OW2: Florent Benoit; Oracle: Linda DeMichiel, Gordon Yorke, Michael Keith; Pramati Technologies: Deepak Anupalli; Red Hat, Inc.: Emmanuel Bernard, Steve Ebersole, Scott Marlow; SAP AG: Rainer Schweigkoffer; Sybase: Evan Ireland; Tmax Soft Inc.: Miju Byon; Versant: Christian von Kutzleben; VMware: Oliver Gierke; individual members: Matthew Adams; Adam Bien; Bernd Mueller; Werner Keil.

The work of the JSR 338 Expert Group was conducted using the jpa-spec.java.net project.

1.2. Document Conventions

The regular Times font is used for information that is prescriptive by this 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.

A monospaced font is used for code examples and to specify the BNF of the Java Persistence query language.

This document is written in terms of the use of Java language metadata annotations. An XML

descriptor (as specified in Chapter 12) may be used as an alternative to annotations or to augment or override annotations. The elements of this descriptor mirror the annotations and have the same semantics. When semantic requirements are written in terms of annotations, it should be understood that the same semantics apply when the XML descriptor is used as an alternative.

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.

This chapter describes requirements on entity classes and instances.

2.1. 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.

The entity class must be a top-level class. An enum or interface must not be designated as an entity.

The entity class must not be final. No methods or persistent instance variables of the entity class may be final.

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.

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 persistent state of an entity is represented by instance variables, which may correspond to JavaBeans properties. An instance variable must 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 methods—i.e., accessor methods (getter/setter methods) or other business methods.

2.2. Persistent Fields and Properties

The persistent state of an entity is accessed by the persistence provider runtime [1: The term "persistence provider runtime" refers to the runtime environment of the persistence implementation. In Java EE environments, this may be the Java EE container or a third-party persistence provider implementation integrated with it.] either via JavaBeans style property accessors ("property access") or via instance variables ("field access"). Whether persistent properties or persistent fields or a combination of the two is used for the provider's access to a given class or entity hierarchy is determined as described in Section 2.3.

Terminology Note: The persistent fields and properties of an entity class are generically referred to in this document as the “attributes” of the class.

The instance variables of a class must be private, protected, or package visibility independent of whether field access or property access is used. When property access is used, the property accessor methods must be public or protected.

It is required that the entity class follow the method signature conventions for JavaBeans read/write properties (as defined by the JavaBeans Introspector class) for persistent properties when property access is 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* may be used as an alternative name for the getter method. [2: Specifically, if get *X* is the name of the getter method and set *X* is the name of the setter method, where *X* is a string, the name of the persistent property is defined by the result of java.beans.Introspector.decapitalize(*X*).]

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

```
T getProperty()

void setProperty(T t)
```

Collection-valued persistent fields and properties must be defined in terms of one of the following collection-valued interfaces regardless of whether the entity class otherwise adheres to the JavaBeans method conventions noted above and whether field or property access is used: *java.util.Collection*, *java.util.Set*, *java.util.List* [3: Portable applications should not expect the order of a list to be maintained across persistence contexts unless the *OrderColumn* construct is used or unless the *OrderBy* construct is used and the modifications to the list observe the specified ordering.], *java.util.Map*. The collection implementation type may be used by the application to initialize fields or properties before the entity is made persistent. Once the entity becomes managed (or detached), subsequent access must be through the interface type.

Terminology Note: The terms “collection” and “collection-valued” are used in this specification to denote any of the above types unless further qualified. In cases where a *java.util.Collection* type (or one of its subtypes) is to be distinguished, the type is identified as such. The terms “map” and “map collection” are used to apply to a collection of type *java.util.Map* when a collection of type *java.util.Map* needs to be distinguished as such.

For collection-valued persistent properties, type *T* must be one of these collection interface types in the method signatures above. Use of the generic variants of these collection types is encouraged (for example, *Set<Order>*).

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

runtime executes this logic when property-based access is used.

Caution should be exercised in adding business logic to the accessor methods when property access is used. The order in which the persistence provider runtime calls these methods when loading or storing persistent state is not defined. Logic contained in such methods therefore should not rely upon a specific invocation order.

If property access is used and lazy fetching is specified, portable applications should not directly access the entity state underlying the property methods of managed instances until after it has been fetched by the persistence provider. [4: Lazy fetching is a hint to the persistence provider and can be specified by means of the *Basic*, *OneToOne*, *OneToMany*, *ManyToOne*, *ManyToMany*, and *ElementCollection* annotations and their XML equivalents. See Chapter 11.]

If a persistence context is joined to a transaction, runtime exceptions thrown by property accessor methods cause the current transaction to be marked for rollback; exceptions thrown by such methods when used by the persistence runtime to load or store persistent state cause the persistence runtime to mark the current transaction for rollback 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* *[5: Note that an instance of Calendar must be fully initialized for the type that it is mapped to.]*, *java.sql.Date*, *java.sql.Time*, *java.sql.Timestamp*, *byte[]*, *Byte[]*, *char[]*, *Character[]*, *java.time.LocalDate*, *java.time.LocalTime*, *java.time.LocalDateTime*, *java.time.OffsetTime*, *java.time.OffsetDateTime*, and user-defined types that implement the *Serializable* interface *)*; enums; entity types; collections of entity types; embeddable classes (see Section 2.5); collections of basic and embeddable types (see Section 2.6).

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 11.

2.2.1. 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 // property access is used
    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);
```

```
        // Update the phone entity instance to refer to this customer
        phone.addCustomer(this);
    }
}
```

2.3. Access Type

2.3.1. Default Access Type

By default, a single access type (field or property access) applies to an entity hierarchy. The default access type of an entity hierarchy is determined by the placement of mapping annotations on the attributes of the entity classes and mapped superclasses of the entity hierarchy that do not explicitly specify an access type. An access type is explicitly specified by means of the *Access* annotation [6: The use of XML as an alternative and the interaction between Java language annotations and XML elements in defining default and explicit access types is described in Chapter 12.], as described in Section 2.3.2.

When annotations are used to define a default access type, the placement of the mapping annotations on either the persistent fields or persistent properties of the entity class specifies the access type as being either field- or property-based access respectively.

When field-based access is used, the object/relational mapping annotations for the entity class annotate the instance variables, and the persistence provider runtime accesses instance variables directly. All non- *transient* instance variables that are not annotated with the *Transient* annotation are persistent.

When property-based access is used, the object/relational mapping annotations for the entity class annotate the getter property accessors [7: These annotations must not be applied to the setter methods.], and the persistence provider runtime accesses persistent state via the property accessor methods. All properties not annotated with the *Transient* annotation are persistent.

Mapping annotations must not be applied to fields or properties that are *transient* or *Transient*.

All such classes in the entity hierarchy whose access type is defaulted in this way must be consistent in their placement of annotations on either fields or properties, such that a single, consistent default access type applies within the hierarchy. Any embeddable classes used by such classes will have the same access type as the default access type of the hierarchy unless the *Access* annotation is specified as defined below.

It is an error if a default access type cannot be determined and an access type is not explicitly specified by means of annotations or the XML descriptor. The behavior of applications that mix the placement of annotations on fields and properties within an entity hierarchy without explicitly specifying the *Access* annotation is undefined.

2.3.2. Explicit Access Type

An access type for an individual entity class, mapped superclass, or embeddable class can be specified for that class independent of the default for the entity hierarchy by means of the *Access* annotation applied to the class. This explicit access type specification does not affect the access type of other entity classes or mapped superclasses in the entity hierarchy. The following rules apply:

When *Access(FIELD)* is applied to an entity class, mapped superclass, or embeddable class, mapping annotations may be placed on the instance variables of that class, and the persistence provider runtime accesses persistent state via the instance variables defined by the class. All non-transient instance variables that are not annotated with the *Transient* annotation are persistent. When *Access(FIELD)* is applied to such a class, it is possible to selectively designate individual attributes within the class for property access. To specify a persistent property for access by the persistence provider runtime, that property must be designated *Access(PROPERTY)*. [8: It is permitted (but redundant) to place Access(FIELD) on a persistent field whose class has field access type or Access(PROPERTY) on a persistent property whose class has property access type. It is not permitted to specify a field as Access(PROPERTY) or a property as Access(FIELD). Note that Access(PROPERTY) must not be placed on the setter methods.] The behavior is undefined if mapping annotations are placed on any properties defined by the class for which *Access(PROPERTY)* is not specified. Persistent state inherited from superclasses is accessed in accordance with the access types of those superclasses.

When *Access(PROPERTY)* is applied to an entity class, mapped superclass, or embeddable class, mapping annotations may be placed on the properties of that class, and the persistence provider runtime accesses persistent state via the properties defined by that class. All properties that are not annotated with the *Transient* annotation are persistent. When *Access(PROPERTY)* is applied to such a class, it is possible to selectively designate individual attributes within the class for instance variable access. To specify a persistent instance variable for access by the persistence provider runtime, that instance variable must be designated *Access(FIELD)*. The behavior is undefined if mapping annotations are placed on any instance variables defined by the class for which *Access(FIELD)* is not specified. Persistent state inherited from superclasses is accessed in accordance with the access types of those superclasses.

Note that when access types are combined within a class, the *Transient* annotation should be used to avoid duplicate persistent mappings.

2.3.3. Access Type of an Embeddable Class

The access type of an embeddable class is determined by the access type of the entity class, mapped superclass, or embeddable class in which it is embedded (including as a member of an element collection) independent of whether the access type of the containing class has been explicitly specified or defaulted. A different access type for an embeddable class can be specified for that embeddable class by means of the *Access* annotation as described above.

2.3.4. Defaulted Access Types of Embeddable Classes and Mapped Superclasses

Care must be exercised when defining an embeddable class or mapped superclass which is used both in a context of field access and in a context of property access and whose access type is not explicitly specified by means of the *Access* annotation or XML mapping file.

Such classes should be defined so that the number, names, and types of the resulting persistent attributes are identical, independent of the access type in use. The behavior of such classes whose attributes are not independent of access type is otherwise undefined with regard to use with the metamodel API if they occur in contexts of differing access types within the same persistence unit.

2.4. Primary Keys and Entity Identity

Every entity must have a primary key.

The primary key must be defined on the entity class that is the root of the entity hierarchy or on a mapped superclass that is a (direct or indirect) superclass of all entity classes in the entity hierarchy. The primary key must be defined exactly once in an entity hierarchy.

A primary key corresponds to one or more fields or properties (“attributes”) of the entity class.

A simple (i.e., non-composite) primary key must correspond to a single persistent field or property of the entity class. The *Id* annotation or *id* XML element must be used to denote a simple primary key. See Section 11.1.21.

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 *EmbeddedId* or *IdClass* annotation is used to denote a composite primary key. See Section 11.1.17 and Section 11.1.22.

A simple primary key or a field or property of a composite primary key should be one of the following types: any Java primitive type; any primitive wrapper type; *java.lang.String*; *java.util.Date*; *java.sql.Date*; *java.math.BigDecimal*; *java.math.BigInteger*. [9: In general, however, approximate numeric types (e.g., floating point types) should never be used in primary keys.] If the primary key is a composite primary key derived from the primary key of another entity, the primary key may contain an attribute whose type is that of the primary key of the referenced entity as described in Section 2.4.1. Entities whose primary keys use types other than these will not be portable. If generated primary keys are used, only integral types will be portable. If *java.util.Date* is used as a primary key field or property, the temporal type should be specified as *DATE*.

The following rules apply for composite primary keys:

- The primary key class must be public and must have a public no-arg constructor.
- The access type (field- or property-based access) of a primary key class is determined by the access type of the entity for which it is the primary key unless the primary key is a embedded id and a

different access type is specified. See Section Section 2.3.

- If property-based access is used, 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 must either be represented and mapped as an embeddable class (see Section 11.1.17) or must be represented as an id class and mapped to multiple fields or properties of the entity class (see Section 11.1.22).
- If the composite primary key class is represented as an id class, the names of primary key fields or properties in the primary key class and those of the entity class to which the id class is mapped must correspond and their types must be the same.
- A primary key that corresponds to a derived identity must conform to the rules of Section 2.4.1.

The value of its primary key uniquely identifies an entity instance within a persistence context and to *EntityManager* operations as described in Chapter 3. The application must not change the value of the primary key [10: This includes not changing the value of a mutable type that is primary key or an attribute of a composite primary key.]. The behavior is undefined if this occurs. [11: The implementation may, but is not required to, throw an exception. Portable applications must not rely on any such specific behavior.]

2.4.1. Primary Keys Corresponding to Derived Identities

The identity of an entity may be derived from the identity of another entity (the “parent” entity) when the former entity (the “dependent” entity) is the owner of a many-to-one or one-to-one relationship to the parent entity and a foreign key maps the relationship from dependent to parent.

If a many-to-one or one-to-one entity relationship corresponds to a primary key attribute, the entity containing this relationship cannot be persisted without the relationship having been assigned an entity since the identity of the entity containing the relationship is derived from the referenced entity. [12: If the application does not set the primary key attribute corresponding to the relationship, the value of that attribute may not be available until after the entity has been flushed to the database.]

Derived identities may be captured by means of simple primary keys or by means of composite primary keys as described in Section 2.4.1.1 below.

If the dependent entity class has primary key attributes in addition to those corresponding to the parent’s primary key or if the parent has a composite primary key, an embedded id or id class must be used to specify the primary key of the dependent entity. It is not necessary that parent entity and dependent entity both use embedded ids or both use id classes to represent composite primary keys when the parent has a composite key.

A dependent entity may have more than one parent entity.

2.4.1.1. Specification of Derived Identities

If the dependent entity uses an id class to represent its primary key, one of the two following rules must be observed:

- The names of the attributes of the id class and the *Id* attributes of the dependent entity class must correspond as follows:
  - The *Id* attribute in the entity class and the corresponding attribute in the id class must have the same name.
  - If an *Id* attribute in the entity class is of basic type, the corresponding attribute in the id class must have the same type.
  - If an *Id* attribute in the entity is a many-to-one or one-to-one relationship to a parent entity, the corresponding attribute in the id class must be of the same Java type as the id class or embedded id of the parent entity (if the parent entity has a composite primary key) or the type of the *Id* attribute of the parent entity (if the parent entity has a simple primary key).
- If the dependent entity has a single primary key attribute (i.e., the relationship attribute), the id class specified by the dependent entity must be the same as the primary key class of the parent entity. The *Id* annotation is applied to the relationship to the parent entity. [13: Note that it is correct to observe the first rule as an alternative in this case.]

If the dependent entity uses an embedded id to represent its primary key, the attribute in the embedded id corresponding to the relationship attribute must be of the same type as the primary key of the parent entity and must be designated by the *MapsId* annotation applied to the relationship attribute. The *value* element of the *MapsId* annotation must be used to specify the name of the attribute within the embedded id to which the relationship attribute corresponds. If the embedded id of the dependent entity is of the same Java type as the primary key of the parent entity, the relationship attribute maps both the relationship to the parent and the primary key of the dependent entity, and in this case the *MapsId* annotation is specified without the *value* element. [14: Note that the parent's primary key might be represented as either an embedded id or as an id class.]

If the dependent entity has a single primary key attribute (i.e, the relationship attribute or an attribute that corresponds to the relationship attribute) and the primary key of the parent entity is a simple primary key, the primary key of the dependent entity is a simple primary key of the same type as that of the parent entity (and neither *EmbeddedId* nor *IdClass* is specified). In this case, either (1) the relationship attribute is annotated *Id*, or (2) a separate *Id* attribute is specified and the relationship attribute is annotated *MapsId* (and the *value* element of the *MapsId* annotation is not specified).

2.4.1.2. Mapping of Derived Identities

A primary key attribute that is derived from the identity of a parent entity is mapped by the corresponding relationship attribute. The default mapping for this relationship is as specified in Section 2.10. In the case where a default mapping does not apply or where a default mapping is to be

overridden, the *JoinColumn* or *JoinColumns* annotation is used on the relationship attribute.

If the dependent entity uses an embedded id to represent its primary key, the *AttributeOverride* annotation may be used to override the default mapping of embedded id attributes that do not correspond to the relationship attributes mapping the derived identity. The embedded id attributes that correspond to the relationship are treated by the provider as "read only"—that is, any updates to them on the part of the application are not propagated to the database.

If the dependent uses an id class, the *Column* annotation may be used to override the default mapping of *Id* attributes that are not relationship attributes.

2.4.1.3. Examples of Derived Identities

Example 1:

The parent entity has a simple primary key:

```
@Entity
public class Employee {
    @Id long empId;
    String empName;

    // ...
}
```

Case (a): The dependent entity uses *IdClass* to represent a composite key:

```
public class DependentId {
    String name; // matches name of @Id attribute
    long emp; // matches name of @Id attribute and type of Employee PK
}

@Entity
@IdClass(DependentId.class)
public class Dependent {
    @Id String name;

    // id attribute mapped by join column default
    @Id @ManyToOne
    Employee emp;

    // ...
}
```

Sample query:

```
SELECT d
FROM Dependent d
WHERE d.name = 'Joe' AND d.emp.empName = 'Sam'
```

Case(b): The dependent entity uses *EmbeddedId* to represent a composite key:

```
@Embeddable
public class DependentId {
    String name;
    long empPK; // corresponds to PK type of Employee
}

@Entity
public class Dependent {
    @EmbeddedId DependentId id;

    // id attribute mapped by join column default
    @MapsId("empPK") // maps empPK attribute of embedded id
    @ManyToOne
    Employee emp;

    // ...
}
```

Sample query:

```
SELECT d
FROM Dependent d
WHERE d.id.name = 'Joe' AND d.emp.empName = 'Sam'
```

Example 2:

The parent entity uses *IdClass*:

```
public class EmployeeId {
    String firstName;
    String lastName;

    // ...
}

@Entity
@IdClass(EmployeeId.class)
public class Employee {
    @Id String firstName
    @Id String lastName

   // ...
}
```

Case (a): The dependent entity uses *IdClass*:

```
public class DependentId {
    String name; // matches name of attribute
    EmployeeId emp; //matches name of attribute and type of Employee PK
}

@Entity
@IdClass(DependentId.class)
public class Dependent {
    @Id
    String name;

    @Id
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })

    @ManyToOne
    Employee emp;
}
```

Sample query:

```
SELECT d
FROM Dependent d
WHERE d.name = 'Joe' AND d.emp.firstName = 'Sam'
```

Case (b): The dependent entity uses *EmbeddedId*. The type of the *empPK* attribute is the same as that of the primary key of *Employee*. The *EmployeeId* class needs to be annotated *Embeddable* or denoted as an embeddable class in the XML descriptor.

```
@Embeddable
public class DependentId {
    String name;
    EmployeeId empPK;
}

@Entity
public class Dependent {
    @EmbeddedId
    DependentId id;

    @MapsId("empPK")
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })

    @ManyToOne
    Employee emp;

    // ...
}
```

Sample query:

```
SELECT d
FROM Dependent d
WHERE d.id.name = 'Joe' AND d.emp.firstName = 'Sam'
```

Note that the following alternative query will yield the same result:

```
SELECT d
FROM Dependent d
WHERE d.id.name = 'Joe' AND d.id.empPK.firstName = 'Sam'
```

Example 3:

The parent entity uses *EmbeddedId*:

```
@Embeddable
public class EmployeeId {
    String firstName;
    String lastName;

    // ...
}

@Entity
public class Employee {
    @EmbeddedId
    EmployeeId empId;

    // ...
}
```

Case (a): The dependent entity uses *IdClass*:

```
public class DependentId {
    String name; // matches name of @Id attribute
    EmployeeId emp; // matches name of @Id attribute and type of embedded id of Employee
}

@Entity
@IdClass(DependentId.class)
public class Dependent {
    @Id
    @Column(name="dep_name") // default column name is overridden
    String name;

    @Id
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })

    @ManyToOne Employee
    emp;
}
```

Sample query:

```
SELECT d
FROM Dependent d
WHERE d.name = 'Joe' and d.emp.empId.firstName = 'Sam'
```

Case (b): The dependent entity uses *EmbeddedId*:

```
@Embeddable
public class DependentId {
    String name;
    EmployeeId empPK; // corresponds to PK type of Employee
}

@Entity
public class Dependent {
    // default column name for "name" attribute is overridden
    @AttributeOverride(name="name", column=@Column(name="dep_name"))
    @EmbeddedId DependentId id;

    @MapsId("empPK")
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })
    @ManyToOne
    Employee emp;

    // ...
}
```

Sample query:

```
SELECT d
FROM Dependent d
WHERE d.id.name = 'Joe' and d.emp.empId.firstName = 'Sam'
```

Note that the following alternative query will yield the same result:

```
SELECT d
FROM Dependent d
WHERE d.id.name = 'Joe' AND d.id.empPK.firstName = 'Sam'
```

Example 4:

The parent entity has a simple primary key:

```
@Entity
public class Person {
    @Id
    String ssn;

    // ...
}
```

Case (a): The dependent entity has a single primary key attribute which is mapped by the relationship attribute. The primary key of *MedicalHistory* is of type *String*.

```
@Entity
public class MedicalHistory {
    // default join column name is overridden
    @Id
    @OneToOne
    @JoinColumn(name="FK")
    Person patient;

    // ...
}
```

Sample query:

```
SELECT m
FROM MedicalHistory m
WHERE m.patient.ssn = '123-45-6789'
```

Case (b): The dependent entity has a single primary key attribute corresponding to the relationship attribute. The primary key attribute is of the same basic type as the primary key of the parent entity. The *MapsId* annotation applied to the relationship attribute indicates that the primary key is mapped by the relationship attribute. [15: Note that the use of PrimaryKeyJoinColumn instead of MapsId would result in the same mapping in this example. Use of MapsId is preferred for the mapping of derived identities.]

```
@Entity
public class MedicalHistory {
    @Id
    String id; // overriding not allowed

    // ...

    // default join column name is overridden
    @MapsId
    @JoinColumn(name="FK")
    @OneToOne
    Person patient;

    // ...
}
```

Sample query:

```
SELECT m
FROM MedicalHistory m WHERE m.patient.ssn = '123-45-6789'
```

Example 5:

The parent entity uses *IdClass*. The dependent's primary key class is of same type as that of the parent entity.

```
public class PersonId {
    String firstName;
    String lastName;
}

@Entity
@IdClass(PersonId.class)
public class Person {
    @Id
    String firstName;

    @Id
    String lastName;

    // ...
}
```

Case (a): The dependent entity uses *IdClass*:

```
@Entity
@IdClass(PersonId.class)
public class MedicalHistory {
    @Id
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })

    @OneToOne
    Person patient;

    // ...
}
```

Sample query:

```
SELECT m
FROM MedicalHistory m
WHERE m.patient.firstName = 'Charles'
```

Case (b): The dependent entity uses the *EmbeddedId* and *MapsId* annotations. The *PersonId* class needs to be annotated *Embeddable* or denoted as an embeddable class in the XML descriptor.

```
@Entity
public class MedicalHistory {
    // all attributes map to relationship:
    AttributeOverride not allowed

    @EmbeddedId
    PersonId id;

    // ...

    @MapsId
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })

    @OneToOne Person patient;

    // ...
}
```

Sample query:

```
SELECT m
FROM MedicalHistory m
WHERE m.patient.firstName = 'Charles'
```

Note that the following alternative query will yield the same result:

```
SELECT m
FROM MedicalHistory m
WHERE m.id.firstName = 'Charles'
```

Example 6:

The parent entity uses *EmbeddedId*. The dependent's primary key is of the same type as that of the parent.

```
@Embeddable
public class PersonId {
    String firstName;
    String lastName;
}

@Entity
public class Person {
    @EmbeddedId PersonId id;

    // ...
}
```

Case (a): The dependent class uses *IdClass*:

```
@Entity
@IdClass(PersonId.class)
public class MedicalHistory {
    @Id
    @OneToOne
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })

    Person patient;

    // ...
}
```

Case (b): The dependent class uses *EmbeddedId*:

```
@Entity
public class MedicalHistory {
    // All attributes are mapped by the relationship
    // AttributeOverride is not allowed
    @EmbeddedId PersonId id;

    // ...

    @MapsId
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })
    @OneToOne
    Person patient;

    // ...
}
```

2.5. Embeddable Classes

An entity may use other fine-grained classes to represent entity state. Instances of these classes, unlike entity instances, do not have persistent identity of their own. Instead, they exist only as part of the state of the entity to which they belong. An entity may have collections of embeddables as well as single-valued embeddable attributes. Embeddables may also be used as map keys and map values. 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.

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*. Embeddable classes must be annotated as *Embeddable* or denoted in the XML descriptor as such. The access type for an embedded object is determined as described in Section 2.3.

An embeddable class may be used to represent the state of another embeddable class.

An embeddable class (including an embeddable class within another embeddable class) may contain a collection of a basic type or other embeddable class. [16: Direct or indirect circular containment dependencies among embeddable classes are not permitted.]

An embeddable class may contain a relationship to an entity or collection of entities. Since instances of embeddable classes themselves have no persistent identity, the relationship *from* the referenced entity is to the *entity* that contains the embeddable instance(s) and not to the embeddable itself. [17: An entity cannot have a unidirectional relationship to the embeddable class of another entity (or itself).] An embeddable class that is used as an embedded id or as a map key must not contain such a relationship.

Additional requirements and restrictions on embeddable classes are described in Section 2.6.

2.6. Collections of Embeddable Classes and Basic Types

A persistent field or property of an entity or embeddable class may correspond to a collection of a basic type or embeddable class (“element collection”). Such a collection, when specified as such by the *ElementCollection* annotation, is mapped by means of a collection table, as defined in Section 11.1.8. If the *ElementCollection* annotation (or XML equivalent) is not specified for the collection-valued field or property, the rules of Section 2.8 apply.

An embeddable class (including an embeddable class within another embeddable class) that is contained within an element collection must not contain an element collection, nor may it contain a relationship to an entity other than a many-to-one or one-to-one relationship. The embeddable class must be on the owning side of such a relationship and the relationship must be mapped by a foreign key mapping. (See Section 2.9)

2.7. Map Collections

Collections of elements and entity relationships can be represented as *java.util.Map* collections.

The map key and the map value independently can each be a basic type, an embeddable class, or an entity.

The *ElementCollection*, *OneToMany*, and *ManyToMany* annotations are used to specify the map as an element collection or entity relationship as follows: when the map value is a basic type or embeddable class, the *ElementCollection* annotation is used; when the map value is an entity, the *OneToMany* or *ManyToMany* annotation is used.

Bidirectional relationships represented as *java.util.Map* collections support the use of the *Map* datatype

on one side of the relationship only.

2.7.1. Map Keys

If the map key type is a basic type, the *MapKeyColumn* annotation can be used to specify the column mapping for the map key. If the *MapKeyColumn* annotation is not specified, the default values of the *MapKeyColumn* annotation apply as described in Section 11.1.33.

If the map key type is an embeddable class, the mappings for the map key columns are defaulted according to the default column mappings for the embeddable class. (See Section 11.1.9). The *AttributeOverride* and *AttributeOverrides* annotations can be used to override these mappings, as described in Section 11.1.4 and Section 11.1.5. If an embeddable class is used as a map key, the embeddable class must implement the *hashCode* and *equals* methods consistently with the database columns to which the embeddable is mapped [18: Note that when an embeddable instance is used as a map key, these attributes represent its identity. Changes to embeddable instances used as map keys have undefined behaviour and should be avoided.].

If the map key type is an entity, the *MapKeyJoinColumn* and *MapKeyJoinColumns* annotations are used to specify the column mappings for the map key. If the primary key of the referenced entity is a simple primary key and the *MapKeyJoinColumn* annotation is not specified, the default values of the *MapKeyJoinColumn* annotation apply as described in Section 11.1.35.

If Java generic types are not used in the declaration of a relationship attribute of type *java.util.Map*, the *MapKeyClass* annotation must be used to specify the type of the key of the map.

The *MapKey* annotation is used to specify the special case where the map key is itself the primary key or a persistent field or property of the entity that is the value of the map. The *MapKeyClass* annotation is not used when *MapKey* is specified.

2.7.2. Map Values

When the value type of the map is a basic type or an embeddable class, a collection table is used to map the map. If Java generic types are not used, the *targetClass* element of the *ElementCollection* annotation must be used to specify the value type for the map. The default column mappings for the map value are derived according to the default mapping rules for the *CollectionTable* annotation defined in Section 11.1.8. The *Column* annotation is used to override these defaults for a map value of basic type. The *AttributeOverride(s)* and *AssociationOverride(s)* annotations are used to override the mappings for a map value that is an embeddable class.

When the value type of the map is an entity, a join table is used to map the map for a many-to-many relationship or, by default, for a one-to-many unidirectional relationship. If the relationship is a bidirectional one-to-many/many-to-one relationship, by default the map is mapped in the table of the entity that is the value of the map. If Java generic types are not used, the *targetEntity* element of the *OneToMany* or *ManyToMany* annotation must be used to specify the value type for the map. Default mappings are described in Section 2.10.

2.8. 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 11 (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 in the same way as if the field or property were annotated with the *Embedded* annotation. See Section 11.1.15 and Section 11.1.16.

If the type of the field or property is one of the following, it is mapped in the same way as it would if it were annotated 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*, *java.time.LocalDate*, *java.time.LocalTime*, *java.time.LocalDateTime*, *java.time.OffsetTime*, *java.time.OffsetDateTime*, *byte[]*, *Byte[]*, *char[]*, *Character[]*, enums, any other type that implements *Serializable*. See Section 11.1.6, Section 11.1.18, Section 11.1.28, and Section 11.1.53.

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

2.9. Entity Relationships

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

If there is an association between two entities, one of the following relationship modeling annotations must be applied to the corresponding persistent property or field 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. [19: For associations of type *java.util.Map*, *target type* refers to the type that is the Map *value*.] Equivalent XML elements may be used as an alternative to these mapping annotations.

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 2.10.

Relationships may be bidirectional or unidirectional. A bidirectional relationship has both an owning side and an inverse (non-owning) 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 3.2.4.

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.

Associations that are specified as *OneToOne* or *OneToMany* support use of the *orphanRemoval* option. The following behaviors apply when *orphanRemoval* is in effect:

If an entity that is the target of the relationship is removed from the relationship (by setting the relationship to null or removing the entity from the relationship collection), the remove operation will be applied to the entity being orphaned. The remove operation is applied at the time of the flush operation. The *orphanRemoval* functionality is intended for entities that are privately "owned" by their parent entity. Portable applications must otherwise not depend upon a specific order of removal, and must not reassign an entity that has been orphaned to another relationship or otherwise attempt to persist it. If the entity being orphaned is a detached, new, or removed entity, the semantics of *orphanRemoval* do not apply.

If the remove operation is applied to a managed source entity, the remove operation will be cascaded to the relationship target in accordance with the rules of Section 3.2.3, (and hence it is not necessary to specify *cascade=REMOVE* for the relationship) [20: If the parent is detached or new or was previously removed before the orphan was associated with it, the remove operation is not applied to the entity being orphaned.].

Section 2.10, defines relationship mapping defaults for entity relationships. Additional mapping annotations (e.g., column and table mapping annotations) may be specified to override or further refine the default mappings and mapping strategies described in Section 2.10.

In addition, this specification also requires support for the following alternative mapping strategies:

The mapping of unidirectional one-to-many relationships by means of foreign key mappings. The *JoinColumn* annotation or corresponding XML element must be used to specify such non-default mappings. See Section 11.1.25.

The mapping of unidirectional and bidirectional one-to-one relationships, bidirectional many-to-one/one-to-many relationships, and unidirectional many-to-one relationships by means of join table mappings. The *JoinTable* annotation or corresponding XML element must be used to specify such non-default mappings. See Section 11.1.27.

Such mapping annotations must be specified on the owning side of the relationship. Any overriding of mapping defaults 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.

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).

Note that it is the application that bears responsibility for maintaining the consistency of runtime relationships—for example, for insuring that the “one” and the “many” sides of a bidirectional relationship are consistent with one another when the application updates the relationship at runtime.

If there are no associated entities for a multi-valued relationship of an entity fetched from the database, the persistence provider is responsible for returning an empty collection as the value of the relationship.

2.10. Relationship Mapping Defaults

This section defines 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.

2.10.1. 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.

2.10.2. Bidirectional ManyToOne / OneToMany Relationships

Assuming that:

- Entity A references a single instance of Entity B.
- Entity B references a collection of Entity A [21: When the relationship is modeled as a *java.util.Map*, "Entity B references a collection of Entity A" means that Entity B references a map collection in which the type of the Map *value* is Entity A. The map key may be a basic type, embeddable class, or an entity.].
- 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*.

2.10.3. 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.

2.10.3.1. 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.

2.10.3.2. 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*

1. 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*.

2.10.4. 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*.

2.10.5. 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.

2.10.5.1. 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> annualReviews) {
        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*.

2.10.5.2. 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*.

2.11. 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 can be annotated with the *Entity* annotation, mapped as entities, and queried for as entities.

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

These concepts are described further in the following sections.

2.11.1. 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
@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")
@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

    // Defaults to FT_EMP.SALARY
    protected Integer salary;

    // ...
}

@Entity
@Table(name="PT_EMP")
@DiscriminatorValue("PT")
// PK column is PT_EMP.EMPID due to _PrimaryKeyJoinColumn_ default
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;

    // ...
}
```

2.11.2. Mapped Superclasses

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

A mapped superclass, unlike an entity, is not queryable and must not be passed as an argument to *EntityManager* or *Query* operations. Persistent relationships defined by a mapped superclass must be unidirectional.

Both abstract and concrete classes may be specified as mapped superclasses. The *MappedSuperclass* annotation (or *mapped-superclass* XML descriptor element) is used to designate a mapped superclass.

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

A class designated as a mapped superclass 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 mapped superclass itself. When applied to the subclasses, the inherited mappings will apply in the context of the subclass tables. Mapping information can be overridden in such subclasses by using the *AttributeOverride* and *AssociationOverride* annotations or corresponding XML elements.

All other entity mapping defaults apply equally to a class designated as a mapped superclass.

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

Example: Concrete class as a mapped superclass

```
@MappedSuperclass
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

    // Defaults to FTEMPLOYEE.SALARY
    protected Integer salary;

    public FTEmployee() {}

    public Integer getSalary() { ... }

    public void setSalary(Integer salary) { ... }
}

@Entity
@Table(name="PT_EMP")
@AssociationOverride(name="address", joincolumns=@JoinColumn(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) { ... }
}
```

2.11.3. Non-Entity Classes in the Entity Inheritance Hierarchy

An entity can have a non-entity superclass, which may be either a concrete or abstract class. [22: The superclass must not be an embeddable class or id 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 entity manager [23: If a transaction-scoped persistence context is used, it is not required to be retained across transactions.]. Any annotations on such superclasses are ignored.

Non-entity classes cannot be passed as arguments to methods of the *EntityManager* or *Query* interfaces [24: This includes instances of a non-entity class that extends an entity class.] 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 {
    protected Integer operationCount; // transient state

    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();
    }
}
```

2.12. 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:

- a single table per class hierarchy
- a joined subclass 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.
- a table per concrete entity class

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

Support for the table per concrete class inheritance mapping strategy is optional in this release. Applications that use this mapping strategy will not be portable.

Support for the combination of inheritance strategies within a single entity inheritance hierarchy is not required by this specification.

2.12.1. 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.

2.12.2. 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.

2.12.3. Table per Concrete 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.

2.13. Naming of Database Objects

Many annotations and annotation elements contain names of database objects or assume default names for database objects.

This specification requires the following with regard to the interpretation of the names referencing database objects. These names include the names of tables, columns, and other database elements. Such names also include names that result from defaulting (e.g., a table name that is defaulted from an entity name or a column name that is defaulted from a field or property name).

By default, the names of database objects must be treated as undelimited identifiers and passed to the database as such.

For example, assuming the use of an English locale, the following must be passed to the database as undelimited identifers so that they will be treated as equivalent for all databases that comply with the SQL Standard's requirements for the treatment of "regular identifiers" (undelimited identifiers) and "delimited identifiers" [2]:

```
@Table(name="Customer")
@Table(name="customer")
@Table(name="cUsTomer")
```

Similarly, the following must be treated as equivalent:

```
@JoinColumn(name="CUSTOMER")
@ManyToOne Customer customer;

@JoinColumn(name="customer")
@ManyToOne Customer customer;

@ManyToOne Customer customer;
```

To specify delimited identifiers, one of the following approaches must be used:

- It is possible to specify that all database identifiers in use for a persistence unit be treated as delimited identifiers by specifying the <*delimited-identifiers/*> element within the *persistence-unit-defaults* element of the object/relational xml mapping file. If the <*delimited-identifiers/*> element is specified, it cannot be overridden.
- It is possible to specify on a per-name basis that a name for a database object is to be interpreted as

a delimited identifier as follows:

- Using annotations, a name is specified as a delimited identifier by enclosing the name within double quotes, whereby the inner quotes are escaped, e.g., *@Table(name="\"customer\"")* .
- When using XML, a name is specified as a delimited identifier by use of double quotes, e.g., _<table name=""customer""/>_ [25: If <delimited-identifiers> is specified and individual annotations or XML elements or attributes use escaped double quotes, the double-quotes appear in the name of the database identifier.]

The following annotations contain elements whose values correspond to names of database identifiers and for which the above rules apply, including when their use is nested within that of other annotations:

- EntityResult(discriminatorColumn **element**)
- FieldResult(column **element**)
- ColumnResult(name **element**)
- CollectionTable(name, catalog, schema **elements**)
- Column(name, columnDefinition, table **elements**)
- DiscriminatorColumn(name, columnDefinition **elements**)
- ForeignKey(name, foreignKeyDefinition **elements**)
- Index(name, columnList **elements**)
- JoinColumn(name, referencedColumnName, columnDefinition, table **elements**)
- JoinTable(name, catalog, schema **elements**)
- MapKeyColumn(name, columnDefinition, table **elements**)
- MapKeyJoinColumn(name, referencedColumnName, columnDefinition, table **elements**)
- NamedStoredProcedureQuery(procedureName **element**)
- OrderColumn(name, columnDefinition **elements**)
- PrimaryKeyJoinColumn(name, referencedColumnName, columnDefinition **elements**)
- SecondaryTable(name, catalog, schema **elements**)
- SequenceGenerator(sequenceName, catalog, schema **elements**)
- StoredProcedureParameter(name **element**)
- Table(name, catalog, schema **elements**)
- TableGenerator(table, catalog, schema, pkColumnName, valueColumnName **elements**)
- UniqueConstraint(name, columnNames **elements**)

The following XML elements and types contain elements or attributes whose values correspond to names of database identifiers and for which the above rules apply:

- entity-mappings(schema, catalog **elements**)
- persistence-unit-defaults(schema, catalog **elements**)
- collection-table(name, catalog, schema **attributes**)
- column(name, table, column-definition **attributes**)
- column-result(name **attribute**)
- discriminator-column(name, column-definition **attributes**)
- entity-result(discriminator-column **attribute**)
- field-result(column **attribute**)
- foreign-key(name, foreign-key-definition **attributes**)
- index(name **attribute**, column-list **element**)
- join-column(name, referenced-column-name, column-definition, table **attributes**)
- join-table(name, catalog, schema **attributes**)
- map-key-column(name, column-definition, table **attributes**)
- map-key-join-column(name, referenced-column-name, column-definition, table **attributes**)
- named-stored-procedure-query(procedure-name **attribute**)
- order-column(name, column-definition **attributes**)
- primary-key-join-column(name, referenced-column-name, column-definition **attributes**)
- secondary-table(name, catalog, schema **attributes**)
- sequence-generator(sequence-name, catalog, schema **attributes**)
- stored-procedure-parameter(name **attribute**)
- table(name, catalog, schema **attributes**)
- table-generator(table, catalog, schema, pk-column-name, value-column-name **attributes**)
- unique-constraint(name **attribute**, column-name **element**)

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 persistent entities by primary key, and to query over persistent 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 relationships between entity managers and persistence contexts are described in Section 3.3 and in further detail in Chapter 7. Section 3.4 describes mechanisms for concurrency control and locking. Section 3.5 describes entity listeners and lifecycle callback methods for entities. Section 3.6 describes support for automatic use of Bean Validation. Section 3.7 describes the use of entity graphs to control the path and boundaries of find and query operations. Section 3.7 describes mechanisms for defining conversions between entity and database representations for attributes of basic types. Section 3.9 describes mechanisms for portable second-level cache configuration. The *Query*, *TypedQuery*, *StoredProcedureQuery*, and related interfaces are described in Section 3.10. Section 3.11 provides a summary of exceptions. The Java Persistence query language is defined in Chapter 4 and the APIs for the construction of Criteria queries in Chapter 6. The definition of persistence units is described in Chapter 8.

3.1.1. EntityManager Interface

```
package jakarta.persistence;

import java.util.Map;
import java.util.List;
import jakarta.persistence.metamodel.Metamodel;
import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaQuery;
import jakarta.persistence.criteria.CriteriaUpdate;
import jakarta.persistence.criteria.CriteriaDelete;

/**
```

```
 * Interface used to interact with the persistence context.
 *
 * <p> An <code>EntityManager</code> 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 <code>EntityManager</code> API is used
 * to create and remove persistent entity instances, to find entities
 * by their primary key, and to query over entities.
 *
 * <p> The set of entities that can be managed by a given
 * <code>EntityManager</code> 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.
 *
 * @see Query
 * @see TypedQuery
 * @see CriteriaQuery
 * @see PersistenceContext
 * @see StoredProcedureQuery
 *
 * @since 1.0
 */
public interface EntityManager {

    /**
     * Make an instance managed and persistent.
     * @param entity  entity instance
     * @throws EntityExistsException if the entity already exists.
     * (If the entity already exists, the <code>EntityExistsException</code> may
     * be thrown when the persist operation is invoked, or the
     * <code>EntityExistsException</code> or another <code>PersistenceException</code>
may be
     * thrown at flush or commit time.)
     * @throws IllegalArgumentException if the instance is not an
     *         entity
     * @throws TransactionRequiredException if there is no transaction when
     *         invoked on a container-managed entity manager of that is of type
     *         <code>PersistenceContextType.TRANSACTION</code>
     */
    public void persist(Object entity);

    /**
     * Merge the state of the given entity into the
     * current persistence context.
     * @param entity  entity instance
```

```
     * @return the managed 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 when
     *         invoked on a container-managed entity manager of that is of type
     *         <code>PersistenceContextType.TRANSACTION</code>
     */
    public <T> T merge(T entity);

    /**
     * Remove the entity instance.
     * @param entity  entity instance
     * @throws IllegalArgumentException if the instance is not an
     *         entity or is a detached entity
     * @throws TransactionRequiredException if invoked on a
     *         container-managed entity manager of type
     *         <code>PersistenceContextType.TRANSACTION</code> and there is
     *         no transaction
     */
    public void remove(Object entity);

    /**
     * Find by primary key.
     * Search for an entity of the specified class and primary key.
     * If the entity instance is contained in the persistence context,
     * it is returned from there.
     * @param entityClass  entity class
     * @param primaryKey  primary key
     * @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
     *         is not a valid type for that entity's primary key or
     *         is null
     */
    public <T> T find(Class<T> entityClass, Object primaryKey);

    /**
     * Find by primary key, using the specified properties.
     * Search for an entity of the specified class and primary key.
     * If the entity instance is contained in the persistence
     * context, it is returned from there.
     * If a vendor-specific property or hint is not recognized,
     * it is silently ignored.
     * @param entityClass  entity class
     * @param primaryKey   primary key
     * @param properties  standard and vendor-specific properties
     *        and hints
```

```
     * @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
     *         is not a valid type for that entity's primary key or
     *         is null
     * @since 2.0
     */
    public <T> T find(Class<T> entityClass, Object primaryKey,
                      Map<String, Object> properties);

    /**
     * Find by primary key and lock.
     * Search for an entity of the specified class and primary key
     * and lock it with respect to the specified lock type.
     * If the entity instance is contained in the persistence context,
     * it is returned from there, and the effect of this method is
     * the same as if the lock method had been called on the entity.
     * <p> If the entity is found within the persistence context and the
     * lock mode type is pessimistic and the entity has a version
     * attribute, the persistence provider must perform optimistic
     * version checks when obtaining the database lock.  If these
     * checks fail, the <code>OptimisticLockException</code> will be thrown.
     * <p>If the lock mode type is pessimistic and the entity instance
     * is found but cannot be locked:
     * <ul>
     * <li> the <code>PessimisticLockException</code> will be thrown if the database
     *    locking failure causes transaction-level rollback
     * <li> the <code>LockTimeoutException</code> will be thrown if the database
     *    locking failure causes only statement-level rollback
     * </ul>
     * @param entityClass  entity class
     * @param primaryKey  primary key
     * @param lockMode  lock mode
     * @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 or
     *         is null
     * @throws TransactionRequiredException if there is no
     *         transaction and a lock mode other than <code>NONE</code> is
     *         specified or if invoked on an entity manager which has
     *         not been joined to the current transaction and a lock
     *         mode other than <code>NONE</code> is specified
     * @throws OptimisticLockException if the optimistic version
     *         check fails
     * @throws PessimisticLockException if pessimistic locking
```

```
     *         fails and the transaction is rolled back
     * @throws LockTimeoutException if pessimistic locking fails and
     *         only the statement is rolled back
     * @throws PersistenceException if an unsupported lock call
     *         is made
     * @since 2.0
     */
    public <T> T find(Class<T> entityClass, Object primaryKey,
                      LockModeType lockMode);

    /**
     * Find by primary key and lock, using the specified properties.
     * Search for an entity of the specified class and primary key
     * and lock it with respect to the specified lock type.
     * If the entity instance is contained in the persistence context,
     * it is returned from there.
     * <p> If the entity is found
     * within the persistence context and the lock mode type
     * is pessimistic and the entity has a version attribute, the
     * persistence provider must perform optimistic version checks
     * when obtaining the database lock.  If these checks fail,
     * the <code>OptimisticLockException</code> will be thrown.
     * <p>If the lock mode type is pessimistic and the entity instance
     * is found but cannot be locked:
     * <ul>
     * <li> the <code>PessimisticLockException</code> will be thrown if the database
     *    locking failure causes transaction-level rollback
     * <li> the <code>LockTimeoutException</code> will be thrown if the database
     *    locking failure causes only statement-level rollback
     * </ul>
     * <p>If a vendor-specific property or hint is not recognized,
     * it is silently ignored.
     * <p>Portable applications should not rely on the standard timeout
     * hint. Depending on the database in use and the locking
     * mechanisms used by the provider, the hint may or may not
     * be observed.
     * @param entityClass  entity class
     * @param primaryKey  primary key
     * @param lockMode  lock mode
     * @param properties  standard and vendor-specific properties
     *        and hints
     * @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 or
     *         is null
     * @throws TransactionRequiredException if there is no
```

```
     *         transaction and a lock mode other than <code>NONE</code> is
     *         specified or if invoked on an entity manager which has
     *         not been joined to the current transaction and a lock
     *         mode other than <code>NONE</code> is specified
     * @throws OptimisticLockException if the optimistic version
     *         check fails
     * @throws PessimisticLockException if pessimistic locking
     *         fails and the transaction is rolled back
     * @throws LockTimeoutException if pessimistic locking fails and
     *         only the statement is rolled back
     * @throws PersistenceException if an unsupported lock call
     *         is made
     * @since 2.0
     */
    public <T> T find(Class<T> entityClass, Object primaryKey,
                      LockModeType lockMode,
                      Map<String, Object> properties);

    /**
     * Get an instance, whose state may be lazily fetched.
     * If the requested instance does not exist in the database,
     * the <code>EntityNotFoundException</code> is thrown when the instance
     * state is first accessed. (The persistence provider runtime is
     * permitted to throw the <code>EntityNotFoundException</code> when
     * <code>getReference</code> 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  entity class
     * @param primaryKey  primary key
     * @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 or
     *         is null
     * @throws EntityNotFoundException if the entity state
     *         cannot be accessed
     */
    public <T> T getReference(Class<T> entityClass,
                              Object primaryKey);

    /**
     * Synchronize the persistence context to the
     * underlying database.
     * @throws TransactionRequiredException if there is
     *         no transaction or if the entity manager has not been
     *         joined to the current transaction
     * @throws PersistenceException if the flush fails
```

```
     */
    public void flush();

    /**
     * Set the flush mode that applies to all objects contained
     * in the persistence context.
     * @param flushMode  flush mode
     */
    public void setFlushMode(FlushModeType flushMode);

    /**
     * Get the flush mode that applies to all objects contained
     * in the persistence context.
     * @return flushMode
     */
    public FlushModeType getFlushMode();

    /**
     * Lock an entity instance that is contained in the persistence
     * context with the specified lock mode type.
     * <p>If a pessimistic lock mode type is specified and the entity
     * contains a version attribute, the persistence provider must
     * also perform optimistic version checks when obtaining the
     * database lock.  If these checks fail, the
     * <code>OptimisticLockException</code> will be thrown.
     * <p>If the lock mode type is pessimistic and the entity instance
     * is found but cannot be locked:
     * <ul>
     * <li> the <code>PessimisticLockException</code> will be thrown if the database
     *    locking failure causes transaction-level rollback
     * <li> the <code>LockTimeoutException</code> will be thrown if the database
     *    locking failure causes only statement-level rollback
     * </ul>
     * @param entity  entity instance
     * @param lockMode  lock mode
     * @throws IllegalArgumentException if the instance is not an
     *         entity or is a detached entity
     * @throws TransactionRequiredException if there is no
     *         transaction or if invoked on an entity manager which
     *         has not been joined to the current transaction
     * @throws EntityNotFoundException if the entity does not exist
     *         in the database when pessimistic locking is
     *         performed
     * @throws OptimisticLockException if the optimistic version
     *         check fails
     * @throws PessimisticLockException if pessimistic locking fails
     *         and the transaction is rolled back
     * @throws LockTimeoutException if pessimistic locking fails and
```

```
     *         only the statement is rolled back
     * @throws PersistenceException if an unsupported lock call
     *         is made
     */
    public void lock(Object entity, LockModeType lockMode);

    /**
     * Lock an entity instance that is contained in the persistence
     * context with the specified lock mode type and with specified
     * properties.
     * <p>If a pessimistic lock mode type is specified and the entity
     * contains a version attribute, the persistence provider must
     * also perform optimistic version checks when obtaining the
     * database lock.  If these checks fail, the
     * <code>OptimisticLockException</code> will be thrown.
     * <p>If the lock mode type is pessimistic and the entity instance
     * is found but cannot be locked:
     * <ul>
     * <li> the <code>PessimisticLockException</code> will be thrown if the database
     *    locking failure causes transaction-level rollback
     * <li> the <code>LockTimeoutException</code> will be thrown if the database
     *    locking failure causes only statement-level rollback
     * </ul>
     * <p>If a vendor-specific property or hint is not recognized,
     * it is silently ignored.
     * <p>Portable applications should not rely on the standard timeout
     * hint. Depending on the database in use and the locking
     * mechanisms used by the provider, the hint may or may not
     * be observed.
     * @param entity  entity instance
     * @param lockMode  lock mode
     * @param properties  standard and vendor-specific properties
     *        and hints
     * @throws IllegalArgumentException if the instance is not an
     *         entity or is a detached entity
     * @throws TransactionRequiredException if there is no
     *         transaction or if invoked on an entity manager which
     *         has not been joined to the current transaction
     * @throws EntityNotFoundException if the entity does not exist
     *         in the database when pessimistic locking is
     *         performed
     * @throws OptimisticLockException if the optimistic version
     *         check fails
     * @throws PessimisticLockException if pessimistic locking fails
     *         and the transaction is rolled back
     * @throws LockTimeoutException if pessimistic locking fails and
     *         only the statement is rolled back
     * @throws PersistenceException if an unsupported lock call
```

```
     *         is made
     * @since 2.0
     */
    public void lock(Object entity, LockModeType lockMode,
                     Map<String, Object> properties);

    /**
     * Refresh the state of the instance from the database,
     * overwriting changes made to the entity, if any.
     * @param entity  entity instance
     * @throws IllegalArgumentException if the instance is not
     *         an entity or the entity is not managed
     * @throws TransactionRequiredException if there is no
     *         transaction when invoked on a container-managed
     *         entity manager of type <code>PersistenceContextType.TRANSACTION</code>
     * @throws EntityNotFoundException if the entity no longer
     *         exists in the database
     */
    public void refresh(Object entity);

    /**
     * Refresh the state of the instance from the database, using
     * the specified properties, and overwriting changes made to
     * the entity, if any.
     * <p> If a vendor-specific property or hint is not recognized,
     * it is silently ignored.
     * @param entity  entity instance
     * @param properties  standard and vendor-specific properties
     *        and hints
     * @throws IllegalArgumentException if the instance is not
     *         an entity or the entity is not managed
     * @throws TransactionRequiredException if there is no
     *         transaction when invoked on a container-managed
     *         entity manager of type <code>PersistenceContextType.TRANSACTION</code>
     * @throws EntityNotFoundException if the entity no longer
     *         exists in the database
     * @since 2.0
     */
    public void refresh(Object entity,
                        Map<String, Object> properties);

    /**
     * Refresh the state of the instance from the database,
     * overwriting changes made to the entity, if any, and
     * lock it with respect to given lock mode type.
     * <p>If the lock mode type is pessimistic and the entity instance
     * is found but cannot be locked:
     * <ul>
```

```
     * <li> the <code>PessimisticLockException</code> will be thrown if the database
     *    locking failure causes transaction-level rollback
     * <li> the <code>LockTimeoutException</code> will be thrown if the
     *    database locking failure causes only statement-level
     *    rollback.
     * </ul>
     * @param entity  entity instance
     * @param lockMode  lock mode
     * @throws IllegalArgumentException if the instance is not
     *         an entity or the entity is not managed
     * @throws TransactionRequiredException if invoked on a
     *         container-managed entity manager of type
     *         <code>PersistenceContextType.TRANSACTION</code> when there is
     *         no transaction; if invoked on an extended entity manager when
     *         there is no transaction and a lock mode other than <code>NONE</code>
     *         has been specified; or if invoked on an extended entity manager
     *         that has not been joined to the current transaction and a
     *         lock mode other than <code>NONE</code> has been specified
     * @throws EntityNotFoundException if the entity no longer exists
     *         in the database
     * @throws PessimisticLockException if pessimistic locking fails
     *         and the transaction is rolled back
     * @throws LockTimeoutException if pessimistic locking fails and
     *         only the statement is rolled back
     * @throws PersistenceException if an unsupported lock call
     *         is made
     * @since 2.0
     */
    public void refresh(Object entity, LockModeType lockMode);

    /**
     * Refresh the state of the instance from the database,
     * overwriting changes made to the entity, if any, and
     * lock it with respect to given lock mode type and with
     * specified properties.
     * <p>If the lock mode type is pessimistic and the entity instance
     * is found but cannot be locked:
     * <ul>
     * <li> the <code>PessimisticLockException</code> will be thrown if the database
     *    locking failure causes transaction-level rollback
     * <li> the <code>LockTimeoutException</code> will be thrown if the database
     *    locking failure causes only statement-level rollback
     * </ul>
     * <p>If a vendor-specific property or hint is not recognized,
     *    it is silently ignored.
     * <p>Portable applications should not rely on the standard timeout
     * hint. Depending on the database in use and the locking
     * mechanisms used by the provider, the hint may or may not
```

```
     * be observed.
     * @param entity  entity instance
     * @param lockMode  lock mode
     * @param properties  standard and vendor-specific properties
     *        and hints
     * @throws IllegalArgumentException if the instance is not
     *         an entity or the entity is not managed
     * @throws TransactionRequiredException if invoked on a
     *         container-managed entity manager of type
     *         <code>PersistenceContextType.TRANSACTION</code> when there is
     *         no transaction; if invoked on an extended entity manager when
     *         there is no transaction and a lock mode other than <code>NONE</code>
     *         has been specified; or if invoked on an extended entity manager
     *         that has not been joined to the current transaction and a
     *         lock mode other than <code>NONE</code> has been specified
     * @throws EntityNotFoundException if the entity no longer exists
     *         in the database
     * @throws PessimisticLockException if pessimistic locking fails
     *         and the transaction is rolled back
     * @throws LockTimeoutException if pessimistic locking fails and
     *         only the statement is rolled back
     * @throws PersistenceException if an unsupported lock call
     *         is made
     * @since 2.0
     */
    public void refresh(Object entity, LockModeType lockMode,
                        Map<String, Object> properties);

    /**
     * Clear the persistence context, causing all managed
     * entities to become detached. Changes made to entities that
     * have not been flushed to the database will not be
     * persisted.
     */
    public void clear();

    /**
     * Remove the given entity from the persistence context, causing
     * a managed entity to become detached.  Unflushed changes made
     * to the entity if any (including removal of the entity),
     * will not be synchronized to the database.  Entities which
     * previously referenced the detached entity will continue to
     * reference it.
     * @param entity  entity instance
     * @throws IllegalArgumentException if the instance is not an
     *         entity
     * @since 2.0
     */
```

```
    public void detach(Object entity);

    /**
     * Check if the instance is a managed entity instance belonging
     * to the current persistence context.
     * @param entity  entity instance
     * @return boolean indicating if entity is in persistence context
     * @throws IllegalArgumentException if not an entity
     */
    public boolean contains(Object entity);

    /**
     * Get the current lock mode for the entity instance.
     * @param entity  entity instance
     * @return lock mode
     * @throws TransactionRequiredException if there is no
     *         transaction or if the entity manager has not been
     *         joined to the current transaction
     * @throws IllegalArgumentException if the instance is not a
     *         managed entity and a transaction is active
     * @since 2.0
     */
    public LockModeType getLockMode(Object entity);

    /**
     * Set an entity manager property or hint.
     * If a vendor-specific property or hint is not recognized, it is
     * silently ignored.
     * @param propertyName name of property or hint
     * @param value  value for property or hint
     * @throws IllegalArgumentException if the second argument is
     *         not valid for the implementation
     * @since 2.0
     */
    public void setProperty(String propertyName, Object value);

    /**
     * Get the properties and hints and associated values that are in effect
     * for the entity manager. Changing the contents of the map does
     * not change the configuration in effect.
     * @return map of properties and hints in effect for entity manager
     * @since 2.0
     */
    public Map<String, Object> getProperties();

    /**
     * Create an instance of <code>Query</code> for executing a
     * Jakarta Persistence query language statement.
```

```
     * @param qlString a Jakarta Persistence query string
     * @return the new query instance
     * @throws IllegalArgumentException if the query string is
     *         found to be invalid
     */
    public Query createQuery(String qlString);

    /**
     * Create an instance of <code>TypedQuery</code> for executing a
     * criteria query.
     * @param criteriaQuery  a criteria query object
     * @return the new query instance
     * @throws IllegalArgumentException if the criteria query is
     *         found to be invalid
     * @since 2.0
     */
    public <T> TypedQuery<T> createQuery(CriteriaQuery<T> criteriaQuery);

    /**
     * Create an instance of <code>Query</code> for executing a criteria
     * update query.
     * @param updateQuery  a criteria update query object
     * @return the new query instance
     * @throws IllegalArgumentException if the update query is
     *         found to be invalid
     * @since 2.1
     */
    public Query createQuery(CriteriaUpdate updateQuery);

    /**
     * Create an instance of <code>Query</code> for executing a criteria
     * delete query.
     * @param deleteQuery  a criteria delete query object
     * @return the new query instance
     * @throws IllegalArgumentException if the delete query is
     *         found to be invalid
     * @since 2.1
     */
    public Query createQuery(CriteriaDelete deleteQuery);

    /**
     * Create an instance of <code>TypedQuery</code> for executing a
     * Jakarta Persistence query language statement.
     * The select list of the query must contain only a single
     * item, which must be assignable to the type specified by
     * the <code>resultClass</code> argument.
     * @param qlString a Jakarta Persistence query string
     * @param resultClass the type of the query result
```

```
     * @return the new query instance
     * @throws IllegalArgumentException if the query string is found
     *         to be invalid or if the query result is found to
     *         not be assignable to the specified type
     * @since 2.0
     */
    public <T> TypedQuery<T> createQuery(String qlString, Class<T> resultClass);

    /**
     * Create an instance of <code>Query</code> for executing a named query
     * (in the Jakarta Persistence query language or in native SQL).
     * @param name the name of a query defined in metadata
     * @return the new query instance
     * @throws IllegalArgumentException if a query has not been
     *         defined with the given name or if the query string is
     *         found to be invalid
     */
    public Query createNamedQuery(String name);

    /**
     * Create an instance of <code>TypedQuery</code> for executing a
     * Jakarta Persistence query language named query.
     * The select list of the query must contain only a single
     * item, which must be assignable to the type specified by
     * the <code>resultClass</code> argument.
     * @param name the name of a query defined in metadata
     * @param resultClass the type of the query result
     * @return the new query instance
     * @throws IllegalArgumentException if a query has not been
     *         defined with the given name or if the query string is
     *         found to be invalid or if the query result is found to
     *         not be assignable to the specified type
     * @since 2.0
     */
    public <T> TypedQuery<T> createNamedQuery(String name, Class<T> resultClass);

    /**
     * Create an instance of <code>Query</code> for executing
     * a native SQL statement, e.g., for update or delete.
     * If the query is not an update or delete query, query
     * execution will result in each row of the SQL result
     * being returned as a result of type Object[] (or a result
     * of type Object if there is only one column in the select
     * list.)  Column values are returned in the order of their
     * appearance in the select list and default JDBC type
     * mappings are applied.
     * @param sqlString a native SQL query string
     * @return the new query instance
```

```
     */
    public Query createNativeQuery(String sqlString);

    /**
     * Create an instance of <code>Query</code> for executing
     * a native SQL query.
     * @param sqlString a native SQL query string
     * @param resultClass the class of the resulting instance(s)
     * @return the new query instance
     */
    public Query createNativeQuery(String sqlString, Class resultClass);

    /**
     * Create an instance of <code>Query</code> 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
     */
    public Query createNativeQuery(String sqlString, String resultSetMapping);

    /**
     * Create an instance of <code>StoredProcedureQuery</code> for executing a
     * stored procedure in the database.
     * <p>Parameters must be registered before the stored procedure can
     * be executed.
     * <p>If the stored procedure returns one or more result sets,
     * any result set will be returned as a list of type Object[].
     * @param name name assigned to the stored procedure query
     * in metadata
     * @return the new stored procedure query instance
     * @throws IllegalArgumentException if a query has not been
     * defined with the given name
     * @since 2.1
     */
    public StoredProcedureQuery createNamedStoredProcedureQuery(String name);

    /**
     * Create an instance of <code>StoredProcedureQuery</code> for executing a
     * stored procedure in the database.
     * <p>Parameters must be registered before the stored procedure can
     * be executed.
     * <p>If the stored procedure returns one or more result sets,
     * any result set will be returned as a list of type Object[].
     * @param procedureName name of the stored procedure in the
     * database
     * @return the new stored procedure query instance
     * @throws IllegalArgumentException if a stored procedure of the
```

```
     * given name does not exist (or the query execution will
     * fail)
     * @since 2.1
     */
    public StoredProcedureQuery createStoredProcedureQuery(String procedureName);

    /**
     * Create an instance of <code>StoredProcedureQuery</code> for executing a
     * stored procedure in the database.
     * <p>Parameters must be registered before the stored procedure can
     * be executed.
     * <p>The <code>resultClass</code> arguments must be specified in the order in
     * which the result sets will be returned by the stored procedure
     * invocation.
     * @param procedureName name of the stored procedure in the
     * database
     * @param resultClasses classes to which the result sets
     * produced by the stored procedure are to
     * be mapped
     * @return the new stored procedure query instance
     * @throws IllegalArgumentException if a stored procedure of the
     * given name does not exist (or the query execution will
     * fail)
     * @since 2.1
     */
    public StoredProcedureQuery createStoredProcedureQuery(
           String procedureName, Class... resultClasses);

    /**
     * Create an instance of <code>StoredProcedureQuery</code> for executing a
     * stored procedure in the database.
     * <p>Parameters must be registered before the stored procedure can
     * be executed.
     * <p>The <code>resultSetMapping</code> arguments must be specified in the order
     * in which the result sets will be returned by the stored
     * procedure invocation.
     * @param procedureName name of the stored procedure in the
     *        database
     * @param resultSetMappings the names of the result set mappings
     *        to be used in mapping result sets
     *        returned by the stored procedure
     * @return the new stored procedure query instance
     * @throws IllegalArgumentException if a stored procedure or
     *         result set mapping of the given name does not exist
     *         (or the query execution will fail)
     */
    public StoredProcedureQuery createStoredProcedureQuery(
             String procedureName, String... resultSetMappings);
```

```
    /**
     * Indicate to the entity manager that a JTA transaction is
     * active and join the persistence context to it.
     * <p>This method should be called on a JTA application
     * managed entity manager that was created outside the scope
     * of the active transaction or on an entity manager of type
     * <code>SynchronizationType.UNSYNCHRONIZED</code> to associate
     * it with the current JTA transaction.
     * @throws TransactionRequiredException if there is
     *         no transaction
     */
    public void joinTransaction();

    /**
     * Determine whether the entity manager is joined to the
     * current transaction. Returns false if the entity manager
     * is not joined to the current transaction or if no
     * transaction is active
     * @return boolean
     * @since 2.1
     */
    public boolean isJoinedToTransaction();

    /**
     * Return an object of the specified type to allow access to the
     * provider-specific API.   If the provider's <code>EntityManager</code>
     * implementation does not support the specified class, the
     * <code>PersistenceException</code> is thrown.
     * @param cls  the class of the object to be returned.  This is
     * normally either the underlying <code>EntityManager</code> implementation
     * class or an interface that it implements.
     * @return an instance of the specified class
     * @throws PersistenceException if the provider does not
     *         support the call
     * @since 2.0
     */
    public <T> T unwrap(Class<T> cls);

    /**
     * Return the underlying provider object for the <code>EntityManager</code>,
     * if available. The result of this method is implementation
     * specific.
     * <p>The <code>unwrap</code> method is to be preferred for new applications.
     * @return underlying provider object for EntityManager
     */
    public Object getDelegate();
```

```
    /**
     * Close an application-managed entity manager.
     * After the close method has been invoked, all methods
     * on the <code>EntityManager</code> instance and any
     * <code>Query</code>, <code>TypedQuery</code>, and
     * <code>StoredProcedureQuery</code> objects obtained from
     * it will throw the <code>IllegalStateException</code>
     * except for <code>getProperties</code>,
     * <code>getTransaction</code>, and <code>isOpen</code> (which will return false).
     * If this method is called when the entity manager is
     * joined to an active transaction, the persistence
     * context remains managed until the transaction completes.
     * @throws IllegalStateException if the entity manager
     *         is container-managed
     */
    public void close();

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

    /**
     * Return the resource-level <code>EntityTransaction</code> object.
     * The <code>EntityTransaction</code> instance may be used serially to
     * begin and commit multiple transactions.
     * @return EntityTransaction instance
     * @throws IllegalStateException if invoked on a JTA
     *         entity manager
     */
    public EntityTransaction getTransaction();

    /**
     * Return the entity manager factory for the entity manager.
     * @return EntityManagerFactory instance
     * @throws IllegalStateException if the entity manager has
     *         been closed
     * @since 2.0
     */
    public EntityManagerFactory getEntityManagerFactory();

    /**
     * Return an instance of <code>CriteriaBuilder</code> for the creation of
     * <code>CriteriaQuery</code> objects.
     * @return CriteriaBuilder instance
     * @throws IllegalStateException if the entity manager has
     *         been closed
```

```
     * @since 2.0
     */
    public CriteriaBuilder getCriteriaBuilder();

    /**
     * Return an instance of <code>Metamodel</code> interface for access to the
     * metamodel of the persistence unit.
     * @return Metamodel instance
     * @throws IllegalStateException if the entity manager has
     *         been closed
     * @since 2.0
     */
    public Metamodel getMetamodel();

    /**
     * Return a mutable EntityGraph that can be used to dynamically create an
     * EntityGraph.
     * @param rootType class of entity graph
     * @return entity graph
     * @since 2.1
     */
    public <T> EntityGraph<T> createEntityGraph(Class<T> rootType);

    /**
     * Return a mutable copy of the named EntityGraph.  If there
     * is no entity graph with the specified name, null is returned.
     * @param graphName name of an entity graph
     * @return entity graph
     * @since 2.1
     */
    public EntityGraph<?> createEntityGraph(String graphName);

    /**
     * Return a named EntityGraph. The returned EntityGraph
     * should be considered immutable.
     * @param graphName  name of an existing entity graph
     * @return named entity graph
     * @throws IllegalArgumentException if there is no EntityGraph of
     *         the given name
     * @since 2.1
     */
    public EntityGraph<?> getEntityGraph(String graphName);

    /**
     * Return all named EntityGraphs that have been defined for the provided
     * class type.
     * @param entityClass  entity class
     * @return list of all entity graphs defined for the entity
```

```
     * @throws IllegalArgumentException if the class is not an entity
     * @since 2.1
     */
    public <T> List<EntityGraph<? super T>> getEntityGraphs(Class<T> entityClass);

}
```

The semantics of

```
public <T> TypedQuery<T> createQuery(String qlString, Class<T>
resultClass)
```

method may be extended in a future release of this specification to support other result types. Applications that specify other result types (e.g., Tuple.class) will not be portable.

The semantics

```
public <T> TypedQuery<T> createNamedQuery(String name, Class<T>
resultClass)
```

method may be extended in a future release of this specification to support other result types. Applications that specify other result types (e.g., Tuple.class) will not be portable.

The *persist*, *merge*, *remove*, and *refresh* methods must be invoked within a transaction context when an entity manager with a transaction-scoped persistence context is used. If there is no transaction context, the *javax.persistence.TransactionRequiredException* is thrown.

Methods that specify a lock mode other than *LockModeType.NONE* must be invoked within a transaction. If there is no transaction or if the entity manager has not been joined to the transaction, the *javax.persistence.TransactionRequiredException* is thrown.

The *find* method (provided it is invoked without a lock or invoked with *LockModeType.NONE*) and the *getReference* method are not required to be invoked within a transaction. If an entity manager with transaction-scoped persistence context is in use, the resulting entities will be detached; if an entity manager with an extended persistence context is used, they will be managed. See Section 3.3 for entity manager use outside a transaction.

The *Query*, *TypedQuery*, *StoredProcedureQuery*, *CriteriaBuilder*, *Metamodel*, and *EntityTransaction* objects obtained from an entity manager are valid while that entity manager is open.

If the argument to the *createQuery* method is not a valid Java Persistence query string or a valid

CriteriaQuery object, the *IllegalArgumentException* may be thrown or the query execution will fail and a *PersistenceException* will be thrown. If the result class specification of a Java Persistence query language query is incompatible with the result of the query, the *IllegalArgumentException* may be thrown when the *createQuery* method is invoked or the query execution will fail and a *PersistenceException* will be thrown when the query is executed. If a native query is not a valid query for the database in use or if the result set specification is incompatible with the result of the query, the query execution will fail and a *PersistenceException* will be thrown when the query is executed. The *PersistenceException* should wrap the underlying database exception when possible.

Runtime exceptions thrown by the methods of the *EntityManager* interface other than the *LockTimeoutException* will cause the current transaction to be marked for rollback if the persistence context is joined to that transaction.

The methods *close*, *isOpen*, *joinTransaction*, and *getTransaction* are used to manage application-managed entity managers and their lifecycle. See Section 7.2.2.

The *EntityManager* interface and other interfaces defined by this specification contain methods that take properties and/or hints as arguments. This specification distinguishes between *properties* and *hints* as follows:

- A property defined by this specification must be observed by the provider unless otherwise explicitly stated.
- A hint specifies a preference on the part of the application. While a hint defined by this specification should be observed by the provider if possible, a hint may or may not always be observed. A portable application must not depend on the observance of a hint.

3.1.2. Example of Use of EntityManager API

```
@Stateless
public class OrderEntryBean implements OrderEntry {
    @PersistenceContext
    EntityManager em;

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

3.2. Entity Instance's Life Cycle

This section describes the *EntityManager* operations for managing an entity instance's lifecycle. An

entity instance can 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 will be removed from the database upon transaction commit.

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. 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.

3.2.2. 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 are 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, the *EntityExistsException* may be thrown when the persist operation is invoked, or the *EntityExistsException* or another *PersistenceException may* be thrown at flush or commit time.
- 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.

3.2.3. 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 relationship 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.

After an entity has been removed, its state (except for generated state) will be that of the entity at the point at which the remove operation was called.

3.2.4. Synchronization to the Database

In general, a persistence context will be synchronized to the database as described below. However, a persistence context of type *SynchronizationType.UNSYNCHRONIZED* or an application-managed persistence context that has been created outside the scope of the current transaction will only be synchronized to the database if it has been joined to the current transaction by the application's use of the *EntityManager joinTransaction* method.

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

An update to the state of an entity includes both the assignment of a new value to a persistent property or field of the entity as well as the modification of a mutable value of a persistent property or field [26: This includes, for example. modifications to persistent attributes of type char[] and byte[].].

Synchronization to the database does not involve a refresh of any managed entities unless the *refresh* operation is explicitly invoked on those entities or cascaded to them as a result of the specification of the *cascade=REFRESH* or *cascade=ALL* annotation element value.

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. In the case of unidirectional one-to-one and one-to-many relationships, it is the developer's responsibility to insure that the semantics of the relationships are adhered to. [27: This might be an issue if unique constraints (such as those described for the default mappings in Section 2.10.3.1 and Section 2.10.5.1) were not applied in the definition of the object/relational mapping.]

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.

The persistence provider runtime is permitted to perform synchronization to the database at other times as well when a transaction is active and the persistence context is joined to the transaction. The *flush* method can be used by the application to force synchronization. It applies to entities associated with the persistence context. The *setFlushMode* methods of the *EntityManager*, *Query*, *TypedQuery*, and *StoredProcedureQuery* interfaces can be used to control synchronization semantics. The effect of *FlushModeType.AUTO* is defined in Section 3.10.8. If *FlushModeType.COMMIT* is specified, flushing will occur at transaction commit; the persistence provider is permitted, but not required, to perform to flush at other times. If there is no transaction active or if the persistence context has not been joined to the current transaction, the persistence provider must not flush to the database.

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 marked for rollback) 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 synchronized 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.

3.2.5. Refreshing an Entity Instance

The state of a managed entity instance is refreshed from the database by invoking the *refresh* method on it or by cascading the refresh operation.

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

- If X is a managed entity, the state of X is refreshed from the database, overwriting changes made to the entity, if any. The refresh operation is cascaded to entities referenced by X if the relationship from X to these other entities is annotated with the *cascade=REFRESH* or *cascade=ALL* annotation

element value.

- If X is a new, detached, or removed entity, the *IllegalArgumentException* is thrown.

3.2.6. Evicting an Entity Instance from the Persistence Context

An entity instance is removed from the persistence context by invoking the *detach* method on it or cascading the detach operation. Changes made to the entity, if any (including removal of the entity), will not be synchronized to the database after such eviction has taken place.

Applications must use the *flush* method prior to the *detach* method to ensure portable semantics if changes have been made to the entity (including removal of the entity). Because the persistence provider may write to the database at times other than the explicit invocation of the *flush* method, portable applications must not assume that changes have not been written to the database if the *flush* method has not been called prior to detach.

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

- If X is a managed entity, the detach operation causes it to become detached. The detach operation is cascaded to entities referenced by X if the relationships from X to these other entities is annotated with the *cascade=DETACH* or *cascade=ALL* annotation element value. Entities which previously referenced X will continue to reference X.
- If X is a new or detached entity, it is ignored by the detach operation.
- If X is a removed entity, the detach operation causes it to become detached. The detach operation is cascaded to entities referenced by X if the relationships from X to these other entities is annotated with the *cascade=DETACH* or c *ascade=ALL* annotation element value. Entities which previously referenced X will continue to reference X. Portable applications should not pass removed entities that have been detached from the persistence context to further EntityManager operations.

3.2.7. Detached Entities

A detached entity results from transaction commit if a transaction-scoped persistence context is used (see Section 3.3); from transaction rollback (see Section 3.3.3); from detaching the entity from the persistence context; from clearing the persistence context; from closing an entity manager; or from serializing an entity or otherwise passing an entity by value—e.g., to a separate application tier, through a remote interface, etc.

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

The application may 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 or fetched by means of an entity graph

If the persistent field or property is an association, the available 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.
- Any entity instance for which an instance variable holding non-primary-key persistent state was accessed by the application.
- Any entity instance that can be reached from another available instance by navigating associations marked *fetch=EAGER*.

3.2.7.1. Merging Detached Entity State

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

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

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

The persistence provider must not merge fields marked LAZY that have not been fetched: it must ignore such fields when merging.

Any *Version* columns used by the entity must be checked by the persistence runtime implementation during the merge operation and/or at flush or commit time. In the absence of *Version* columns there is no additional version checking done by the persistence provider runtime during the merge operation.

3.2.7.2. Detached Entities and Lazy Loading

Serializing entities and merging those entities back into a persistence context may not be interoperable

across vendors when lazy properties or fields and/or relationships are used.

A vendor is required to support the serialization and subsequent deserialization and merging of detached entity instances (which may contain lazy properties or fields and/or relationships that have not been fetched) back into a separate JVM instance of that vendor's runtime, where both runtime instances have access to the entity classes and any required vendor persistence implementation classes.

When interoperability across vendors is required, the application must not use lazy loading.

3.2.8. Managed Instances

It is the responsibility of the application to insure that an instance is managed in only a single persistence context. The behavior is undefined if the same Java instance is made managed in more than one persistence context.

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 or has been returned by *getReference*, 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, merge, remove, or detach is immediately visible to the *contains* method, whereas the actual insertion, modification, or deletion of the database representation for the entity may be deferred until the end of the transaction.

3.2.9. Load State

An entity is considered to be loaded if all attributes with *FetchType.EAGER* —whether explictly specified or by default—(including relationship and other collection-valued attributes) have been loaded from the database or assigned by the application. Attributes with *FetchType.LAZY* may or may not have been loaded. The available state of the entity instance and associated instances is as described in Section 3.2.7.

An attribute that is an embeddable is considered to be loaded if the embeddable attribute was loaded from the database or assigned by the application, and, if the attribute references an embeddable instance (i.e., is not null), the embeddable instance state is known to be loaded (i.e., all attributes of the embeddable with *FetchType.EAGER* have been loaded from the database or assigned by the application).

A collection-valued attribute is considered to be loaded if the collection was loaded from the database or the value of the attribute was assigned by the application, and, if the attribute references a collection instance (i.e., is not null), each element of the collection (e.g. entity or embeddable) is considered to be loaded.

A single-valued relationship attribute is considered to be loaded if the relationship attribute was loaded from the database or assigned by the application, and, if the attribute references an entity instance (i.e., is not null), the entity instance state is known to be loaded.

A basic attribute is considered to be loaded if its state has been loaded from the database or assigned by the application.

The *PersistenceUtil.isLoaded* methods can be used to determine the load state of an entity and its attributes regardless of the persistence unit with which the entity is associated. The *PersistenceUtil.isLoaded* methods return true if the above conditions hold, and false otherwise. If the persistence unit is known, the *PersistenceUnitUtil.isLoaded* methods can be used instead. See Section 7.11.

Persistence provider contracts for determining the load state of an entity or entity attribute are described in Section 9.8.1.

3.3. Persistence Context Lifetime and Synchronization Type

The lifetime of a container-managed persistence context can either be scoped to a transaction (transaction-scoped persistence context), or have a lifetime scope that extends beyond that of a single transaction (extended persistence context). The enum *PersistenceContextType* is used to define the persistence context lifetime scope for container-managed entity managers. The persistence context lifetime scope is defined when the EntityManager instance is created (whether explicitly, or in conjunction with injection or JNDI lookup). See Section 7.6.

```
package javax.persistence;

public enum PersistenceContextType {
    TRANSACTION,
    EXTENDED
}
```

By default, the lifetime of the persistence context of a container-managed entity manager corresponds to the scope of a transaction (i.e., it is of type *PersistenceContextType.TRANSACTION*).

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, and they can be updated as managed objects between transactions. [28: Note that when a new transaction is begun, the managed objects in an extended persistence context are *not* reloaded from the database.] Navigation from a managed object in an extended persistence context results in one or more other managed objects regardless of whether a transaction is active.

When an EntityManager with an extended persistence context is used, the persist, remove, merge, and refresh operations can be called regardless of whether a transaction is active. The effects of these operations will be committed to the database when the extended persistence context is enlisted in a transaction and the transaction commits.

The scope of the persistence context of an application-managed entity manager is extended. It is the responsibility of the application to manage the lifecycle of the persistence context.

Container-managed persistence contexts are described further in Section 7.6. Persistence contexts managed by the application are described further in Section 7.7.

3.3.1. Synchronization with the Current Transaction

By default, a container-managed persistence context is of *SynchronizationType.SYNCHRONIZED* and is automatically joined to the current transaction. A persistence context of *SynchronizationType.UNSYNCHRONIZED* will not be enlisted in the current transaction, unless the *EntityManager joinTransaction* method is invoked.

By default, an application-managed persistence context that is associated with a JTA entity manager and that is created within the scope of an active transaction is automatically joined to that transaction. An application-managed JTA persistence context that is created outside the scope of a transaction or an application-managed persistence context of type *SynchronizationType.UNSYNCHRONIZED* will not be joined to that transaction unless the *EntityManager joinTransaction* method is invoked.

An application-managed persistence context associated with a resource-local entity manager is always automatically joined to any resource-local transaction that is begun for that entity manager.

Persistence context synchronization type is described further in Section 7.6.1.

3.3.2. Transaction Commit

The managed entities of a transaction-scoped persistence context become detached when the transaction commits; the managed entities of an extended persistence context remain managed.

3.3.3. Transaction Rollback

For both transaction-scoped persistence contexts and for extended persistence contexts that are joined to the current transaction, transaction rollback causes all *pre-existing* managed instances and removed instances [29: These are instances that were persistent in the database at the start of the transaction.] to become detached. The instances' state will be the state of the instances at the point at which the transaction was rolled back. Transaction rollback typically causes the persistence context to be in an inconsistent state at the point of rollback. In particular, the state of version attributes and generated state (e.g., generated primary keys) may be inconsistent. Instances that were formerly managed by the persistence context (including new instances that were made persistent in that transaction) may therefore not be reusable in the same manner as other detached objects—for example, they may fail when passed to the merge operation. [30: It is unspecified as to whether instances that were not persistent in the database behave as new instances or detached instances after rollback. This may be implementation-dependent.]

Because a transaction-scoped persistence context's lifetime is scoped to a transaction regardless of whether it is joined to that transaction, the container closes the persistence context upon transaction rollback. However, an extended persistence context that is not joined to a transaction is unaffected by transaction rollback.

3.4. Locking and Concurrency

This specification assumes the use of optimistic concurrency control. It assumes that the databases to which persistence units are mapped will be accessed by the implementation using read-committed isolation (or a vendor equivalent in which long-term read locks are not held), and that writes to the database will typically occur only when the *flush* method has been invoked—whether explicitly by the application, or by the persistence provider runtime in accordance with the flush mode setting.

If a transaction is active and the persistence context is joined to the transaction, a compliant implementation of this specification is permitted to write to the database immediately (i.e., whenever a managed entity is updated, created, and/or removed), however, the configuration of an implementation to require such non-deferred database writes is outside the scope of this specification. [32: Applications may require that database isolation levels higher than read-committed be in effect. The configuration of the setting database isolation levels, however, is outside the scope of this specification.]

In addition, both pessimistic and optimistic locking are supported for selected entities by means of specified lock modes. Optimistic locking is described in Section 3.4.1 and Section 3.4.2; pessimistic locking in Section 3.4.3. Section 3.4.4 describes the setting of optimistic and pessimistic lock modes. The configuration of the setting of optimistic lock modes is described in Section 3.4.4.1, and the configuration of the setting of pessimistic lock modes is described in Section 3.4.4.2.

3.4.1. Optimistic Locking

Optimistic locking is a technique that is used to insure that updates to the database data corresponding to the state of an entity are made only when no intervening transaction has updated that data since the entity state was read. This insures that updates or deletes to that data are consistent with the current state of the database and that intervening updates are not lost. Transactions that would cause this constraint to be violated result in an *OptimisticLockException* being thrown and the transaction marked for rollback.

Portable applications that wish to enable optimistic locking for entities must specify *Version* attributes for those entities—i.e., persistent properties or fields annotated with the *Version* annotation or specified in the XML descriptor as version attributes. Applications are strongly encouraged to enable optimistic locking for all entities that may be concurrently accessed or that may be merged from a disconnected state. Failure to use optimistic locking may lead to inconsistent entity state, lost updates and other state irregularities. If optimistic locking is not defined as part of the entity state, the application must bear the burden of maintaining data consistency.

3.4.2. Version Attributes

The *Version* field or property is used by the persistence provider to perform optimistic locking. It is accessed and/or set by the persistence provider in the course of performing lifecycle operations on the entity instance. An entity is automatically enabled for optimistic locking if it has a property or field mapped with a *Version* mapping.

An entity may access the state of its version field or property or export a method for use by the application to access the version, but must not modify the version value. [33: Bulk update statements, however, are permitted to set the value of version attributes. See Section 4.10.] With the exception noted in Section 4.10, only the persistence provider is permitted to set or update the value of the version attribute in the object.

The version attribute is updated by the persistence provider runtime when the object is written to the database. All non-relationship fields and properties and all relationships owned by the entity are included in version checks. [34: This includes owned relationships maintained in join tables.]

The persistence provider's implementation of the merge operation must examine the version attribute when an entity is being merged and throw an *OptimisticLockException* if it is discovered that the object being merged is a stale copy of the entity—i.e. that the entity has been updated since the entity became detached. Depending on the implementation strategy used, it is possible that this exception may not be thrown until *flush* is called or commit time, whichever happens first.

The persistence provider runtime is required to use only the version attribute when performing optimistic lock checking. Persistence provider implementations may provide additional mechanisms beside version attributes to enable optimistic lock checking. However, support for such mechanisms is not required of an implementation of this specification. [35: Such additional mechanisms may be standardized by a future release of this specification.]

If only some entities contain version attributes, the persistence provider runtime is required to check those entities for which version attributes have been specified. The consistency of the object graph is not guaranteed, but the absence of version attributes on some of the entities will not stop operations from completing.

3.4.3. Pessimistic Locking

While optimistic locking is typically appropriate in dealing with moderate contention among concurrent transactions, in some applications it may be useful to immediately obtain long-term database locks for selected entities because of the often late failure of optimistic transactions. Such immediately obtained long-term database locks are referred to here as "pessimistic" locks. [36: Implementations are permitted to use database mechanisms other than locking to achieve the semantic effects described here, for example, multiversion concurrency control mechanisms.]

Pessimistic locking guarantees that once a transaction has obtained a pessimistic lock on an entity instance:

- no other transaction (whether a transaction of an application using the Java Persistence API or any other transaction using the underlying resource) may successfully modify or delete that instance until the transaction holding the lock has ended.
- if the pessimistic lock is an exclusive lock [37: This is achieved by using a lock with LockModeType.PESSIMISTIC_WRITE or LockModeType.PESSIMISTIC_FORCE_INCREMENT as described in Section 3.4.4.], that same transaction may modify or delete that entity instance.

When an entity instance is locked using pessimistic locking, the persistence provider must lock the database row(s) that correspond to the non-collection-valued persistent state of that instance. If a joined inheritance strategy is used, or if the entity is otherwise mapped to a secondary table, this entails locking the row(s) for the entity instance in the additional table(s). Entity relationships for which the locked entity contains the foreign key will also be locked, but not the state of the referenced entities (unless those entities are explicitly locked). Element collections and relationships for which the entity does not contain the foreign key (such as relationships that are mapped to join tables or unidirectional one-to-many relationships for which the target entity contains the foreign key) will not be locked by default.

Element collections and relationships owned by the entity that are contained in join tables will be locked if the *javax.persistence.lock.scope* property is specified with a value of *PessimisticLockScope.EXTENDED*. The state of entities referenced by such relationships will not be locked (unless those entities are explicitly locked). This property may be passed as an argument to the methods of the *EntityManager*, *Query*, and *TypedQuery* interfaces that allow lock modes to be specified or used with the *NamedQuery* annotation.

Locking such a relationship or element collection generally locks only the rows in the join table or collection table for that relationship or collection. This means that phantoms will be possible.

The values of the *javax.persistence.lock.scope* property are defined by the *PessimisticLockScope* enum.

```
package javax.persistence;

public enum PessimisticLockScope {
  NORMAL,
  EXTENDED
}
```

This specification does not define the mechanisms a persistence provider uses to obtain database locks, and a portable application should not rely on how pessimistic locking is achieved on the database. [38: For example, a persistence provider may use an underlying database platform's SELECT FOR UPDATE statements to implement pessimistic locking if that construct provides appropriate semantics, or the provider may use an isolation level of repeatable read.] In particular, a persistence provider or the underlying database management system may lock more rows than the ones selected by the application.

Whenever a pessimistically locked entity containing a version attribute is updated on the database, the persistence provider must also update (increment) the entity's version column to enable correct interaction with applications using optimistic locking. See Section 3.4.2 and Section 3.4.4.

Pessimistic locking may be applied to entities that do not contain version attributes. However, in this case correct interaction with applications using optimistic locking cannot be ensured.

3.4.4. Lock Modes

Lock modes are intended to provide a facility that enables the effect of "repeatable read" semantics for the items read, whether "optimistically" (as described in Section 3.4.4.1) or "pessimistically" (as described in Section 3.4.4.2).

Lock modes can be specified by means of the EntityManager *lock* method, the methods of the *EntityManager*, *Query*, and *TypedQuery* interfaces that allow lock modes to be specified, and the *NamedQuery* annotation.

Lock mode values are defined by the *LockModeType* enum. Six distinct lock modes are defined. The lock mode type values *READ* and *WRITE* are synonyms of *OPTIMISTIC* and *OPTIMISTIC_FORCE_INCREMENT* respectively. [39: The lock mode type NONE may be specified as a value of lock mode arguments and also provides a default value for annotations.] The latter are to be preferred for new applications.

```
package javax.persistence;

public enum LockModeType {
  READ,
  WRITE,
  OPTIMISTIC,
  OPTIMISTIC_FORCE_INCREMENT,
  PESSIMISTIC_READ,
  PESSIMISTIC_WRITE,
  PESSIMISTIC_FORCE_INCREMENT,
  NONE
}
```

3.4.4.1. OPTIMISTIC, OPTIMISTIC_FORCE_INCREMENT

The lock modes *OPTIMISTIC* and *OPTIMISTIC_FORCE_INCREMENT* are used for optimistic locking. The lock mode type values *READ* and *WRITE* are synonymous with *OPTIMISTIC* and *OPTIMISTIC_FORCE_INCREMENT* respectively.

The semantics of requesting locks of type *LockModeType.OPTIMISTIC* and *LockModeType.OPTIMISTIC_FORCE_INCREMENT* are the following.

If transaction T1 calls *lock(entity, LockModeType.OPTIMISTIC)* on a versioned object, the entity manager must ensure that neither of the following phenomena can occur:

- P1 (Dirty read): Transaction T1 modifies a row. Another transaction T2 then reads that row and obtains the modified value, before T1 has committed or rolled back. Transaction T2 eventually commits successfully; it does not matter whether T1 commits or rolls back and whether it does so before or after T2 commits.
- P2 (Non-repeatable read): Transaction T1 reads a row. Another transaction T2 then modifies or deletes that row, before T1 has committed. Both transactions eventually commit successfully.

This will generally be achieved by the entity manager acquiring a lock on the underlying database row. While with optimistic concurrency concurrency, long-term database read locks are typically not obtained immediately, a compliant implementation is permitted to obtain an immediate lock (so long as it is retained until commit completes). If the lock is deferred until commit time, it must be retained until the commit completes. Any implementation that supports repeatable reads in a way that prevents the above phenomena is permissible.

The persistence implementation is not required to support calling *lock(entity, LockModeType.OPTIMISTIC)* on a non-versioned object. When it cannot support such a lock call, it must throw the *PersistenceException*. When supported, whether for versioned or non-versioned objects, *LockModeType.OPTIMISTIC* must always prevent the phenomena P1 and P2. Applications that call *lock(entity, LockModeType.OPTIMISTIC)* on non-versioned objects will not be portable.

If transaction T1 calls *lock(entity, LockModeType.OPTIMISTIC_FORCE_INCREMENT)* on a versioned object, the entity manager must avoid the phenomena P1 and P2 (as with *LockModeType.OPTIMISTIC*) and must also force an update (increment) to the entity's version column. A forced version update may be performed immediately, or may be deferred until a flush or commit. If an entity is removed before a deferred version update was to have been applied, the forced version update is omitted.

The persistence implementation is not required to support calling *lock(entity, LockModeType.OPTIMISTIC_FORCE_INCREMENT)* on a non-versioned object. When it cannot support such a lock call, it must throw the *PersistenceException*. When supported, whether for versioned or non-versioned objects, *LockModeType.OPTIMISTIC_FORCE_INCREMENT* must always prevent the phenomena P1 and P2. For non-versioned objects, whether or not *LockModeType.OPTIMISTIC_FORCE_INCREMENT* has any additional behavior is vendor-specific. Applications that call *lock(entity, LockModeType.OPTIMISTIC_FORCE_INCREMENT)* on non-versioned objects will not be portable.

For versioned objects, it is permissible for an implementation to use *LockModeType.OPTIMISTIC_FORCE_INCREMENT* where *LockModeType.OPTIMISTIC* was requested, but not vice versa.

If a versioned object is otherwise updated or removed, then the implementation must ensure that the requirements of *LockModeType.OPTIMISTIC_FORCE_INCREMENT* are met, even if no explicit call to *EntityManager.lock* was made.

For portability, an application should not depend on vendor-specific hints or configuration to ensure repeatable read for objects that are not updated or removed via any mechanism other than the use of version attributes and the EntityManager *lock* method. However, it should be noted that if an implementation has acquired up-front pessimistic locks on some database rows, then it is free to ignore *lock(entity, LockModeType.OPTIMISTIC)* calls on the entity objects representing those rows.

3.4.4.2. PESSIMISTIC_READ, PESSIMISTIC_WRITE, PESSIMISTIC_FORCE_INCREMENT

The lock modes *PESSIMISTIC_READ*, *PESSIMISTIC_WRITE*, and *PESSIMISTIC_FORCE_INCREMENT* are used to immediately obtain long-term database locks. [40: Databases concurrency control mechanisms that provide comparable semantics, e.g., multiversion concurrency control, can be used by the provider.]

The semantics of requesting locks of type *LockModeType.PESSIMISTIC_READ*, *LockModeType.PESSIMISTIC_WRITE*, and *LockModeType.PESSIMISTIC_FORCE_INCREMENT* are the following.

If transaction T1 calls *lock(entity, LockModeType.PESSIMISTIC_READ)* or *lock(entity, LockModeType.PESSIMISTIC_WRITE)* on an object, the entity manager must ensure that neither of the following phenomena can occur:

- P1 (Dirty read): Transaction T1 modifies a row. Another transaction T2 then reads that row and obtains the modified value, before T1 has committed or rolled back.

- P2 (Non-repeatable read): Transaction T1 reads a row. Another transaction T2 then modifies or deletes that row, before T1 has committed or rolled back.

Any such lock must be obtained immediately and retained until transaction T1 completes (commits or rolls back).

Avoidance of phenomena P1 and P2 is generally achieved by the entity manager acquiring a long-term lock on the underlying database row(s). Any implementation that supports pessimistic repeatable reads as described above is permissible.

A lock with *LockModeType.PESSIMISTIC_WRITE* can be obtained on an entity instance to force serialization among transactions attempting to update the entity data. A lock with *LockModeType.PESSIMISTIC_READ* can be used to query data using repeatable-read semantics without the need to reread the data at the end of the transaction to obtain a lock, and without blocking other transactions reading the data. A lock with *LockModeType.PESSIMISTIC_WRITE* can be used when querying data and there is a high likelihood of deadlock or update failure among concurrent updating transactions.

The persistence implementation must support calling *lock(entity, LockModeType.PESSIMISTIC_READ)* and *lock(entity, LockModeType.PESSIMISTIC_WRITE)* on a non-versioned entity as well as on a versioned entity.

It is permissible for an implementation to use *LockModeType.PESSIMISTIC_WRITE* where *LockModeType.PESSIMISTIC_READ* was requested, but not vice versa.

When the lock cannot be obtained, and the database locking failure results in transaction-level rollback, the provider must throw the *PessimisticLockException* and ensure that the JTA transaction or EntityTransaction has been marked for rollback.

When the lock cannot be obtained, and the database locking failure results in only statement-level rollback, the provider must throw the *LockTimeoutException* (and must not mark the transaction for rollback).

When an application locks an entity with *LockModeType.PESSIMISTIC_READ* and later updates that entity, the lock must be converted to an exclusive lock when the entity is flushed to the database. [41: The persistence provider is not required to flush the entity to the database immediately.] If the lock conversion fails, and the database locking failure results in transaction-level rollback, the provider must throw the *PessimisticLockException* and ensure that the JTA transaction or EntityTransaction has been marked for rollback. When the lock conversion fails, and the database locking failure results in only statement-level rollback, the provider must throw the *LockTimeoutException* (and must not mark the transaction for rollback).

When *lock(entity, LockModeType.PESSIMISTIC_READ)*, *lock(entity, LockModeType.PESSIMISTIC_WRITE)*, or *lock(entity, LockModeType.PESSIMISTIC_FORCE_INCREMENT)* is invoked on a versioned entity that is already in the persistence context, the provider must also perform optimistic version checks when

obtaining the lock. An *OptimisticLockException* must be thrown if the version checks fail. Depending on the implementation strategy used by the provider, it is possible that this exception may not be thrown until flush is called or commit time, whichever occurs first.

If transaction T1 calls *lock(entity, LockModeType.PESSIMISTIC_FORCE_INCREMENT)* on a versioned object, the entity manager must avoid the phenomenon P1 and P2 (as with *LockModeType.PESSIMISTIC_READ* and *LockModeType.PESSIMISTIC_WRITE*) and must also force an update (increment) to the entity's version column.

The persistence implementation is not required to support calling *lock(entity, LockModeType.PESSIMISTIC_FORCE_INCREMENT)* on a non-versioned object. When it cannot support such a lock call, it must throw the *PersistenceException*. When supported, whether for versioned or non-versioned objects, *LockModeType.PESSIMISTIC_FORCE_INCREMENT* must always prevent the phenomena P1 and P2. For non-versioned objects, whether or not *LockModeType.PESSIMISTIC_FORCE_INCREMENT* has any additional behavior is vendor-specific. Applications that call *lock(entity, LockModeType.PESSIMISTIC_FORCE_INCREMENT)* on non-versioned objects will not be portable.

For versioned objects, it is permissible for an implementation to use *LockModeType.PESSIMISTIC_FORCE_INCREMENT* where *LockModeType.PESSIMISTIC_READ* or *LockModeType.PESSIMISTIC_WRITE* was requested, but not vice versa.

If a versioned object locked with *LockModeType.PESSIMISTIC_READ* or *LockModeType.PESSIMISTIC_WRITE* is updated, then the implementation must ensure that the requirements of *LockModeType.PESSIMISTIC_FORCE_INCREMENT* are met.

3.4.4.3. Lock Mode Properties and Uses

The following property is defined by this specification for use in pessimistic locking, as described in Section 3.4.3:

```
javax.persistence.lock.scope
```

This property may be used with the methods of the *EntityManager* interface that allow lock modes to be specified, the *Query* and *TypedQuery setLockMode* methods, and the *NamedQuery* annotation. When specified, this property must be observed. The provider is permitted to lock more (but not fewer) rows than requested.

The following hint is defined by this specification for use in pessimistic locking.

```
javax.persistence.lock.timeout // time in milliseconds
```

This hint may be used with the methods of the *EntityManager* interface that allow lock modes to be specified, the *Query.setLockMode* method and the *NamedQuery* annotation. It may also be passed as a

property to the *Persistence.createEntityManagerFactory* method and used in the *properties* element of the *persistence.xml* file. See Section 3.1.1, Section 3.10.9, Section 8.2.1.9, Section 9.7, and Section 10.4.1. When used in the *createEntityManagerFactory* method, the *persistence.xml* file, and the *NamedQuery* annotation, the timeout hint serves as a default value which can be selectively overridden by use in the methods of the *EntityManager*, *Query*, and *TypedQuery* interfaces as specified above. When this hint is not specified, database timeout values are assumed to apply.

A timeout value of *0* is used to specify “no wait” locking.

Portable applications should not rely on this hint. Depending on the database in use and the locking mechanisms used by the persistence provider, the hint may or may not be observed.

Vendors are permitted to support the use of additional, vendor-specific locking hints. Vendor-specific hints must not use the *javax.persistence* namespace. Vendor-specific hints must be ignored if they are not understood.

If the same property or hint is specified more than once, the following order of overriding applies, in order of decreasing precedence:

- argument to method of *EntityManager*, *Query*, or *TypedQuery* interface
- specification to *NamedQuery* (annotation or XML)
- argument to *createEntityManagerFactory* method
- specification in *persistence.xml*

3.4.5. OptimisticLockException

Provider implementations may defer writing to the database until the end of the transaction, when consistent with the lock mode and flush mode settings in effect. In this case, an optimistic lock check may not occur until commit time, and the *OptimisticLockException* may be thrown in the “before completion” phase of the commit. If the *OptimisticLockException* must be caught or handled by the application, the *flush* method should be used by the application to force the database writes to occur. This will allow the application to catch and handle optimistic lock exceptions.

The *OptimisticLockException* provides an API to return the object that caused the exception to be thrown. The object reference is not guaranteed to be present every time the exception is thrown but should be provided whenever the persistence provider can supply it. Applications cannot rely upon this object being available.

In some cases an *OptimisticLockException* will be thrown and wrapped by another exception, such as a *RemoteException*, when VM boundaries are crossed. Entities that may be referenced in wrapped exceptions should implement *Serializable* so that marshalling will not fail.

An *OptimisticLockException* always causes the transaction to be marked for rollback.

Refreshing objects or reloading objects in a new transaction context and then retrying the transaction is a potential response to an *OptimisticLockException*.

3.5. Entity Listeners and Callback Methods

A method may be designated as a lifecycle callback method to receive notification of entity lifecycle events. A lifecycle callback method can be defined on an entity class, a mapped superclass, or an entity listener class associated with an entity or mapped superclass. An entity listener class is a class whose methods are invoked in response to lifecycle events on an entity. Any number of entity listener classes can be defined for an entity class or mapped superclass.

Default entity listeners—entity listener classes whose callback methods apply to all entities in the persistence unit—can be specified by means of the XML descriptor.

Lifecycle callback methods and entity listener classes are defined by means of metadata annotations or the XML descriptor. When annotations are used, one or more entity listener classes are denoted using the *EntityListeners* annotation on the entity class or mapped superclass. If multiple entity listeners are defined, the order in which they are invoked is determined by the order in which they are specified in the *EntityListeners* annotation. The XML descriptor may be used as an alternative to specify the invocation order of entity listeners or to override the order specified in metadata annotations.

Any subset or combination of annotations may be specified on an entity class, mapped superclass, or listener class. A single class must not have more than one lifecycle callback method for the same lifecycle event. The same method may be used for multiple callback events.

Multiple entity classes and mapped superclasses in an inheritance hierarchy may define listener classes and/or lifecycle callback methods directly on the class. Section 3.5.5 describes the rules that apply to method invocation order in this case.

3.5.1. Entity Listeners

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

Entity listener classes in Java EE environments support dependency injection through the Contexts and Dependency Injection API (CDI) [7] when CDI is enabled [42: CDI is enabled by default in Java EE. See the Java EE specification [6].]. An entity listener class that makes use of CDI injection may also define lifecycle callback methods annotated with the *PostConstruct* and *PreDestroy* annotations. These methods will be invoked after injection has taken place and before the entity listener instance is destroyed respectively.

The persistence provider is responsible for using the CDI SPI to create instances of the entity listener class; to perform injection upon such instances; to invoke their *PostConstruct* and *PreDestroy* methods, if any; and to dispose of the entity listener instances.

The persistence provider is only required to support CDI injection into entity listeners in Java EE container environments [43: The persistence provider may support CDI injection into entity listeners in other environments in which the BeanManager is available.]. If the CDI is not enabled, the persistence provider must not invoke entity listeners that depend upon CDI injection.

An entity listener is a noncontextual object. In supporting injection into entity listeners, the persistence provider must behave as if it carries out the following steps involving the use of the CDI SPI. (See [7]).

- Obtain a *BeanManager* instance. (See Section 9.1)
- Create an *AnnotatedType* instance for the entity listener class.
- Create an *InjectionTarget* instance for the annotated type.
- Create a *CreationalContext*.
- Instantiate the listener by calling the *InjectionTarget produce* method.
- Inject the listener instance by calling the *InjectionTarget inject* method on the instance.
- Invoke the *PostConstruct* callback, if any, by calling the *InjectionTarget postConstruct* method on the instance.

When the listener instance is to be destroyed, the persistence provider must behave as if it carries out the following steps.

- Call the *InjectionTarget preDestroy* method on the instance.
- Call the *InjectionTarget dispose* method on the instance
- Call the *CreationalContext release* method.

Persistence providers may optimize the steps above, e.g. by avoiding calls to the actual CDI SPI and relying on container-specific interfaces instead, as long as the outcome is the same.

Entity listeners that do not make use of CDI injection are stateless. The lifecycle of such entity listeners is unspecified.

When invoked from within a Java EE environment, the callback listeners for an entity share the enterprise naming context of the invoking component, and the entity callback methods are invoked in the transaction and security contexts of the calling component at the time at which the callback method is invoked. [44: For example, if a transaction commit occurs as a result of the normal termination of a session bean business method with transaction attribute *RequiresNew*, the *PostPersist* and *PostRemove* callbacks are executed in the naming context, the transaction context, and the security context of that component.]

3.5.2. Lifecycle Callback Methods

Entity lifecycle callback methods can be defined on an entity listener class and/or directly on an entity class or mapped superclass.

Lifecycle callback methods are annotated with annotations designating the callback events for which they are invoked or are mapped to the callback event using the XML descriptor.

The annotations (and XML elements) used for callback methods on the entity class or mapped superclass and for callback methods on the entity listener class are the same. The signatures of

individual methods, however, differ.

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

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

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

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

The *Object* argument is the entity instance for which the callback method is invoked. It may be declared as the actual entity type.

The callback methods can have public, private, protected, or package level access, but must not be *static* or *final*.

The following annotations designate lifecycle event callback methods of the corresponding types.

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

The following rules apply to lifecycle callback methods:

- Lifecycle callback methods may throw unchecked/runtime exceptions. A runtime exception thrown by a callback method that executes within a transaction causes that transaction to be marked for rollback if the persistence context is joined to the transaction.
- Lifecycle callbacks can invoke JNDI, JDBC, JMS, and enterprise beans.

In general, the lifecycle method of a portable application should not invoke *EntityManager* or query operations, access other entity instances, or modify relationships within the same persistence context [45: Note that this caution applies also to the actions of objects that might be injected into an entity listener] [46: The semantics of such operations may be standardized in a future release of this specification.]. A lifecycle callback method may modify the non-relationship state of the entity on which it is invoked.

3.5.3. 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. For entities to which the merge operation has been applied and causes the creation of newly managed instances, the *PrePersist* callback methods will be invoked for the managed instance after the entity state has been copied to it. These *PrePersist* and *PreRemove* 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, merge, and remove operations.

The *PostPersist* and *PostRemove* callback methods are invoked for an entity after the entity has been made persistent or removed. 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. These database operations may occur directly after the persist, merge, or remove operations have been invoked or they may occur directly after a flush operation has occurred (which may be at the end of the transaction). Generated primary key values are available in the *PostPersist* method.

The *PreUpdate* and *PostUpdate* callbacks occur before and after the database update operations to entity data respectively. These database operations may occur at the time the entity state is updated or they may occur at the time state is flushed to the database (which may be 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.

It is implementation-dependent as to whether callback methods are invoked before or after the cascading of the lifecycle events to related entities. Applications should not depend on this ordering.

3.5.4. Example

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

    @Id
    public Long getAccountId() { ... }

    // ...

    public Integer getBalance() { ... }

    // ...

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

    // ...

    public void deposit(Integer amount) { ... }

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

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

    @PostLoad
    protected void adjustPreferredStatus() {
        preferred = (getBalance() >= AccountManager.getPreferredStatusLevel());
    }
}

public class AlertMonitor {
    @PostPersist
    public void newAccountAlert(Account acct) {
        Alerts.sendMarketingInfo(acct.getAccountId(), acct.getBalance());
    }
}
```

3.5.5. Multiple Lifecycle Callback Methods for an Entity Lifecycle Event

If multiple callback methods are defined for an entity lifecycle event, the ordering of the invocation of these methods is as follows.

Default listeners, if any, are invoked first, in the order specified in the XML descriptor. Default listeners apply to all entities in the persistence unit, unless explicitly excluded by means of the *ExcludeDefaultListeners* annotation or *exclude-default-listeners* XML element.

The lifecycle callback methods defined on the entity listener classes for an entity class or mapped superclass are invoked in the same order as the specification of the entity listener classes in the *EntityListeners* annotation.

If multiple classes in an inheritance hierarchy—entity classes and/or mapped superclasses—define entity listeners, the listeners defined for a superclass are invoked before the listeners defined for its subclasses in this order. The *ExcludeSuperclassListeners* annotation or *exclude-superclass-listeners* XML element may be applied to an entity class or mapped superclass to exclude the invocation of the listeners defined by the entity listener classes for the superclasses of the entity or mapped superclass. The excluded listeners are excluded from the class to which the *ExcludeSuperclassListeners* annotation or element has been specified and its subclasses [47: Excluded listeners may be reintroduced on an entity class by listing them explicitly in the *EntityListeners* annotation or XML *entity-listeners* element.]. The *ExcludeSuperclassListeners* annotation (or *exclude-superclass-listeners* XML element) does not cause default entity listeners to be excluded from invocation.

If a lifecycle callback method for the same lifecycle event is also specified on the entity class and/or one or more of its entity or mapped superclasses, the callback methods on the entity class and/or superclasses are invoked after the other lifecycle callback methods, most general superclass first. A class is permitted to override an inherited callback method of the same callback type, and in this case, the overridden method is not invoked [48: If a method overrides an inherited callback method but specifies a different lifecycle event or is not a lifecycle callback method, the overridden method will not be invoked.].

Callback methods are invoked by the persistence provider runtime in the order specified. If the callback method execution terminates normally, the persistence provider runtime then invokes the next callback method, if any.

The XML descriptor may be used to override the lifecycle callback method invocation order specified in annotations.

3.5.6. Example

There are several entity classes and listeners for animals:

```
@Entity
public class Animal {

    // ...

    @PostPersist
    protected void postPersistAnimal() {
        // ...
    }
}

@Entity
@EntityListeners(PetListener.class)
public class Pet extends Animal {
    // ...
}

@Entity
@EntityListeners({CatListener.class, CatListener2.class})
public class Cat extends Pet {
    // ...
}

public class PetListener {
    @PostPersist
    protected void postPersistPetListenerMethod(Object pet) {
        // ...
    }
}

public class CatListener {
    @PostPersist
    protected void postPersistCatListenerMethod(Object cat) {
        // ...
    }
}

public class CatListener2 {
    @PostPersist
    protected void postPersistCatListener2Method(Object cat) {
        // ...
    }
}
```

If a *PostPersist* event occurs on an instance of *Cat*, the following methods are called in order:

1. postPersistPetListenerMethod
2. postPersistCatListenerMethod
3. postPersistCatListener2Method
4. postPersistAnimal

Assume that *SiameseCat* is defined as a subclass of *Cat*:

```
@EntityListeners(SiameseCatListener.class)
@Entity
public class SiameseCat extends Cat {
    // ...

    @PostPersist
    protected void postPersistSiameseCat() {
        // ...
    }
}

public class SiameseCatListener {
    @PostPersist
    protected void postPersistSiameseCatListenerMethod(Object cat) {
        // ...
    }
}
```

If a *PostPersist* event occurs on an instance of *SiameseCat*, the following methods are called in order:

1. postPersistPetListenerMethod
2. postPersistCatListenerMethod
3. postPersistCatListener2Method
4. postPersistSiameseCatListenerMethod
5. postPersistAnimal
6. postPersistSiameseCat

Assume the definition of *SiameseCat* were instead:

```
@EntityListeners(SiameseCatListener.class)
@Entity
public class SiameseCat extends Cat {
    // ...

    @PostPersist
    protected void postPersistAnimal() {
        // ...
    }
}
```

In this case, the following methods would be called in order, where *postPersistAnimal* is the *PostPersist* method defined in the *SiameseCat* class:

1. postPersistPetListenerMethod
2. postPersistCatListenerMethod
3. postPersistCatListener2Method
4. postPersistSiameseCatListenerMethod
5. postPersistAnimal

3.5.7. Exceptions

Lifecycle callback methods may throw runtime exceptions. A runtime exception thrown by a callback method that executes within a transaction causes that transaction to be marked for rollback if the persistence context is joined to the transaction. No further lifecycle callback methods will be invoked after a runtime exception is thrown.

3.5.8. Specification of Callback Listener Classes and Lifecycle Methods in the XML Descriptor

The XML descriptor can be used as an alternative to metadata annotations to specify entity listener classes and their binding to entities or to override the invocation order of lifecycle callback methods as specified in annotations.

3.5.8.1. Specification of Callback Listeners

The *entity-listener* XML descriptor element is used to specify the lifecycle listener methods of an entity listener class. The lifecycle listener methods are specified by using the *pre-persist*, *post-persist*, *pre-remove*, *post-remove*, *pre-update*, *post-update*, and/or *post-load* elements.

An entity listener class can define multiple callback methods. However, at most one method of an entity listener class can be designated as a pre-persist method, post-persist method, pre-remove method, post-remove method, pre-update method, post-update method, and/or post-load method,

regardless of whether the XML descriptor is used to define entity listeners or whether some combination of annotations and XML descriptor elements is used.

3.5.8.2. Specification of the Binding of Entity Listener Classes to Entities

The *entity-listeners* subelement of the *persistence-unit-defaults* element is used to specify the default entity listeners for the persistence unit.

The *entity-listeners* subelement of the *entity* or *mapped-superclass* element is used to specify the entity listener classes for the respective entity or mapped superclass and its subclasses.

The binding of entity listeners to entity classes is additive. The entity listener classes bound to the superclasses of an entity or mapped superclass are applied to it as well.

The *exclude-superclass-listeners* element specifies that the listener methods for superclasses are not to be invoked for an entity class (or mapped superclass) and its subclasses.

The *exclude-default-listeners* element specifies that default entity listeners are not to be invoked for an entity class (or mapped superclass) and its subclasses.

Explicitly listing an excluded default or superclass listener for a given entity class or mapped superclass causes it to be applied to that entity or mapped superclass and its subclasses.

In the case of multiple callback methods for a single lifecycle event, the invocation order rules described in Section 3.5.5 apply.

3.6. Bean Validation

This specification defines support for use of Bean Validation [5] within Java Persistence applications.

Managed classes (entities, mapped superclasses, and embeddable classes) may be configured to include Bean Validation constraints.

Automatic validation using these constraints is achieved by specifying that Java Persistence delegate validation to the Bean Validation implementation upon the pre-persist, pre-update, and pre-remove entity lifecycle events described in Section 3.5.3.

Validation can also be achieved by the application calling the *validate* method of a *Validator* instance upon an instance of a managed class, as described in the Bean Validation specification [5].

3.6.1. Automatic Validation Upon Lifecycle Events

This specification supports the use of bean validation for the automatic validation of entities upon the pre-persist, pre-update, and pre-remove lifecycle validation events. These lifecycle validation events occur immediately after the point at which all the *PrePersist*, *PreUpdate*, and *PreRemove* lifecycle callback method invocations respectively have been completed, or immediately after the point at which such lifecycle callback methods would have been completed (in the event that such callback

methods are not present).

In the case where an entity is persisted and subsequently modified in a single transaction or when an entity is modified and subsequently removed in a single transaction, it is implementation dependent as to whether the pre-update validation event occurs. Portable applications should not rely on this behavior.

3.6.1.1. Enabling Automatic Validation

The *validation-mode* element of the *persistence.xml* file determines whether the automatic lifecycle event validation is in effect. The values of the *validation-mode* element are *AUTO*, *CALLBACK*, *NONE*. The default validation mode is *AUTO*.

If the application creates the entity manager factory using the *Persistence.createEntityManagerFactory* method, the validation mode can be specified using the *javax.persistence.validation.mode* map key, which will override the value specified (or defaulted) in the *persistence.xml* file. The map values for this key are *"auto"*, *"callback"*, *"none"*.

If the auto validation mode is specified by the *validation-mode* element or the *javax.persistence.validation.mode* property, or if neither the *validation-mode* element nor the *javax.persistence.validation.mode* property is specified, and a Bean Validation provider is present in the environment, the persistence provider must perform the automatic validation of entities as described in Section 3.6.1.2. If no Bean Validation provider is present in the environment, no lifecycle event validation takes place.

If the callback validation mode is specified by the *validation-mode* element or the *javax.persistence.validation.mode* property, the persistence provider must perform the lifecycle event validation as described in Section 3.6.1.2. It is an error if there is no Bean Validation provider present in the environment, and the provider must throw the *PersistenceException* if the *javax.persistence.validation.mode* property value *"callback"* has been passed to the *Persistence.createEntityManagerFactory* method.

If the none validation mode is specified by the *validation-mode* element or the *javax.persistence.validation.mode* property, the persistence provider must not perform lifecycle event validation.

3.6.1.2. Requirements for Automatic Validation upon Lifecycle Events

For each event type, a list of groups is targeted for validation. By default, the default Bean Validation group (the group *Default*) will be validated upon the pre-persist and pre-update lifecycle validation events, and no group will be validated upon the pre-remove event.

This default validation behavior can be overridden by specifying the target groups using the following validation properties in the *persistence.xml* file or by passing these properties in the configuration of the entity manager factory through the *createEntityManagerFactory* method:

- *javax.persistence.validation.group.pre-persist*
- *javax.persistence.validation.group.pre-update*
- *javax.persistence.validation.group.pre-remove*

The value of a validation property must be a list of the targeted groups. A targeted group must be specified by its fully qualified class name. Names must be separated by a comma.

When one of the above events occurs for an entity, the persistence provider must validate that entity by obtaining a *Validator* instance from the validator factory in use (see Section 3.6.2) and invoking its *validate* method with the targeted groups. If the list of targeted groups is empty, no validation is performed. If the set of *ConstraintViolation* objects returned by the *validate* method is not empty, the persistence provider must throw the *javax.validation.ConstraintViolationException* containing a reference to the returned set of *ConstraintViolation* objects, and must mark the transaction for rollback if the persistence context is joined to the transaction.

The validator instance that is used for automatic validation upon lifecycle events must use a *TraversableResolver* that has the following behavior:

- Attributes that have not been loaded must not be loaded.
- Validation cascade (*@Valid*) must not occur for entity associations (single- or multi-valued).

These requirements guarantee that no unloaded attribute or association will be loaded by side effect and that no entity will be validated more than once during a given flush cycle.

Embeddable attributes must be validated only if the *Valid* annotation has been specified on them.

It is the responsibility of the persistence provider to pass an instance implementing the *javax.validation.TraversableResolver* interface to the Bean Validation provider by calling *ValidatorFactory.usingContext().traversableResolver(tr).getValidator()* where *tr* is the resolver having the behavior described above.

3.6.2. Providing the ValidatorFactory

In Java EE environments, a *ValidatorFactory* instance is made available by the Java EE container. The container is responsible for passing this validator factory to the persistence provider via the map that is passed as an argument to the *createContainerEntityManagerFactory* call. The map key used by the container must be the standard property name *javax.persistence.validation.factory*.

In Java SE environments, the application can pass the *ValidatorFactory* instance via the map that is passed as an argument to the *Persistence.createEntityManagerFactory* call. The map key used must be the standard property name *javax.persistence.validation.factory*. If no *ValidatorFactory* instance is provided by the application, and if a Bean Validation provider is present in the classpath, the persistence provider must instantiate the *ValidatorFactory* using the default bootstrapping approach defined by the Bean Validation specification [5], namely *Validation.buildDefaultValidatorFactory()*.

3.7. Entity Graphs

An entity graph is a template that captures the path and boundaries for an operation or query. It is defined in the form of metadata or an object created by the dynamic *EntityGraph* API.

Entity graphs are used in the specification of "fetch plans" for query or *find* operations.

The *EntityGraph*, *AttributeNode*, and *Subgraph* interfaces are used to dynamically construct entity graphs. The annotations to statically define entity graphs, namely *NamedEntityGraph*, *NamedAttributeNode*, and *NamedSubgraph*, are described in Section 10.3. The *named-entity-graph* XML element and its subelements may be used to override these annotations or to define additional named entity graphs.

The semantics of entity graphs with regard to find and query operations are described in Section 3.7.4.

3.7.1. EntityGraph Interface

```
package jakarta.persistence;

import jakarta.persistence.metamodel.Attribute;
import java.util.List;

/**
 * This type represents the root of an entity graph that will be used
 * as a template to define the attribute nodes and boundaries of a
 * graph of entities and entity relationships. The root must be an
 * entity type.
 * <p>
 * The methods to add subgraphs implicitly create the
 * corresponding attribute nodes as well; such attribute nodes
 * should not be redundantly specified.
 *
 * @param <T> The type of the root entity.
 *
 * @see AttributeNode
 * @see Subgraph
 * @see NamedEntityGraph
 *
 * @since 2.1
 */
public interface EntityGraph<T> {

    /**
     * Return the name of a named EntityGraph (an entity graph
     * defined by means of the <code>NamedEntityGraph</code>
     * annotation, XML descriptor element, or added by means of the
```

```
     * <code>addNamedEntityGraph</code> method.  Returns null if the
     * EntityGraph is not a named EntityGraph.
     */
    public String getName();

    /**
     * Add one or more attribute nodes to the entity graph.
     *
     * @param attributeName  name of the attribute
     * @throws IllegalArgumentException if the attribute is not an
     *         attribute of this entity.
     * @throws IllegalStateException if the EntityGraph has been
     *         statically defined
     */
    public void addAttributeNodes(String ... attributeName);

    /**
     * Add one or more attribute nodes to the entity graph.
     *
     * @param attribute  attribute
     * @throws IllegalStateException if the EntityGraph has been
     *         statically defined
     */
    public void addAttributeNodes(Attribute<T, ?>... attribute);

    /**
     * Add a node to the graph that corresponds to a managed
     * type. This allows for construction of multi-node entity graphs
     * that include related managed types.
     *
     * @param attribute  attribute
     * @return subgraph for the attribute
     * @throws IllegalArgumentException if the attribute's target type
     *         is not a managed type
     * @throws IllegalStateException if the EntityGraph has been
     *         statically defined
     */
    public <X> Subgraph<X> addSubgraph(Attribute<T, X> attribute);

    /**
     * Add a node to the graph that corresponds to a managed
     * type with inheritance.  This allows for multiple subclass
     * subgraphs to be defined for this node of the entity
     * graph. Subclass subgraphs will automatically include the
     * specified attributes of superclass subgraphs.
     *
     * @param attribute  attribute
     * @param type  entity subclass
```

```
     * @return subgraph for the attribute
     * @throws IllegalArgumentException if the attribute's target
     *         type is not a managed type
     * @throws IllegalStateException if the EntityGraph has been
     *         statically defined
     */
    public <X> Subgraph<? extends X> addSubgraph(Attribute<T, X> attribute, Class<?
extends X> type);

    /**
     * Add a node to the graph that corresponds to a managed
     * type. This allows for construction of multi-node entity graphs
     * that include related managed types.
     *
     * @param attributeName  name of the attribute
     * @return subgraph for the attribute
     * @throws IllegalArgumentException if the attribute is not an
     *         attribute of this entity.
     * @throws IllegalArgumentException if the attribute's target type
     *         is not a managed type
     * @throws IllegalStateException if the EntityGraph has been
     *         statically defined
     */
    public <X> Subgraph<X> addSubgraph(String attributeName);

    /**
     * Add a node to the graph that corresponds to a managed
     * type with inheritance.  This allows for multiple subclass
     * subgraphs to be defined for this node of the entity graph.
     * Subclass subgraphs will automatically include the specified
     * attributes of superclass subgraphs.
     *
     * @param attributeName  name of the attribute
     * @param type  entity subclass
     * @return subgraph for the attribute
     * @throws IllegalArgumentException if the attribute is not an
     *         attribute of this managed type.
     * @throws IllegalArgumentException if the attribute's target type
     *         is not a managed type
     * @throws IllegalStateException if this EntityGraph has been
     *         statically defined
     */
    public <X> Subgraph<X> addSubgraph(String attributeName, Class<X> type);

    /**
     * Add a node to the graph that corresponds to a map key
     * that is a managed type. This allows for construction of
     * multi-node entity graphs that include related managed types.
```

```
     *
     * @param attribute  attribute
     * @return subgraph for the key attribute
     * @throws IllegalArgumentException if the attribute's target type
     *         is not an entity
     * @throws IllegalStateException if this EntityGraph has been
     *         statically defined
     */
    public <X> Subgraph<X> addKeySubgraph(Attribute<T, X> attribute);

    /**
     * Add a node to the graph that corresponds to a map key
     * that is a managed type with inheritance. This allows for
     * construction of multi-node entity graphs that include related
     * managed types.  Subclass subgraphs will include the specified
     * attributes of superclass subgraphs.
     *
     * @param attribute  attribute
     * @param type  entity subclass
     * @return subgraph for the key attribute
     * @throws IllegalArgumentException if the attribute's target type
     *         is not an entity
     * @throws IllegalStateException if this EntityGraph has been
     *         statically defined
     */
    public <X> Subgraph<? extends X> addKeySubgraph(Attribute<T, X> attribute, Class<?
extends X> type);

    /**
     * Add a node to the graph that corresponds to a map key
     * that is a managed type. This allows for construction of
     * multi-node entity graphs that include related managed types.
     *
     * @param attributeName  name of the attribute
     * @return subgraph for the key attribute
     * @throws IllegalArgumentException if the attribute is not an
     *         attribute of this entity.
     * @throws IllegalArgumentException if the attribute's target type
     *         is not an entity
     * @throws IllegalStateException if this EntityGraph has been
     *          statically defined
     */
    public <X> Subgraph<X> addKeySubgraph(String attributeName);

    /**
     * Add a node to the graph that corresponds to a map key
     * that is a managed type with inheritance. This allows for
     * construction of multi-node entity graphs that include related
```

```
     * managed types. Subclass subgraphs will automatically include
     * the specified attributes of superclass subgraphs
     *
     * @param attributeName  name of the attribute
     * @param type  entity subclass
     * @return subgraph for the key attribute
     * @throws IllegalArgumentException if the attribute is not an
     *         attribute of this entity.
     * @throws IllegalArgumentException if the attribute's target type
     *         is not a managed type
     * @throws IllegalStateException if this EntityGraph has been
     *         statically defined
     */
    public <X> Subgraph<X> addKeySubgraph(String attributeName, Class<X> type);


    /**
     * Add additional attributes to this entity graph that
     * correspond to attributes of subclasses of this EntityGraph's
     * entity type.  Subclass subgraphs will automatically include the
     * specified attributes of superclass subgraphs.
     *
     * @param type  entity subclass
     * @return subgraph for the subclass
     * @throws IllegalArgumentException if the type is not an entity type
     * @throws IllegalStateException if the EntityGraph has been
     *         statically defined
     */
    public <T> Subgraph<? extends T> addSubclassSubgraph(Class<? extends T> type);


    /**
     * Return the attribute nodes of this entity that are included in
     * the entity graph.
     * @return attribute nodes for the annotated entity type or empty
     *         list if none have been defined
     */
    public List<AttributeNode<?>> getAttributeNodes();

}
```

3.7.2. AttributeNode Interface

```
package jakarta.persistence;

import java.util.Map;

/**
 * Represents an attribute node of an entity graph.
 *
 * @param <T> The type of the attribute.
 *
 * @see EntityGraph
 * @see Subgraph
 * @see NamedAttributeNode
 *
 * @since 2.1
 */
public interface AttributeNode<T> {

    /**
     * Return the name of the attribute corresponding to the
     * attribute node.
     * @return name of the attribute
     */
    public String getAttributeName();

    /**
     * Return the Map&#060;Class, Subgraph&#062; of subgraphs associated
     * with this attribute node.
     * @return Map of subgraphs associated with this attribute node
     * or empty Map if none have been defined
     */
    public Map<Class, Subgraph> getSubgraphs();

    /**
     * Return the Map&#060;Class, Subgraph&#062; of subgraphs associated
     * with this attribute node's map key.
     * @return Map of subgraphs associated with this attribute
     * node's map key or empty Map if none have been defined
     */
    public Map<Class, Subgraph> getKeySubgraphs();
}
```

3.7.3. Subgraph Interface

```
package jakarta.persistence;
```

```
import jakarta.persistence.metamodel.Attribute;
import java.util.List;

/**
 * This type represents a subgraph for an attribute node that
 * corresponds to a Managed Type. Using this class, an entity subgraph
 * can be embedded within an EntityGraph.
 *
 * @param <T> The type of the attribute.
 *
 * @see EntityGraph
 * @see AttributeNode
 * @see NamedSubgraph
 *
 * @since 2.1
 */
public interface Subgraph<T> {

    /**
     * Add one or more attribute nodes to the entity graph.
     *
     * @param attributeName  name of the attribute
     * @throws IllegalArgumentException if the attribute is not an
     *         attribute of this managed type.
     * @throws IllegalStateException if the EntityGraph has been
     *         statically defined
     */
    public void addAttributeNodes(String ... attributeName);

    /**
     * Add one or more attribute nodes to the entity graph.
     * @param attribute  attribute
     *
     * @throws IllegalStateException if this EntityGraph has been
     *         statically defined
     */
    public void addAttributeNodes(Attribute<T, ?>... attribute);

    /**
     * Add a node to the graph that corresponds to a managed
     * type. This allows for construction of multi-node entity graphs
     * that include related managed types.
     *
     * @param attribute  attribute
     * @return subgraph for the attribute
     * @throws IllegalArgumentException if the attribute's target
     *         type is not a managed type
     * @throws IllegalStateException if the EntityGraph has been
```

```
     *         statically defined
     */
    public <X> Subgraph<X> addSubgraph(Attribute<T, X> attribute);

    /**
     * Add a node to the graph that corresponds to a managed
     * type with inheritance.  This allows for multiple subclass
     * subgraphs to be defined for this node of the entity
     * graph. Subclass subgraphs will automatically include the specified
     * attributes of superclass subgraphs
     *
     * @param attribute  attribute
     * @param type  entity subclass
     * @return subgraph for the attribute
     * @throws IllegalArgumentException if the attribute's target
     *         type is not a managed type
     * @throws IllegalStateException if this EntityGraph has been
     *         statically defined
     */
    public <X> Subgraph<? extends X> addSubgraph(Attribute<T, X> attribute, Class<?
extends X> type);

    /**
     * Add a node to the graph that corresponds to a managed
     * type. This allows for construction of multi-node entity graphs
     * that include related managed types.
     *
     * @param attributeName  name of the attribute
     * @return subgraph for the attribute
     * @throws IllegalArgumentException if the attribute is not an
     *         attribute of this managed type.
     * @throws IllegalArgumentException if the attribute's target
     *         type is not a managed type
     * @throws IllegalStateException if this EntityGraph has been
     *         statically defined
     */
    public <X> Subgraph<X> addSubgraph(String attributeName);

    /**
     * Add a node to the graph that corresponds to a managed
     * type with inheritance.  This allows for multiple subclass
     * subgraphs to be defined for this node of the entity
     * graph. Subclass subgraphs will automatically include the
     * specified attributes of superclass subgraphs
     *
     * @param attributeName  name of the attribute
     * @param type  entity subclass
     * @return subgraph for the attribute
```

```
     * @throws IllegalArgumentException if the attribute is not
     *         an attribute of this managed type.
     * @throws IllegalArgumentException if the attribute's target
     *         type is not a managed type
     * @throws IllegalStateException if this EntityGraph has been
     *         statically defined
     */
    public <X> Subgraph<X> addSubgraph(String attributeName, Class<X> type);

    /**
     * Add a node to the graph that corresponds to a map key
     * that is a managed type. This allows for construction of
     * multinode entity graphs that include related managed types.
     *
     * @param attribute  attribute
     * @return subgraph for the key attribute
     * @throws IllegalArgumentException if the attribute's target
     *         type is not a managed type entity
     * @throws IllegalStateException if this EntityGraph has been
     *         statically defined
     */
    public <X> Subgraph<X> addKeySubgraph(Attribute<T, X> attribute);

    /**
     * Add a node to the graph that corresponds to a map key
     * that is a managed type with inheritance. This allows for
     * construction of multi-node entity graphs that include related
     * managed types.  Subclass subgraphs will automatically include
     * the specified attributes of superclass subgraphs
     *
     * @param attribute  attribute
     * @param type  entity subclass
     * @return subgraph for the attribute
     * @throws IllegalArgumentException if the attribute's target
     *         type is not a managed type entity
     * @throws IllegalStateException if this EntityGraph has been
     *         statically defined
     */
    public <X> Subgraph<? extends X> addKeySubgraph(Attribute<T, X> attribute, Class<?
extends X> type);

    /**
     * Add a node to the graph that corresponds to a map key
     * that is a managed type. This allows for construction of
     * multi-node entity graphs that include related managed types.
     *
     * @param attributeName  name of the attribute
     * @return subgraph for the key attribute
```

```
     * @throws IllegalArgumentException if the attribute is not an
     *         attribute of this entity.
     * @throws IllegalArgumentException if the attribute's target
     *         type is not a managed type
     * @throws IllegalStateException if this EntityGraph has been
     *          statically defined
     */
    public <X> Subgraph<X> addKeySubgraph(String attributeName);

    /**
     * Add a node to the graph that corresponds to a map key
     * that is a managed type with inheritance. This allows for
     * construction of multi-node entity graphs that include related
     * managed types.  Subclass subgraphs will include the specified
     * attributes of superclass subgraphs
     *
     * @param attributeName  name of the attribute
     * @param type  entity subclass
     * @return subgraph for the attribute
     * @throws IllegalArgumentException if the attribute is not an
     *         attribute of this entity.
     * @throws IllegalArgumentException if the attribute's target
     *         type is not a managed type
     * @throws IllegalStateException if this EntityGraph has been
     *         statically defined
     */
    public <X> Subgraph<X> addKeySubgraph(String attributeName, Class<X> type);

    /**
     * Return the attribute nodes corresponding to the attributes of
     * this managed type that are included in the subgraph.
     * @return list of attribute nodes included in the subgraph or
     * empty List if none have been defined
     */
    public List<AttributeNode<?>> getAttributeNodes();

    /**
     * Return the type for which this subgraph was defined.
     * @return managed type referenced by the subgraph
     */
    public Class<T> getClassType();
}
```

3.7.4. Use of Entity Graphs in find and query operations

An entity graph can be used with the *find* method or as a query hint to override or augment *FetchType* semantics.

The standard properties *javax.persistence.fetchgraph* and *javax.persistence.loadgraph* are used to specify such graphs to queries and *find* operations.

The default fetch graph for an entity or embeddable is defined to consist of the transitive closure of all of its attributes that are specified as *FetchType.EAGER* (or defaulted as such).

The persistence provider is permitted to fetch additional entity state beyond that specified by a fetch graph or load graph. It is required, however, that the persistence provider fetch all state specified by the fetch or load graph.

3.7.4.1. Fetch Graph Semantics

When the *javax.persistence.fetchgraph* property is used to specify an entity graph, attributes that are specified by attribute nodes of the entity graph are treated as *FetchType.EAGER* and attributes that are not specified are treated as *FetchType.LAZY*.

The following rules apply, depending on attribute type. The rules of this section are applied recursively.

A primary key or version attribute never needs to be specified in an attribute node of a fetch graph. (This applies to composite primary keys as well, including embedded id primary keys.) When an entity is fetched, its primary key and version attributes are always fetched. It is not incorrect, however, to specify primary key attributes or version attributes.

Attributes other than primary key and version attributes are assumed not to be fetched unless the attribute is specified. The following rules apply to the specification of attributes.

- If the attribute is an embedded attribute, and the attribute is specified in an attribute node, but a subgraph is not specified for the attribute, the default fetch graph for the embeddable is fetched. If a subgraph is specified for the attribute, the attributes of the embeddable are fetched according to their specification in the corresponding subgraph.
- If the attribute is an element collection of basic type, and the attribute is specified in an attribute node, the element collection together with its basic elements is fetched.
- If the attribute is an element collection of embeddables, and the attribute is specified in an attribute node, but a subgraph is not specified for the attribute, the element collection together with the default fetch graph of its embeddable elements is fetched. If a subgraph is specified for the attribute, the attributes of the embeddable elements are fetched according to the corresponding subgraph specification.
- If the attribute is a one-to-one or many-to-one relationship, and the attribute is specified in an attribute node, but a subgraph is not specified for the attribute, the default fetch graph of the target entity is fetched. If a subgraph is specified for the attribute, the attributes of the target entity are fetched according to the corresponding subgraph specification.
- If the attribute is a one-to-many or many-to-many relationship, and the attribute is specified in an attribute node, but a subgraph is not specified, the collection is fetched and the default fetch graphs of the referenced entities are fetched. If a subgraph is specified for the attribute, the entities in the collection are fetched according to the corresponding subgraph specification.

- If the key of a map which has been specified in an attribute node is a basic type, it is fetched. If the key of a map which has been specified in an attribute node is an embedded type, the default fetch graph is fetched for the embeddable. Otherwise, if the key of the map is an entity, and a map key subgraph is not specified for the attribute node, the map key is fetched according to its default fetch graph. If a key subgraph is specified for the map key attribute, the map key attribute is fetched according to the map key subgraph specification.

Examples:

```
@NamedEntityGraph
@Entity
public class Phonenumber {
    @Id
    protected String number;

    protected PhoneTypeEnum type;

    // ...
}
```

In the above example, only the *number* attribute would be eagerly fetched.

```
@NamedEntityGraph(
    attributeNodes={@NamedAttributeNode("projects")}
)
@Entity
public class Employee {
    @Id
    @GeneratedValue
    protected long id;

    @Basic
    protected String name;

    @Basic
    protected String employeeNumber;

    @OneToMany()
    protected List<Dependents> dependents;

    @OneToMany()
    protected List<Project> projects;

    @OneToMany()
    protected List<PhoneNumber> phoneNumbers;
```

```
    // ...
}

@Entity
@Inheritance
public class Project {
    @Id
    @GeneratedValue
    protected long id;

    String name;

    @OneToOne(fetch=FetchType.EAGER)
    protected Requirements doc;

    // ...
}

@Entity
public class LargeProject extends Project {
    @OneToOne(fetch=FetchType.LAZY)
    protected Employee approver;

    // ...
}

@Entity
public class Requirements {
    @Id
    protected long id;

    @Lob
    protected String description;

    @OneToOne(fetch=FetchType.LAZY)
    protected Approval approval

    // ...
}
```

In the above example, the *Employee* entity's primary key will be fetched as well as the related *Project* instances, whose default fetch graph (*id*, *name*, and *doc* attributes) will be fetched. The related *Requirements* object will be fetched according to its default fetch graph.

If the *approver* attribute of *LargeProject* were *FetchType.EAGER*, and if any of the projects were instances of *LargeProject*, their *approver* attributes would also be fetched. Since the type of the *approver* attribute is *Employee*, the approver's default fetch graph (*id*, *name*, and *employeeNumber*

attributes) would also be fetched.

3.7.4.2. Load Graph Semantics

When the *javax.persistence.loadgraph* property is used to specify an entity graph, attributes that are specified by attribute nodes of the entity graph are treated as *FetchType.EAGER* and attributes that are not specified are treated according to their specified or default FetchType.

The following rules apply. The rules of this section are applied recursively.

- A primary key or version attribute never needs to be specified in an attribute node of a load graph. (This applies to composite primary keys as well, including embedded id primary keys.) When an entity is fetched, its primary key and version attributes are always fetched. It is not incorrect, however, to specify primary key attributes or version attributes.
- If the attribute is an embedded attribute, and the attribute is specified in an attribute node, but a subgraph is not specified for the attribute, the default fetch graph for the embeddable is fetched. If a subgraph is specified for the attribute, attributes that are specified by the subgraph are also fetched.
- If the attribute is an element collection of basic type, and the attribute is specified in an attribute node, the element collection together with its basic elements is fetched.
- If the attribute is an element collection of embeddables, and the attribute is specified in an attribute node, the element collection together with the default fetch graph of its embeddable elements is fetched. If a subgraph is specified for the attribute, attributes that are specified by the subgraph are also fetched.
- If the attribute is a one-to-one or many-to-one relationship, and the attribute is specified in an attribute node, the default fetch graph of the target entity is fetched. If a subgraph is specified for the attribute, attributes that are specified by the subgraph are also fetched.
- If the attribute is a one-to-many or many-to-many relationship, and the attribute is specified in an attribute node, the collection is fetched and the default fetch graphs of the referenced entities are fetched. If a subgraph is specified for the attribute, attributes that are specified by the subgraph are also fetched.
- If the key of a map which has been specified in an attribute node is a basic type, it is fetched. If the key of a map which has been specified in an attribute node is an embedded type, the default fetch graph is fetched for the embeddable. Otherwise, if the key of the map is an entity, the map key is fetched according to its default fetch graph. If a key subgraph is specified for the map key attribute, additional attributes are fetched as specified in the key subgraph.

Examples:

```
@NamedEntityGraph
@Entity
public class Phonenumber {
    @Id
    protected String number;

    protected PhoneTypeEnum type;

    // ...
}
```

In the above example, the *number* and *type* attributes are fetched.

```
@NamedEntityGraph(
    attributeNodes={@NamedAttributeNode("projects")}
)
@Entity
public class Employee {
    @Id
    @GeneratedValue
    protected long id;

    @Basic
    protected String name;

    @Basic
    protected String employeeNumber;

    @OneToMany()
    protected List<Dependents> dependents;

    @OneToMany()
    protected List<Project> projects;

    @OneToMany()
    protected List<PhoneNumber> phoneNumbers;

    // ...
}

@Entity
@Inheritance
public class Project {
    @Id
    @GeneratedValue
    protected long id;
```

```
    String name;

    @OneToOne(fetch=FetchType.EAGER)
    protected Requirements doc;

    // ...
}

@Entity
public class LargeProject extends Project {
    @OneToOne(fetch=FetchType.LAZY)
    protected Employee approver;

    // ...
}

@Entity
public class Requirements {
    @Id
    protected long id;

    @Lob
    protected String description;

    @OneToOne(fetch=FetchType.LAZY)
    protected Approval approval

    // ...
}
```

In the above example, the default fetch graph (*id*, *name*, *employeeNumber* attributes) of *Employee* is fetched. The default fetch graphs of the related *Project* instances (*id*, *name*, and *doc* attributes) and their *Requirements* instances (*id* and *description* attributes) are also fetched.

3.8. Type Conversion of Basic Attributes

The attribute conversion facility allows the developer to specify methods to convert between the entity attribute representation and the database representation for attributes of basic types. Converters can be used to convert basic attributes defined by entity classes, mapped superclasses, or embeddable classes. [49: We plan to provide a facility for more complex attribute conversions in a future release of this specification.]

An attribute converter must implement the *javax.persistence.AttributeConverter* interface. A converter implementation class must be annotated with the *Converter* annotation or defined in the XML descriptor as a converter. If the value of the *autoApply* element of the *Converter* annotation is *true*, the

converter will be applied to all attributes of the target type, including to basic attribute values that are contained within other, more complex attribute types. See Section 10.6.

```
package jakarta.persistence;

/**
 * A class that implements this interface can be used to convert
 * entity attribute state into database column representation
 * and back again.
 * Note that the X and Y types may be the same Java type.
 *
 * @param <X>  the type of the entity attribute
 * @param <Y>  the type of the database column
 */
public interface AttributeConverter<X,Y> {

    /**
     * Converts the value stored in the entity attribute into the
     * data representation to be stored in the database.
     *
     * @param attribute  the entity attribute value to be converted
     * @return  the converted data to be stored in the database
     *          column
     */
    public Y convertToDatabaseColumn (X attribute);

    /**
     * Converts the data stored in the database column into the
     * value to be stored in the entity attribute.
     * Note that it is the responsibility of the converter writer to
     * specify the correct <code>dbData</code> type for the corresponding
     * column for use by the JDBC driver: i.e., persistence providers are
     * not expected to do such type conversion.
     *
     * @param dbData  the data from the database column to be
     *                converted
     * @return  the converted value to be stored in the entity
     *          attribute
     */
    public X convertToEntityAttribute (Y dbData);
}
```

Attribute converter classes in Java EE environments support dependency injection through the Contexts and Dependency Injection API (CDI) [7] when CDI is enabled [50: CDI is enabled by default in Java EE. See the Java EE specification [6].]. An attribute converter class that makes use of CDI injection may also define lifecycle callback methods annotated with the *PostConstruct* and *PreDestroy*

annotations. These methods will be invoked after injection has taken place and before the attribute converter instance is destroyed respectively.

The persistence provider is responsible for using the CDI SPI to create instances of the attribute converter class; to perform injection upon such instances; to invoke their *PostConstruct* and *PreDestroy* methods, if any; and to dispose of the attribute converter instances.

The persistence provider is only required to support CDI injection into attribute converters in Java EE container environments [51: The persistence provider may support CDI injection into attribute converters in other environments in which the BeanManager is available.]. If CDI is not enabled, the persistence provider must not invoke attribute converters that depend upon CDI injection.

An attribute converter is a noncontextual object. In supporting injection into attribute converters, the persistence provider must behave as if it carries out the following steps involving the use of the CDI SPI. (See [7]).

- Obtain a *BeanManager* instance. (See Section 9.1.)
- Create an *AnnotatedType* instance for the attribute converter class.
- Create an *InjectionTarget* instance for the annotated type.
- Create a *CreationalContext*.
- Instantiate the listener by calling the *InjectionTarget produce* method.
- Inject the listener instance by calling the *InjectionTarget inject* method on the instance.
- Invoke the *PostConstruct* callback, if any, by calling the *InjectionTarget postConstruct* method on the instance.

When the listener instance is to be destroyed, the persistence provider must behave as if it carries out the following steps.

- Call the *InjectionTarget preDestroy* method on the instance.
- Call the *InjectionTarget dispose* method on the instance.
- Call the *CreationalContext release* method.

Persistence providers may optimize the steps above, e.g. by avoiding calls to the actual CDI SPI and relying on container-specific interfaces instead, as long as the outcome is the same.

Attribute converters that do not make use of CDI injection are stateless. The lifecycle of such attribute converters is unspecified.

The conversion of all basic types is supported except for the following: Id attributes (including the attributes of embedded ids and derived identities), version attributes, relationship attributes, and attributes explicitly annotated as *Enumerated* or *Temporal* or designated as such in the XML descriptor. Auto-apply converters will not be applied to such attributes, and applications that apply converters to such attributes through use of the *Convert* annotation will not be portable.

Type conversion may be specified at the level of individual attributes by means of the *Convert* annotation. The *Convert* annotation may also be used to override or disable an auto-apply conversion. See Section 11.1.10.

The *Convert* annotation may be applied directly to an attribute of an entity, mapped superclass, or embeddable class to specify conversion of the attribute or to override the use of a converter that has been specified as *autoApply=true*. When persistent properties are used, the *Convert* annotation is applied to the getter method.

The *Convert* annotation may be applied to an entity that extends a mapped superclass to specify or override the conversion mapping for an inherited basic or embedded attribute.

The persistence provider runtime is responsible for invoking the specified conversion methods for the target attribute type when loading the entity attribute from the database and before storing the entity attribute state to the database. The persistence provider must apply any conversion methods to instances of attribute values in path expressions used within Java Persistence query language queries or criteria queries (such as in comparisons, bulk updates, etc.) before sending them to the database for the query execution. When such converted attributes are used in comparison operations with literals or parameters, the value of the literal or parameter to which they are compared must also be converted. If the result of a Java Persistence query language query or criteria query includes one or more entity attributes for which conversion mappings have been specified, the persistence provider must apply the specified conversions to the corresponding values in the query result before returning them to the application. The use of functions, including aggregates, on converted attributes is undefined. If an exception is thrown from a conversion method, the persistence provider must wrap the exception in a PersistenceException and, if the persistence context is joined to a transaction, mark the transaction for rollback.

3.9. Caching

This specification supports the use of a second-level cache by the persistence provider. The second-level cache, if used, underlies the persistence context, and is largely transparent to the application.

A second-level cache is typically used to enhance performance. Use of a cache, however, may have consequences in terms of the up-to-dateness of the data seen by the application, resulting in “stale reads”. A stale read is defined as the reading of entities or entity state that is older than the point at which the persistence context was started.

This specification defines the following portable configuration options that can be used by the application developer to control caching behavior. Persistence providers may support additional provider-specific options, but must observe all specification-defined options.

3.9.1. The shared-cache-mode Element

Whether the entities and entity-related state of a persistence unit will be cached is determined by the value of the *shared-cache-mode* element of the *persistence.xml* file.

The *shared-cache-mode* element has five possible values: *ALL*, *NONE*, *ENABLE_SELECTIVE*, *DISABLE_SELECTIVE*, *UNSPECIFIED*.

A value of *ALL* causes all entities and entity-related state and data to be cached.

A value of *NONE* causes caching to be disabled for the persistence unit. Persistence providers must not cache if *NONE* is specified.

The values *ENABLE_SELECTIVE* and *DISABLE_SELECTIVE* are used in conjunction with the *Cacheable* annotation (or XML element). The *Cacheable* annotation specifies whether an entity should be cached if such selective caching is enabled by the *persistence.xml shared-cache-mode* element. The *Cacheable* element is specified on the entity class. It applies to the given entity and its subclasses unless subsequently overridden by a subclass.

- *Cacheable(false)* means that the entity and its state must not be cached by the provider.
- A value of *ENABLE_SELECTIVE* enables the cache and causes entities for which *Cacheable(true)* (or its XML equivalent) is specified to be cached. Entities for which *Cacheable(true)* is not specified or for which *Cacheable(false)* is specified must not be cached.
- A value of *DISABLE_SELECTIVE* enables the cache and causes all entities to be cached except those for which *Cacheable(false)* is specified. Entities for which *Cacheable(false)* is specified must not be cached.

If either the *shared-cache-mode* element is not specified in the *persistence.xml* file or the value of the *shared-cache-mode* element is *UNSPECIFIED*, and the *javax.persistence.sharedCache.mode* property is not specified, the behavior is not defined, and provider-specific defaults may apply. If the *shared-cache-mode* element and the *javax.persistence.sharedCache.mode* property are not specified, the semantics of the *Cacheable* annotation (and XML equivalent) are undefined.

The persistence provider is not required to support use of a second-level cache. If the persistence provider does not support use of a second-level cache or a second-level cache is not installed, this element will be ignored and no caching will occur.

Further control over the second-level cache is described in Section 7.10.

3.9.2. Cache Retrieve Mode and Cache Store Mode Properties

Cache retrieve mode and cache store mode properties may be specified at the level of the persistence context by means of the EntityManager *setProperty* method. These properties may be specified for the *EntityManager find* and *refresh* methods and the *Query*, *TypedQuery*, *and StoredProcedureQuery setHint* methods. Cache retrieve mode and/or cache store mode properties specified for the *find*, *refresh*, and *Query*, *TypedQuery*, and *StoredProcedureQuery setHint* methods override those specified for the persistence context for the specified *find* and *refresh* invocations, and for the execution of the specified queries respectively.

If caching is disabled by the *NONE* value of the *shared-cache-mode* element, cache retrieve mode and cache store mode properties must be ignored. Otherwise, if the *ENABLE_SELECTIVE* value is specified,

but *Cacheable(true)* is not specified for a particular entity, they are ignored for that entity; if the *DISABLE_SELECTIVE* value is specified, they are ignored for any entities for which *Cacheable(false)* is specified.

Cache retrieve mode and cache store mode properties must be observed when caching is enabled, regardless of whether caching is enabled due to the specification of the *shared-cache-mode* element or enabled due to provider-specific options. Applications that make use of cache retrieve mode or cache store mode properties but which do not specify the *shared-cache-mode* element will not be portable.

The cache retrieve mode and cache store mode properties are *javax.persistence.cache.retrieveMode* and *javax.persistence.cache.storeMode* respectively. These properties have the semantics defined below.

The *retrieveMode* property specifies the behavior when data is retrieved by the *find* methods and by the execution of queries. The *retrieveMode* property is ignored for the *refresh* method, which always causes data to be retrieved from the database, not the cache.

```
package jakarta.persistence;

/**
 * Used as the value of the
 * <code>jakarta.persistence.cache.retrieveMode</code> property to
 * specify the behavior when data is retrieved by the
 * <code>find</code> methods and by queries.
 *
 * @since 2.0
 */
public enum CacheRetrieveMode {

    /**
     * Read entity data from the cache: this is
     * the default behavior.
     */
    USE,

    /**
     * Bypass the cache: get data directly from
     * the database.
     */
    BYPASS
}
```

The *storeMode* property specifies the behavior when data is read from the database and when data is committed into the database.

```
package jakarta.persistence;

/**
 * Used as the value of the
 * <code>jakarta.persistence.cache.storeMode</code> property to specify
 * the behavior when data is read from the database and when data is
 * committed into the database.
 *
 * @since 2.0
 */
public enum CacheStoreMode {

    /**
     * Insert entity data into cache when read from database
     * and insert/update entity data when committed into database:
     * this is the default behavior. Does not force refresh
     * of already cached items when reading from database.
     */
    USE,

    /**
     * Don't insert into cache.
     */
    BYPASS,

    /**
     * Insert/update entity data into cache when read
     * from database and when committed into database.
     * Forces refresh of cache for items read from database.
     */
    REFRESH
}
```

3.10. Query APIs

The *Query* and *TypedQuery* APIs are used for the execution of both static queries and dynamic queries. These APIs also support parameter binding and pagination control. The *StoredProcedureQuery* API is used for the execution of queries that invoke stored procedures defined in the database.

3.10.1. Query Interface

```
package jakarta.persistence;

import java.util.Calendar;
import java.util.Date;
```

```
import java.util.List;
import java.util.Set;
import java.util.Map;
import java.util.stream.Stream;

/**
 * Interface used to control query execution.
 *
 * @see TypedQuery
 * @see StoredProcedureQuery
 * @see Parameter
 *
 * @since 1.0
 */
public interface Query {

    /**
     * Execute a SELECT query and return the query results
     * as an untyped List.
     * @return a list of the results
     * @throws IllegalStateException if called for a Java
     *         Persistence query language UPDATE or DELETE statement
     * @throws QueryTimeoutException if the query execution exceeds
     *         the query timeout value set and only the statement is
     *         rolled back
     * @throws TransactionRequiredException if a lock mode other than
     *         <code>NONE</code> has been set and there is no transaction
     *         or the persistence context has not been joined to the transaction
     * @throws PessimisticLockException if pessimistic locking
     *         fails and the transaction is rolled back
     * @throws LockTimeoutException if pessimistic locking
     *         fails and only the statement is rolled back
     * @throws PersistenceException if the query execution exceeds
     *         the query timeout value set and the transaction
     *         is rolled back
     */
    List getResultList();

    /**
     * Execute a SELECT query and return the query results
     * as an untyped <code>java.util.stream.Stream</code>.
     * By default this method delegates to <code>getResultList().stream()</code>,
     * however persistence provider may choose to override this method
     * to provide additional capabilities.
     *
     * @return a stream of the results
     * @throws IllegalStateException if called for a Java
     *         Persistence query language UPDATE or DELETE statement
```

```
     * @throws QueryTimeoutException if the query execution exceeds
     *         the query timeout value set and only the statement is
     *         rolled back
     * @throws TransactionRequiredException if a lock mode other than
     *         <code>NONE</code> has been set and there is no transaction
     *         or the persistence context has not been joined to the transaction
     * @throws PessimisticLockException if pessimistic locking
     *         fails and the transaction is rolled back
     * @throws LockTimeoutException if pessimistic locking
     *         fails and only the statement is rolled back
     * @throws PersistenceException if the query execution exceeds
     *         the query timeout value set and the transaction
     *         is rolled back
     * @see Stream
     * @see #getResultList()
     * @since 2.2
     */
    default Stream getResultStream() {
        return getResultList().stream();
    }

    /**
     * Execute a SELECT query that returns a single untyped result.
     * @return the result
     * @throws NoResultException if there is no result
     * @throws NonUniqueResultException if more than one result
     * @throws IllegalStateException if called for a Java
     *         Persistence query language UPDATE or DELETE statement
     * @throws QueryTimeoutException if the query execution exceeds
     *         the query timeout value set and only the statement is
     *         rolled back
     * @throws TransactionRequiredException if a lock mode other than
     *         <code>NONE</code> has been set and there is no transaction
     *         or the persistence context has not been joined to the transaction
     * @throws PessimisticLockException if pessimistic locking
     *         fails and the transaction is rolled back
     * @throws LockTimeoutException if pessimistic locking
     *         fails and only the statement is rolled back
     * @throws PersistenceException if the query execution exceeds
     *         the query timeout value set and the transaction
     *         is rolled back
     */
    Object getSingleResult();

    /**
     * Execute an update or delete statement.
     * @return the number of entities updated or deleted
     * @throws IllegalStateException if called for a Java
```

```
     *         Persistence query language SELECT statement or for
     *         a criteria query
     * @throws TransactionRequiredException if there is
     *         no transaction or the persistence context has not
     *         been joined to the transaction
     * @throws QueryTimeoutException if the statement execution
     *         exceeds the query timeout value set and only
     *         the statement is rolled back
     * @throws PersistenceException if the query execution exceeds
     *         the query timeout value set and the transaction
     *         is rolled back
     */
    int executeUpdate();

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

    /**
     * The maximum number of results the query object was set to
     * retrieve. Returns <code>Integer.MAX_VALUE</code> if <code>setMaxResults</code> was
not
     * applied to the query object.
     * @return maximum number of results
     * @since 2.0
     */
    int getMaxResults();

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

    /**
     * The position of the first result the query object was set to
     * retrieve. Returns 0 if <code>setFirstResult</code> was not applied to the
     * query object.
     * @return position of the first result
     * @since 2.0
     */
```

```
    int getFirstResult();

    /**
     * Set a query property or hint. The hints elements may be used
     * to specify query properties and hints. Properties defined by
     * this specification must be observed by the provider.
     * Vendor-specific hints that are not recognized by a provider
     * must be silently ignored. Portable applications should not
     * rely on the standard timeout hint. Depending on the database
     * in use and the locking mechanisms used by the provider,
     * this hint may or may not be observed.
     * @param hintName  name of the property or hint
     * @param value  value for the property or hint
     * @return the same query instance
     * @throws IllegalArgumentException if the second argument is not
     *         valid for the implementation
     */
    Query setHint(String hintName, Object value);

    /**
     * Get the properties and hints and associated values that are
     * in effect for the query instance.
     * @return query properties and hints
     * @since 2.0
     */
    Map<String, Object> getHints();

    /**
     * Bind the value of a <code>Parameter</code> object.
     * @param param  parameter object
     * @param value  parameter value
     * @return the same query instance
     * @throws IllegalArgumentException if the parameter
     *         does not correspond to a parameter of the
     *         query
     * @since 2.0
     */
    <T> Query setParameter(Parameter<T> param, T value);

    /**
     * Bind an instance of <code>java.util.Calendar</code> to a <code>Parameter</code>
object.
     * @param param parameter object
     * @param value  parameter value
     * @param temporalType  temporal type
     * @return the same query instance
     * @throws IllegalArgumentException if the parameter does not
     *         correspond to a parameter of the query
```

```
     * @since 2.0
     */
    Query setParameter(Parameter<Calendar> param, Calendar value,
                       TemporalType temporalType);

    /**
     * Bind an instance of <code>java.util.Date</code> to a <code>Parameter</code>
object.
     * @param param parameter object
     * @param value  parameter value
     * @param temporalType  temporal type
     * @return the same query instance
     * @throws IllegalArgumentException if the parameter does not
     *         correspond to a parameter of the query
     * @since 2.0
     */
    Query setParameter(Parameter<Date> param, Date value,
                       TemporalType temporalType);

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

    /**
     * Bind an instance of <code>java.util.Calendar</code> to a named parameter.
     * @param name  parameter name
     * @param value  parameter value
     * @param temporalType  temporal type
     * @return the same query instance
     * @throws IllegalArgumentException if the parameter name does
     *         not correspond to a parameter of the query or if
     *         the value argument is of incorrect type
     */
    Query setParameter(String name, Calendar value,
                       TemporalType temporalType);

    /**
     * Bind an instance of <code>java.util.Date</code> to a named parameter.
     * @param name  parameter name
     * @param value  parameter value
     * @param temporalType  temporal type
```

```
     * @return the same query instance
     * @throws IllegalArgumentException if the parameter name does
     *         not correspond to a parameter of the query or if
     *         the value argument is of incorrect type
     */
    Query setParameter(String name, Date value,
                       TemporalType temporalType);

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

    /**
     * Bind an instance of <code>java.util.Calendar</code> to a positional
     * parameter.
     * @param position  position
     * @param value  parameter value
     * @param temporalType  temporal type
     * @return the same query instance
     * @throws IllegalArgumentException if position does not
     *         correspond to a positional parameter of the query or
     *         if the value argument is of incorrect type
     */
    Query setParameter(int position, Calendar value,
                       TemporalType temporalType);

    /**
     * Bind an instance of <code>java.util.Date</code> to a positional parameter.
     * @param position  position
     * @param value  parameter value
     * @param temporalType  temporal type
     * @return the same query instance
     * @throws IllegalArgumentException if position does not
     *         correspond to a positional parameter of the query or
     *         if the value argument is of incorrect type
     */
    Query setParameter(int position, Date value,
                       TemporalType temporalType);

    /**
     * Get the parameter objects corresponding to the declared
```

```
     * parameters of the query.
     * Returns empty set if the query has no parameters.
     * This method is not required to be supported for native
     * queries.
     * @return set of the parameter objects
     * @throws IllegalStateException if invoked on a native
     *         query when the implementation does not support
     *         this use
     * @since 2.0
     */
    Set<Parameter<?>> getParameters();

    /**
     * Get the parameter object corresponding to the declared
     * parameter of the given name.
     * This method is not required to be supported for native
     * queries.
     * @param name  parameter name
     * @return parameter object
     * @throws IllegalArgumentException if the parameter of the
     *         specified name does not exist
     * @throws IllegalStateException if invoked on a native
     *         query when the implementation does not support
     *         this use
     * @since 2.0
     */
    Parameter<?> getParameter(String name);

    /**
     * Get the parameter object corresponding to the declared
     * parameter of the given name and type.
     * This method is required to be supported for criteria queries
     * only.
     * @param name  parameter name
     * @param type  type
     * @return parameter object
     * @throws IllegalArgumentException if the parameter of the
     *         specified name does not exist or is not assignable
     *         to the type
     * @throws IllegalStateException if invoked on a native
     *         query or Jakarta Persistence query language query when
     *         the implementation does not support this use
     * @since 2.0
     */
    <T> Parameter<T> getParameter(String name, Class<T> type);

    /**
     * Get the parameter object corresponding to the declared
```

```
     * positional parameter with the given position.
     * This method is not required to be supported for native
     * queries.
     * @param position  position
     * @return parameter object
     * @throws IllegalArgumentException if the parameter with the
     *         specified position does not exist
     * @throws IllegalStateException if invoked on a native
     *         query when the implementation does not support
     *         this use
     * @since 2.0
     */
    Parameter<?> getParameter(int position);

    /**
     * Get the parameter object corresponding to the declared
     * positional parameter with the given position and type.
     * This method is not required to be supported by the provider.
     * @param position  position
     * @param type  type
     * @return parameter object
     * @throws IllegalArgumentException if the parameter with the
     *         specified position does not exist or is not assignable
     *         to the type
     * @throws IllegalStateException if invoked on a native
     *         query or Jakarta Persistence query language query when
     *         the implementation does not support this use
     * @since 2.0
     */
    <T> Parameter<T> getParameter(int position, Class<T> type);

    /**
     * Return a boolean indicating whether a value has been bound
     * to the parameter.
     * @param param parameter object
     * @return boolean indicating whether parameter has been bound
     * @since 2.0
     */
    boolean isBound(Parameter<?> param);

    /**
     * Return the input value bound to the parameter.
     * (Note that OUT parameters are unbound.)
     * @param param parameter object
     * @return parameter value
     * @throws IllegalArgumentException if the parameter is not
     *         a parameter of the query
     * @throws IllegalStateException if the parameter has not been
```

```
     *         been bound
     * @since 2.0
     */
    <T> T getParameterValue(Parameter<T> param);

    /**
     * Return the input value bound to the named parameter.
     * (Note that OUT parameters are unbound.)
     * @param name  parameter name
     * @return parameter value
     * @throws IllegalStateException if the parameter has not been
     *         been bound
     * @throws IllegalArgumentException if the parameter of the
     *         specified name does not exist
     * @since 2.0
     */
    Object getParameterValue(String name);

    /**
     * Return the input value bound to the positional parameter.
     * (Note that OUT parameters are unbound.)
     * @param position  position
     * @return parameter value
     * @throws IllegalStateException if the parameter has not been
     *         been bound
     * @throws IllegalArgumentException if the parameter with the
     *         specified position does not exist
     * @since 2.0
     */
    Object getParameterValue(int position);

    /**
     * Set the flush mode type to be used for the query execution.
     * The flush mode type applies to the query regardless of the
     * flush mode type in use for the entity manager.
     * @param flushMode  flush mode
     * @return the same query instance
     */
    Query setFlushMode(FlushModeType flushMode);

    /**
     * Get the flush mode in effect for the query execution.
     * If a flush mode has not been set for the query object,
     * returns the flush mode in effect for the entity manager.
     * @return flush mode
     * @since 2.0
     */
    FlushModeType getFlushMode();
```

```
    /**
     * Set the lock mode type to be used for the query execution.
     * @param lockMode  lock mode
     * @return the same query instance
     * @throws IllegalStateException if the query is found not to be
     *         a Jakarta Persistence query language SELECT query
     *         or a CriteriaQuery query
     * @since 2.0
     */
    Query setLockMode(LockModeType lockMode);

    /**
     * Get the current lock mode for the query.  Returns null if a lock
     * mode has not been set on the query object.
     * @return lock mode
     * @throws IllegalStateException if the query is found not to be
     *         a Jakarta Persistence query language SELECT query or
     *         a Criteria API query
     * @since 2.0
     */
    LockModeType getLockMode();

    /**
     * Return an object of the specified type to allow access to
     * the provider-specific API.  If the provider's query
     * implementation does not support the specified class, the
     * <code>PersistenceException</code> is thrown.
     * @param cls  the class of the object to be returned.  This is
     *             normally either the underlying query
     *             implementation class or an interface that it
     *             implements.
     * @return an instance of the specified class
     * @throws PersistenceException if the provider does not support
     *         the call
     * @since 2.0
     */
    <T> T unwrap(Class<T> cls);
}
```

3.10.2. TypedQuery Interface

```
package jakarta.persistence;

import java.util.List;
import java.util.Date;
```

```
import java.util.Calendar;
import java.util.stream.Stream;

/**
 * Interface used to control the execution of typed queries.
 * @param <X> query result type
 *
 * @see Query
 * @see Parameter
 *
 * @since 2.0
 */
public interface TypedQuery<X> extends Query {

    /**
     * Execute a SELECT query and return the query results
     * as a typed List.
     * @return a list of the results
     * @throws IllegalStateException if called for a Java
     *         Persistence query language UPDATE or DELETE statement
     * @throws QueryTimeoutException if the query execution exceeds
     *         the query timeout value set and only the statement is
     *         rolled back
     * @throws TransactionRequiredException if a lock mode other than
     *         <code>NONE</code> has been set and there is no transaction
     *         or the persistence context has not been joined to the
     *         transaction
     * @throws PessimisticLockException if pessimistic locking
     *         fails and the transaction is rolled back
     * @throws LockTimeoutException if pessimistic locking
     *         fails and only the statement is rolled back
     * @throws PersistenceException if the query execution exceeds
     *         the query timeout value set and the transaction
     *         is rolled back
     */
    List<X> getResultList();

    /**
     * Execute a SELECT query and return the query results
     * as a typed <code>java.util.stream.Stream</code>.
     * By default this method delegates to <code>getResultList().stream()</code>,
     * however persistence provider may choose to override this method
     * to provide additional capabilities.
     *
     * @return a stream of the results
     * @throws IllegalStateException if called for a Java
     *         Persistence query language UPDATE or DELETE statement
     * @throws QueryTimeoutException if the query execution exceeds
```

```
     *          the query timeout value set and only the statement is
     *          rolled back
     * @throws TransactionRequiredException if a lock mode other than
     *          <code>NONE</code> has been set and there is no transaction
     *          or the persistence context has not been joined to the transaction
     * @throws PessimisticLockException if pessimistic locking
     *          fails and the transaction is rolled back
     * @throws LockTimeoutException if pessimistic locking
     *          fails and only the statement is rolled back
     * @throws PersistenceException if the query execution exceeds
     *          the query timeout value set and the transaction
     *          is rolled back
     * @see Stream
     * @see #getResultList()
     * @since 2.2
     */
    default Stream<X> getResultStream() {
        return getResultList().stream();
    }

    /**
     * Execute a SELECT query that returns a single result.
     * @return the result
     * @throws NoResultException if there is no result
     * @throws NonUniqueResultException if more than one result
     * @throws IllegalStateException if called for a Java
     *          Persistence query language UPDATE or DELETE statement
     * @throws QueryTimeoutException if the query execution exceeds
     *          the query timeout value set and only the statement is
     *          rolled back
     * @throws TransactionRequiredException if a lock mode other than
     *          <code>NONE</code> has been set and there is no transaction
     *          or the persistence context has not been joined to the
     *          transaction
     * @throws PessimisticLockException if pessimistic locking
     *          fails and the transaction is rolled back
     * @throws LockTimeoutException if pessimistic locking
     *          fails and only the statement is rolled back
     * @throws PersistenceException if the query execution exceeds
     *          the query timeout value set and the transaction
     *          is rolled back
     */
    X getSingleResult();

    /**
     * Set the maximum number of results to retrieve.
     * @param maxResult  maximum number of results to retrieve
     * @return the same query instance
```

```
     * @throws IllegalArgumentException if the argument is negative
     */
    TypedQuery<X> setMaxResults(int maxResult);

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

    /**
     * Set a query property or hint. The hints elements may be used
     * to specify query properties and hints. Properties defined by
     * this specification must be observed by the provider.
     * Vendor-specific hints that are not recognized by a provider
     * must be silently ignored. Portable applications should not
     * rely on the standard timeout hint. Depending on the database
     * in use and the locking mechanisms used by the provider,
     * this hint may or may not be observed.
     * @param hintName  name of property or hint
     * @param value  value for the property or hint
     * @return the same query instance
     * @throws IllegalArgumentException if the second argument is not
     *         valid for the implementation
     */
    TypedQuery<X> setHint(String hintName, Object value);

    /**
     * Bind the value of a <code>Parameter</code> object.
     * @param param  parameter object
     * @param value  parameter value
     * @return the same query instance
     * @throws IllegalArgumentException if the parameter
     *         does not correspond to a parameter of the
     *         query
     */
     <T> TypedQuery<X> setParameter(Parameter<T> param, T value);

    /**
     * Bind an instance of <code>java.util.Calendar</code> to a <code>Parameter</code>
object.
     * @param param  parameter object
     * @param value  parameter value
     * @param temporalType  temporal type
     * @return the same query instance
```

```
     * @throws IllegalArgumentException if the parameter does not
     *         correspond to a parameter of the query
     */
    TypedQuery<X> setParameter(Parameter<Calendar> param,
                               Calendar value,
                               TemporalType temporalType);

    /**
     * Bind an instance of <code>java.util.Date</code> to a <code>Parameter</code>
object.
     * @param param  parameter object
     * @param value  parameter value
     * @param temporalType  temporal type
     * @return the same query instance
     * @throws IllegalArgumentException if the parameter does not
     *         correspond to a parameter of the query
     */
    TypedQuery<X> setParameter(Parameter<Date> param, Date value,
                               TemporalType temporalType);

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

    /**
     * Bind an instance of <code>java.util.Calendar</code> to a named parameter.
     * @param name  parameter name
     * @param value  parameter value
     * @param temporalType  temporal type
     * @return the same query instance
     * @throws IllegalArgumentException if the parameter name does
     *         not correspond to a parameter of the query or if
     *         the value argument is of incorrect type
     */
    TypedQuery<X> setParameter(String name, Calendar value,
                               TemporalType temporalType);

    /**
     * Bind an instance of <code>java.util.Date</code> to a named parameter.
     * @param name   parameter name
     * @param value  parameter value
```

```
     * @param temporalType  temporal type
     * @return the same query instance
     * @throws IllegalArgumentException if the parameter name does
     *         not correspond to a parameter of the query or if
     *         the value argument is of incorrect type
     */
    TypedQuery<X> setParameter(String name, Date value,
                               TemporalType temporalType);

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

    /**
     * Bind an instance of <code>java.util.Calendar</code> to a positional
     * parameter.
     * @param position  position
     * @param value  parameter value
     * @param temporalType  temporal type
     * @return the same query instance
     * @throws IllegalArgumentException if position does not
     *         correspond to a positional parameter of the query
     *         or if the value argument is of incorrect type
     */
    TypedQuery<X> setParameter(int position, Calendar value,
                               TemporalType temporalType);

    /**
     * Bind an instance of <code>java.util.Date</code> to a positional parameter.
     * @param position  position
     * @param value  parameter value
     * @param temporalType  temporal type
     * @return the same query instance
     * @throws IllegalArgumentException if position does not
     *         correspond to a positional parameter of the query
     *         or if the value argument is of incorrect type
     */
    TypedQuery<X> setParameter(int position, Date value,
                               TemporalType temporalType);

     /**
```

```
     * Set the flush mode type to be used for the query execution.
     * The flush mode type applies to the query regardless of the
     * flush mode type in use for the entity manager.
     * @param flushMode  flush mode
     * @return the same query instance
     */
    TypedQuery<X> setFlushMode(FlushModeType flushMode);

    /**
     * Set the lock mode type to be used for the query execution.
     * @param lockMode  lock mode
     * @return the same query instance
     * @throws IllegalStateException if the query is found not to
     *         be a Jakarta Persistence query language SELECT query
     *         or a CriteriaQuery query
     */
    TypedQuery<X> setLockMode(LockModeType lockMode);

}
```

3.10.3. Tuple Interface

```
package jakarta.persistence;

import java.util.List;

/**
 * Interface for extracting the elements of a query result tuple.
 *
 * @see TupleElement
 *
 * @since 2.0
 */
public interface Tuple {

    /**
     * Get the value of the specified tuple element.
     * @param tupleElement  tuple element
     * @return value of tuple element
     * @throws IllegalArgumentException if tuple element
     *         does not correspond to an element in the
     *         query result tuple
     */
    <X> X get(TupleElement<X> tupleElement);

    /**
```

```
     * Get the value of the tuple element to which the
     * specified alias has been assigned.
     * @param alias  alias assigned to tuple element
     * @param type of the tuple element
     * @return value of the tuple element
     * @throws IllegalArgumentException if alias
     *         does not correspond to an element in the
     *         query result tuple or element cannot be
     *         assigned to the specified type
     */
    <X> X get(String alias, Class<X> type);

    /**
     * Get the value of the tuple element to which the
     * specified alias has been assigned.
     * @param alias  alias assigned to tuple element
     * @return value of the tuple element
     * @throws IllegalArgumentException if alias
     *         does not correspond to an element in the
     *         query result tuple
     */
    Object get(String alias);

    /**
     * Get the value of the element at the specified
     * position in the result tuple. The first position is 0.
     * @param i  position in result tuple
     * @param type  type of the tuple element
     * @return value of the tuple element
     * @throws IllegalArgumentException if i exceeds
     *         length of result tuple  or element cannot be
     *         assigned to the specified type
     */
    <X> X get(int i, Class<X> type);

    /**
     * Get the value of the element at the specified
     * position in the result tuple. The first position is 0.
     * @param i  position in result tuple
     * @return value of the tuple element
     * @throws IllegalArgumentException if i exceeds
     *         length of result tuple
     */
    Object get(int i);

    /**
     * Return the values of the result tuple elements as an array.
     * @return tuple element values
```

```
     */
    Object[] toArray();

    /**
     * Return the tuple elements.
     * @return tuple elements
     */
    List<TupleElement<?>> getElements();
}
```

3.10.4. TupleElement Interface

```
package jakarta.persistence;

/**
 * The <code>TupleElement</code> interface defines an element that is returned in
 * a query result tuple.
 * @param <X> the type of the element
 *
 * @see Tuple
 *
 * @since 2.0
 */
public interface TupleElement<X> {

    /**
     * Return the Java type of the tuple element.
     * @return the Java type of the tuple element
     */
    Class<? extends X> getJavaType();

    /**
     * Return the alias assigned to the tuple element or null,
     * if no alias has been assigned.
     * @return alias
     */
    String getAlias();
}
```

3.10.5. Parameter Interface

```
package jakarta.persistence;

/**
 * Type for query parameter objects.
 * @param <T> the type of the parameter
 *
 * @see Query
 * @see TypedQuery
 *
 * @since 2.0
 */
public interface Parameter<T> {

    /**
     * Return the parameter name, or null if the parameter is
     * not a named parameter or no name has been assigned.
     * @return parameter name
     */
    String getName();

    /**
     * Return the parameter position, or null if the parameter
     * is not a positional parameter.
     * @return position of parameter
     */
    Integer getPosition();

    /**
     * Return the Java type of the parameter. Values bound to the
     * parameter must be assignable to this type.
     * This method is required to be supported for criteria queries
     * only.   Applications that use this method for Java
     * Persistence query language queries and native queries will
     * not be portable.
     * @return the Java type of the parameter
     * @throws IllegalStateException if invoked on a parameter
     *         obtained from a query language
     *         query or native query when the implementation does
     *         not support this use
     */
     Class<T> getParameterType();
}
```

3.10.6. StoredProcedureQuery Interface

```
package jakarta.persistence;

import java.util.Calendar;
import java.util.Date;
import java.util.List;

/**
 * Interface used to control stored procedure query execution.
 *
 * <p>
 * Stored procedure query execution may be controlled in accordance with
 * the following:
 * <ul>
 * <li>The <code>setParameter</code> methods are used to set the values of
 * all required <code>IN</code> and <code>INOUT</code> parameters.
 * It is not required to set the values of stored procedure parameters
 * for which default values have been defined by the stored procedure.</li>
 * <li>
 * When <code>getResultList</code> and <code>getSingleResult</code> are
 * called on a <code>StoredProcedureQuery</code> object, the provider
 * will call <code>execute</code> on an unexecuted stored procedure
 * query before processing <code>getResultList</code> or
 * <code>getSingleResult</code>.</li>
 * <li>
 * When <code>executeUpdate</code> is called on a
 * <code>StoredProcedureQuery</code> object, the provider will call
 * <code>execute</code> on an unexecuted stored procedure query
 *  followed by <code>getUpdateCount</code>.  The results of
 * <code>executeUpdate</code> will be those of <code>getUpdateCount</code>.</li>
 * <li>
 * The <code>execute</code> method supports both the simple case where
 * scalar results are passed back only via <code>INOUT</code> and
 * <code>OUT</code> parameters as well as the most general case
 * (multiple result sets and/or update counts, possibly also in
 * combination with output parameter values).</li>
 * <li>
 * The <code>execute</code> method returns true if the first result is a
 * result set, and false if it is an update count or there are no results
 * other than through <code>INOUT</code> and <code>OUT</code> parameters,
 * if any.</li>
 * <li>
 * If the <code>execute</code> method returns true, the pending result set
 * can be obtained by calling <code>getResultList</code> or
 * <code>getSingleResult</code>.</li>
 * <li>
 * The <code>hasMoreResults</code> method can then be used to test
 * for further results.</li>
 * <li>
```

```
 * If <code>execute</code> or <code>hasMoreResults</code> returns false,
 * the <code>getUpdateCount</code> method can be called to obtain the
 * pending result if it is an update count.  The <code>getUpdateCount</code>
 * method will return either the update count (zero or greater) or -1
 * if there is no update count (i.e., either the next result is a result set
 * or there is no next update count).</li>
 * <li>
 * For portability, results that correspond to JDBC result sets and
 * update counts need to be processed before the values of any
 * <code>INOUT</code> or <code>OUT</code> parameters are extracted.</li>
 * <li>
 * After results returned through <code>getResultList</code> and
 * <code>getUpdateCount</code> have been exhausted, results returned through
 * <code>INOUT</code> and <code>OUT</code> parameters can be retrieved.</li>
 * <li>
 * The <code>getOutputParameterValue</code> methods are used to retrieve
 * the values passed back from the procedure through <code>INOUT</code>
 * and <code>OUT</code> parameters.</li>
 * <li>
 * When using <code>REF_CURSOR</code> parameters for result sets the
 * update counts should be exhausted before calling <code>getResultList</code>
 * to retrieve the result set.  Alternatively, the <code>REF_CURSOR</code>
 * result set can be retrieved through <code>getOutputParameterValue</code>.
 * Result set mappings will be applied to results corresponding to
 * <code>REF_CURSOR</code> parameters in the order the <code>REF_CURSOR</code>
 * parameters were registered with the query.</li>
 * <li>
 * In the simplest case, where results are returned only via
 * <code>INOUT</code> and <code>OUT</code> parameters, <code>execute</code>
 * can be followed immediately by calls to
 * <code>getOutputParameterValue</code>.</li>
 * </ul>
 *
 * @see Query
 * @see Parameter
 *
 * @since 2.1
 */
public interface StoredProcedureQuery extends Query {

    /**
     * Set a query property or hint. The hints elements may be used
     * to specify query properties and hints. Properties defined by
     * this specification must be observed by the provider.
     * Vendor-specific hints that are not recognized by a provider
     * must be silently ignored. Portable applications should not
     * rely on the standard timeout hint. Depending on the database
     * in use, this hint may or may not be observed.
```

```
     * @param hintName  name of the property or hint
     * @param value  value for the property or hint
     * @return the same query instance
     * @throws IllegalArgumentException if the second argument is not
     *         valid for the implementation
     */
    StoredProcedureQuery setHint(String hintName, Object value);

    /**
     * Bind the value of a <code>Parameter</code> object.
     * @param param  parameter object
     * @param value  parameter value
     * @return the same query instance
     * @throws IllegalArgumentException if the parameter does not
     *         correspond to a parameter of the query
     */
    <T> StoredProcedureQuery setParameter(Parameter<T> param,
                                          T value);

    /**
     * Bind an instance of <code>java.util.Calendar</code> to a <code>Parameter</code>
object.
     * @param param parameter object
     * @param value  parameter value
     * @param temporalType  temporal type
     * @return the same query instance
     * @throws IllegalArgumentException if the parameter does not
     *         correspond to a parameter of the query
     */
    StoredProcedureQuery setParameter(Parameter<Calendar> param,
                                      Calendar value,
                                      TemporalType temporalType);

    /**
     * Bind an instance of <code>java.util.Date</code> to a <code>Parameter</code>
object.
     * @param param parameter object
     * @param value  parameter value
     * @param temporalType  temporal type
     * @return the same query instance
     * @throws IllegalArgumentException if the parameter does not
     *         correspond to a parameter of the query
     */
    StoredProcedureQuery setParameter(Parameter<Date> param,
                                      Date value,
                                      TemporalType temporalType);

    /**
```

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

    /**
     * Bind an instance of <code>java.util.Calendar</code> to a named parameter.
     * @param name  parameter name
     * @param value  parameter value
     * @param temporalType  temporal type
     * @return the same query instance
     * @throws IllegalArgumentException if the parameter name does
     *         not correspond to a parameter of the query or if the
     *         value argument is of incorrect type
     */
    StoredProcedureQuery setParameter(String name,
                                      Calendar value,
                                      TemporalType temporalType);

    /**
     * Bind an instance of <code>java.util.Date</code> to a named parameter.
     * @param name  parameter name
     * @param value  parameter value
     * @param temporalType  temporal type
     * @return the same query instance
     * @throws IllegalArgumentException if the parameter name does
     *         not correspond to a parameter of the query or if the
     *         value argument is of incorrect type
     */
    StoredProcedureQuery setParameter(String name,
                                      Date value,
                                      TemporalType temporalType);

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

```
    /**
     * Bind an instance of <code>java.util.Calendar</code> to a positional
     * parameter.
     * @param position  position
     * @param value  parameter value
     * @param temporalType  temporal type
     * @return the same query instance
     * @throws IllegalArgumentException if position does not
     *         correspond to a positional parameter of the query or
     *         if the value argument is of incorrect type
     */
    StoredProcedureQuery setParameter(int position,
                                      Calendar value,
                                      TemporalType temporalType);

    /**
     * Bind an instance of <code>java.util.Date</code> to a positional parameter.
     * @param position  position
     * @param value  parameter value
     * @param temporalType  temporal type
     * @return the same query instance
     * @throws IllegalArgumentException if position does not
     *         correspond to a positional parameter of the query or
     *         if the value argument is of incorrect type
     */
    StoredProcedureQuery setParameter(int position,
                                      Date value,
                                      TemporalType temporalType);

    /**
     * Set the flush mode type to be used for the query execution.
     * The flush mode type applies to the query regardless of the
     * flush mode type in use for the entity manager.
     * @param flushMode  flush mode
     * @return the same query instance
     */
    StoredProcedureQuery setFlushMode(FlushModeType flushMode);

    /**
     * Register a positional parameter.
     * All parameters must be registered.
     * @param position  parameter position
     * @param type  type of the parameter
     * @param mode  parameter mode
     * @return the same query instance
     */
    StoredProcedureQuery registerStoredProcedureParameter(
```

```
    int position,
    Class type,
    ParameterMode mode);

  /**
   * Register a named parameter.
   * @param parameterName  name of the parameter as registered or
   *              specified in metadata
   * @param type  type of the parameter
   * @param mode  parameter mode
   * @return the same query instance
   */
  StoredProcedureQuery registerStoredProcedureParameter(
    String parameterName,
    Class type,
    ParameterMode mode);

  /**
   *  Retrieve a value passed back from the procedure
   *  through an INOUT or OUT parameter.
   *  For portability, all results corresponding to result sets
   *  and update counts must be retrieved before the values of
   *  output parameters.
   *  @param position  parameter position
   *  @return the result that is passed back through the parameter
   *  @throws IllegalArgumentException if the position does
   *          not correspond to a parameter of the query or is
   *          not an INOUT or OUT parameter
   */
  Object getOutputParameterValue(int position);

  /**
   *  Retrieve a value passed back from the procedure
   *  through an INOUT or OUT parameter.
   *  For portability, all results corresponding to result sets
   *  and update counts must be retrieved before the values of
   *  output parameters.
   *  @param parameterName  name of the parameter as registered or
   *               specified in metadata
   *  @return the result that is passed back through the parameter
   *  @throws IllegalArgumentException if the parameter name does
   *          not correspond to a parameter of the query or is
   *          not an INOUT or OUT parameter
   */
  Object getOutputParameterValue(String parameterName);

  /**
   * Return true if the first result corresponds to a result set,
```

```
     * and false if it is an update count or if there are no results
     * other than through INOUT and OUT parameters, if any.
     * @return  true if first result corresponds to result set
     * @throws QueryTimeoutException if the query execution exceeds
     *         the query timeout value set and only the statement is
     *         rolled back
     * @throws PersistenceException if the query execution exceeds
     *         the query timeout value set and the transaction
     *         is rolled back
     */
    boolean execute();

    /**
     * Return the update count of -1 if there is no pending result or
     * if the first result is not an update count.  The provider will
     * call <code>execute</code> on the query if needed.
     * @return the update count or -1 if there is no pending result
     * or if the next result is not an update count.
     * @throws TransactionRequiredException if there is
     *         no transaction or the persistence context has not
     *         been joined to the transaction
     * @throws QueryTimeoutException if the statement execution
     *         exceeds the query timeout value set and only
     *         the statement is rolled back
     * @throws PersistenceException if the query execution exceeds
     *         the query timeout value set and the transaction
     *         is rolled back
     */
    int executeUpdate();

    /**
     * Retrieve the list of results from the next result set.
     * The provider will call <code>execute</code> on the query
     * if needed.
     * A <code>REF_CURSOR</code> result set, if any, will be retrieved
     * in the order the <code>REF_CURSOR</code> parameter was
     * registered with the query.
     * @return a list of the results or null is the next item is not
     * a result set
     * @throws QueryTimeoutException if the query execution exceeds
     *         the query timeout value set and only the statement is
     *         rolled back
     * @throws PersistenceException if the query execution exceeds
     *         the query timeout value set and the transaction
     *         is rolled back
     */
    List getResultList();
```

```
    /**
     * Retrieve a single result from the next result set.
     * The provider will call <code>execute</code> on the query
     * if needed.
     * A <code>REF_CURSOR</code> result set, if any, will be retrieved
     * in the order the <code>REF_CURSOR</code> parameter was
     * registered with the query.
     * @return the result or null if the next item is not a result set
     * @throws NoResultException if there is no result in the next
     *         result set
     * @throws NonUniqueResultException if more than one result
     * @throws QueryTimeoutException if the query execution exceeds
     *         the query timeout value set and only the statement is
     *         rolled back
     * @throws PersistenceException if the query execution exceeds
     *         the query timeout value set and the transaction
     *         is rolled back
     */
    Object getSingleResult();

    /**
     * Return true if the next result corresponds to a result set,
     * and false if it is an update count or if there are no results
     * other than through INOUT and OUT parameters, if any.
     * @return  true if next result corresponds to result set
     * @throws QueryTimeoutException if the query execution exceeds
     *         the query timeout value set and only the statement is
     *         rolled back
     * @throws PersistenceException if the query execution exceeds
     *         the query timeout value set and the transaction
     *         is rolled back
     */
    boolean hasMoreResults();

    /**
     * Return the update count or  -1 if there is no pending result
     * or if the next result is not an update count.
     * @return  update count or -1 if there is no pending result or if
     *          the next result is not an update count
     * @throws QueryTimeoutException if the query execution exceeds
     *         the query timeout value set and only the statement is
     *         rolled back
     * @throws PersistenceException if the query execution exceeds
     *         the query timeout value set and the transaction
     *         is rolled back
     */
    int getUpdateCount();
```

```
}
```

3.10.7. Query Execution

Java Persistence query language, Criteria API, and native SQL select queries are executed using the *getResultList* and *getSingleResult* methods. Update and delete operations (update and delete "queries") are executed using the *executeUpdate* method.

- For *TypedQuery* instances, the query result type is determined in the case of criteria queries by the type of the query specified when the *CriteriaQuery* object is created, as described in Section 6.5.1. In the case of Java Persistence query language queries, the type of the result is determined by the *resultClass* argument to the *createQuery* or *createNamedQuery* method, and the select list of the query must contain only a single item which must be assignable to the specified type.
- For *Query* instances, the elements of a query result whose select list consists of more than one select expression are of type *Object[]*. If the select list consists of only one select expression, the elements of the query result are of type *Object*. When native SQL queries are used, the SQL result set mapping (see Section 3.10.16), determines how many items (entities, scalar values, etc.) are returned. If multiple items are returned, the elements of the query result are of type *Object[]*. If only a single item is returned as a result of the SQL result set mapping or if a result class is specified, the elements of the query result are of type *Object*.

Stored procedure queries can be executed using the *getResultList*, *getSingleResult*, and *execute* methods. Stored procedures that perform only updates or deletes can be executed using the *executeUpdate* method. Stored procedure query execution is described in detail in Section 3.10.17.3.

An *IllegalArgumentException* is thrown if a parameter instance is specified that does not correspond to a parameter of the query, if a parameter name is specified that does not correspond to a named parameter of the query, if a positional value is specified that does not correspond to a positional parameter of the query, 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. See Section 3.10.11, Section 3.10.12, and Section 3.10.13 for supported parameter usage.

The effect of applying *setMaxResults* or *setFirstResult* to a query involving fetch joins over collections is undefined. The use of *setMaxResults* and *setFirstResult* is not supported for stored procedure queries.

Query and *TypedQuery* methods other than the *executeUpdate* method are not required to be invoked within a transaction context, unless a lock mode other than *LockModeType.NONE* has been specified for the query. In particular, the *getResultList* and *getSingleResult* methods are not required to be invoked within a transaction context unless such a lock mode has been specified for the query [52: A lock mode is specified for a query by means of the *setLockMode* method or by specifying the lock mode in the *NamedQuery* annotation.]. If an entity manager with transaction-scoped persistence context is in use, the resulting entities will be detached; if an entity manager with an extended persistence context is used, they will be managed. See Chapter 7 for further discussion of entity manager use outside a transaction and persistence context types.

Whether a *StoredProcedureQuery* should be invoked in a transaction context should be determined by the transactional semantics and/or requirements of the stored procedure implementation and the database in use. In particular, problems may occur if the stored procedure initiates a transaction and a transaction is already in effect. The state of any entities returned by the stored procedure query invocation is determined as decribed above.

Runtime exceptions other than the *NoResultException*, *NonUniqueResultException*, *QueryTimeoutException*, and *LockTimeoutException* thrown by the methods of the *Query*, *TypedQuery*, and *StoredProcedureQuery* interfaces other than those methods specified below cause the current transaction to be marked for rollback if the persistence context is joined to the transaction. On database platforms on which a query timeout causes transaction rollback, the persistence provider must throw the *PersistenceException* instead of the *QueryTimeoutException*.

Runtime exceptions thrown by the following methods of the *Query*, *TypedQuery*, and *StoredProcedureQuery* interfaces do not cause the current transaction to be marked for rollback: *getParameters*, *getParameter*, *getParameterValue*, *getOutputParameterValue*, *getLockMode*.

Runtime exceptions thrown by the methods of the *Tuple*, *TupleElement*, and *Parameter* interfaces do not cause the current transaction to be marked for rollback.

3.10.7.1. 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();
}
```

3.10.8. Queries and Flush Mode

The flush mode setting affects the result of a query as follows.

When queries are executed within a transaction, if *FlushModeType.AUTO* is set on the *Query*, *TypedQuery*, or *StoredProcedureQuery* object, or if the flush mode setting for the persistence context is *AUTO* (the default) and a flush mode setting has not been specified for the query object, the persistence provider is responsible for ensuring that all updates to the state of all entities in the persistence context which could potentially affect the result of the query are visible to the processing of the query. The persistence provider implementation may achieve this by flushing those entities to the database or by some other means. If *FlushModeType.COMMIT* is set, the effect of updates made to entities in the persistence context upon queries is unspecified.

If the persistence context has not been joined to the current transaction, the persistence provider must not flush to the database regardless of the flush mode setting.

```
package jakarta.persistence;

public enum FlushModeType {

    /**
     * Flushing to occur at transaction commit.  The provider may flush
     * at other times, but is not required to.
     */
    COMMIT,

    /**
     * (Default) Flushing to occur at query execution.
     */
    AUTO
}
```

If there is no transaction active, the persistence provider must not flush to the database.

3.10.9. Queries and Lock Mode

The *setLockMode* method of the *Query* or *TypedQuery* interface or the *lockMode* element of the *NamedQuery* annotation may be used to lock the results of a query. A lock is obtained for each entity specified in the query result (including entities passed to constructors in the query SELECT clause). [53: Note that the setLockMode method may be called more than once (with different values) on a Query or TypedQuery object.]

If the lock mode type is *PESSIMISTIC_READ*, *PESSIMISTIC_WRITE*, or *PESSIMISTIC_FORCE_INCREMENT*, and the query returns scalar data (e.g., the values of entity field or properties, including scalar data passed to constructors in the query SELECT clause), the underlying database rows will be locked [54: Note that locking will not occur for data passed to aggregate functions. Further, queries involving aggregates with pessimistic locking may not be supported on all database platforms.], but the version columns (if any) for any entities corresponding to such scalar data will not be updated unless the entities themselves are also otherwise retrieved and updated.

If the lock mode type is *OPTIMISTIC* or *OPTIMISTIC_FORCE_INCREMENT*, and the query returns scalar data, any entities returned by the query will be locked, but no locking will occur for scalar data that does not correspond to the state of any entity instance in the query result.

If a lock mode other than *NONE* is specified for a query, the query must be executed within a transaction (and the persistence context must be joined to the transaction) or the *TransactionRequiredException* will be thrown.

Locking is supported for Java Persistence query language queries and criteria queries only. If the *setLockMode* or *getLockMode* method is invoked on a query that is not a Java Persistence query language select query or a criteria query, the *IllegalStateException* may be thrown or the query

execution will fail.

3.10.10. Query Hints

The following hint is defined by this specification for use in query configuration.

```
javax.persistence.query.timeout // time in milliseconds
```

This hint may be used with the *Query*, *TypedQuery*, or *StoredProcedureQuery setHint* method or the *NamedQuery*, *NamedNativeQuery*, and *NamedStoredProcedureQuery* annotations. It may also be passed as a property to the *Persistence.createEntityManagerFactory* method and used in the *properties* element of the *persistence.xml* file. See Section 3.10.1, Section 8.2.1.9, Section 9.7, Section 10.4. When used in the *createEntityManagerFactory* method, the *persistence.xml* file, and annotations, the *timeout* hint serves as a default value which can be selectively overridden by use in the *setHint* method.

Portable applications should not rely on this hint. Depending on the persistence provider and database in use, the hint may or may not be observed.

Vendors are permitted to support the use of additional, vendor-specific hints. Vendor-specific hints must not use the *javax.persistence* namespace. Vendor-specific hints must be ignored if they are not understood.

3.10.11. Parameter Objects

Parameter objects can be used for criteria queries and for Java Persistence query language queries.

Implementations may support the use of *Parameter* objects for native queries, however support for *Parameter* objects with native queries is not required by this specification. The use of *Parameter* objects for native queries will not be portable. The mixing of parameter objects with named or positional parameters is undefined.

Portable applications should not attempt to reuse a *Parameter* object obtained from a *Query* or *TypedQuery* instance in the context of a different *Query* or *TypedQuery* instance.

3.10.12. Named Parameters

Named parameters can be used for Java Persistence query language queries, for criteria queries (although use of *Parameter* objects is to be preferred), and for stored procedure queries that support named parameters.

Named parameters follow the rules for identifiers defined in [a4760]. Named parameters are case-sensitive. The mixing of named and positional parameters is undefined.

A named parameter of a Java Persistence query language query is an identifier that is prefixed by the " *:* " symbol. The parameter names passed to the *setParameter* methods of the *Query* and *TypedQuery*

interfaces do not include this " *:* " prefix.

3.10.13. Positional Parameters

Only positional parameter binding and positional access to result items may be portably used for native queries, except for stored procedure queries for which named parameters have been defined. When binding the values of positional parameters, the numbering starts as “ *1* ”. It is assumed that for native queries the parameters themselves use the SQL syntax (i.e., “ *?* ”, rather than “ *?1* ”).

The use of positional parameters is not supported for criteria queries.

3.10.14. Named Queries

Named queries are static queries expressed in metadata or queries registered by means of the EntityManagerFactory *addNamedQuery* method. Named queries can be defined in the Java Persistence query language or in SQL. Query names are scoped to the persistence unit.

The following is an example of the definition of a named query defined in metadata:

```
@NamedQuery(
    name="findAllCustomersWithName",
    query="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();
```

3.10.15. 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 following query returns the average salary of all employees, including subtypes of *Employee*, such as *Manager* and *Exempt*.

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

Entity type expressions, described in Section 4.6.17.5, as well as the use of downcasting, described in Section 4.4.9, can be used to restrict query polymorphism.

3.10.16. SQL Queries

Queries may be expressed in native SQL. The result of a native SQL query may consist of entities, unmanaged instances created via constructors, scalar values, or some combination of these.

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 the Java Persistence query language cannot be used). Native SQL queries are not expected to be portable across databases.

3.10.16.1. Returning Managed Entities from Native Queries

The persistence provider is responsible for performing the mapping between the values returned by the SQL query and entity attributes in accordance with the object/relational mapping metadata for the entity or entities. In particular, the names of the columns in the SQL result are used to map to the entity attributes as defined by this metadata. This mapping includes the mapping of the attributes of any embeddable classes that are part of the non-collection-valued entity state and attributes corresponding to foreign keys contained as part of the entity state [55: Support for joins is currently limited to single-valued relationships that are mapped directly—i.e., not via join tables.].

When an entity is to be returned from a native query, 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.

In the simplest case—i.e., when the results of the query are limited to entities of a single entity class and the mapping information can be derived from the columns of the SQL result and the object/relational mapping metadata—it is sufficient to specify only the expected class of the entity result.

The following example illustrates the case where 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 *SqlResultSetMapping* metadata annotation—which is designed to handle more complex cases—can be used as an alternative here. See Section 10.4.4 for the definition of the *SqlResultSetMapping* metadata annotation and related annotations.

For the query shown above, the *SqlResultSetMapping* metadata for the query result type might be specified as follows:

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

The same results as produced by the query above can then obtained by the following:

```
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");
```

When multiple entities are returned by a SQL query or when the column names of the SQL result do not correspond to those of the object/relational mapping metadata, a *SqlResultSetMapping* metadata definition must be provided to specify the entity mapping.

The following query and *SqlResultSetMapping* metadata illustrates the return of multiple entity types. It 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 the column names of the SQL result do not correspond to those of the object/relational mapping metadata, more explicit SQL result mapping metadata must be provided to enable the persistence provider runtime to map the JDBC results into the expected objects. This might arise, for example, when column aliases must be used in the SQL SELECT clause when the SQL result would otherwise contain multiple columns of the same name or when columns in the SQL result are the results of operators or functions. The *FieldResult* annotation element within the *EntityResult* annotation is used

to specify the mapping of such columns to entity attributes.

The following example combining multiple entity types includes aliases in the SQL statement. This requires that the column names be explicitly mapped to the entity fields corresponding to those columns. 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)
})
```

When the returned entity type contains an embeddable class, the *FieldResult* element must use a dot (“ . ”) notation to indicate which column maps to which field or property of the contained embeddable.

Example:

```
Query q = em.createNativeQuery(
    "SELECT c.id AS customer_id, " +
        "c.street AS customer_street, " +
        "c.city AS customer_city, " +
        "c.state AS customer_state, " +
        "c.status AS customer_status " +
        "FROM Customer c " +
        "WHERE c.status = 'GOLD' ",
    "CustomerResults");

@SqlResultSetMapping(name="CustomerResults", entities={
    @EntityResult(entityClass=com.acme.Customer.class, fields={
        @FieldResult(name="id", column="customer_id"),
        @FieldResult(name="address.street", column="customer_street"),
        @FieldResult(name="address.city", column="customer_city"),
        @FieldResult(name="address.state", column="customer_state"),
        @FieldResult(name="status", column="customer_status")
    })
})
```

When the returned entity type is the owner of a single-valued relationship and the foreign key is a composite foreign key (composed of multiple columns), a *FieldResult* element should be used for each of the foreign key columns. The *FieldResult* element must use the dot (" . ") notation form to indicate the column that maps to each property or field of the target entity primary key.

If the target entity has a primary key of type *IdClass*, this specification takes the form of the name of the field or property for the relationship, followed by a dot (" . "), followed by the name of the field or property of the primary key in the target entity. The latter will be annotated with *Id*, as specified in Section 11.1.22.

Example:

```
Query q = em.createNativeQuery(
    "SELECT o.id AS order_id, " +
        "o.quantity AS order_quantity, " +
        "o.item_id AS order_item_id, " +
        "o.item_name AS order_item_name, " +
        "i.id, i.name, i.description " +
        "FROM Order o, Item i " +
        "WHERE (order_quantity > 25) AND (order_item_id = i.id) " +
        "AND (order_item_name = i.name)",
    "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.id", column="order_item_id")}),
        @FieldResult(name="item.name", column="order_item_name")}),
    @EntityResult(entityClass=com.acme.Item.class)
})
```

If the target entity has a primary key of type *EmbeddedId*, this specification is composed of the name of the field or property for the relationship, followed by a dot (“ . ”), followed by the name or the field or property of the primary key (i.e., the name of the field or property annotated as *EmbeddedId*), followed by the name of the corresponding field or property of the embedded primary key class.

Example:

```
Query q = em.createNativeQuery(
    "SELECT o.id AS order_id, " +
        "o.quantity AS order_quantity, " +
        "o.item_id AS order_item_id, " +
        "o.item_name AS order_item_name, " +
        "i.id, i.name, i.description " +
        "FROM Order o, Item i " +
        "WHERE (order_quantity > 25) AND (order_item_id = i.id) AND (order_item_name =
i.name)",
    "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.itemPk.id", column="order_item_id")}),
        @FieldResult(name="item.itemPk.name", column="order_item_name")}),
    @EntityResult(entityClass=com.acme.Item.class)
})
```

The *FieldResult* elements for the composite foreign key are combined to form the primary key *EmbeddedId* class for the target entity. This may then be used to subsequently retrieve the entity if the relationship is to be eagerly loaded.

The dot-notation form is not required to be supported for any usage other than for embeddables, composite foreign keys, or composite primary keys.

3.10.16.2. Returning Unmanaged Instances

Instances of other classes (including non-managed entity instances) as well as scalar results can be returned by a native query. These can be used singly, or in combination, including with entity results.

Scalar Results

Scalar results can be included in the query result by specifying the *ColumnResult* annotation element of the *SqlResultSetMapping* annotation. The intended type of the result can be specified using the *type* element of the *ColumnResult* annotation.

```
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, " +
        "i.availabilityDate AS item_shipdate " +
        "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"),
        @ColumnResult(name="item_shipdate", type=java.util.Date.class)
    })
```

Constructor Results

The mapping to constructors is specified using the *ConstructorResult* annotation element of the *SqlResultSetMapping* annotation. The *targetClass* element of the *ConstructorResult* annotation specifies the class whose constructor corresponds to the specified columns. All columns corresponding to arguments of the intended constructor must be specified using the *columns* element of the *ConstructorResult* annotation in the same order as that of the argument list of the constructor. Any entities returned as constructor results will be in either the new or the detached state, depending on whether a primary key is retrieved for the constructed object.

Example:

```
Query q = em.createNativeQuery(
    "SELECT c.id, c.name, COUNT(o) as orderCount, AVG(o.price) AS avgOrder " +
        "FROM Customer c, Orders o " +
        "WHERE o.cid = c.id " +
        "GROUP BY c.id, c.name",
    "CustomerDetailsResult");

@SqlResultSetMapping(name="CustomerDetailsResult", classes={
    @ConstructorResult(targetClass=com.acme.CustomerDetails.class, columns={
        @ColumnResult(name="id"),
        @ColumnResult(name="name"),
        @ColumnResult(name="orderCount"),
        @ColumnResult(name="avgOrder", type=Double.class)})
})
```

3.10.16.3. Combinations of Result Types

When a *SqlResultSetMapping* specifies more than one mapping type (i.e., more than one of *EntityResult*, *ConstructorResult*, *ColumnResult*), then for each row in the SQL result, the query execution will result in an *Object[]* instance whose elements are as follows, in order: any entity results (in the order in which they are defined in the *entities* element); any instances of classes corresponding to constructor results (in the order defined in the *classes* element); and any instances corresponding to column results (in the order defined in the *columns* element). If there are any columns whose result mappings have not been specified, they are ignored.

3.10.16.4. Restrictions

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. A SQL result set mapping must not be used to map results to the non-persistent state of an entity.

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

3.10.17. Stored Procedures

The *StoredProcedureQuery* interface supports the use of database stored procedures.

Stored procedures can be specified either by means of the *NamedStoredProcedureQuery* annotation or dynamically. Annotations for the specification of stored procedures are described in Section 10.4.3.

3.10.17.1. Named Stored Procedure Queries

Unlike in the case of a named native query, the *NamedStoredProcedureQuery* annotation names a stored procedure that exists in the database rather than providing a stored procedure definition. The

NamedStoredProcedureQuery annotation specifies the types of all parameters to the stored procedure, their corresponding parameter modes (IN, OUT, INOUT, REF_CURSOR [56: Note that REF_CURSOR parameters are used by some databases to return result sets from stored procedures.]), and how result sets, if any, are to be mapped. The name that is assigned to the stored procedure in the *NamedStoredProcedureQuery* annotation is passed as an argument to the *createNamedStoredProcedureQuery* method to create an executable *StoredProcedureQuery* object.

A stored procedure may return more than one result set. As with native queries, the mapping of result sets can be specified either in terms of a *resultClasses* or as a *resultSetMappings* annotation element. If there are multiple result sets, it is assumed that they will be mapped using the same mechanism — e.g., all via a set of result class mappings or all via a set of result set mappings. The order of the specification of these mappings must be the same as the order in which the result sets will be returned by the stored procedure invocation. If the stored procedure returns one or more result sets and no *resultClasses* or *resultSetMappings* element has been specified, any result set will be returned as a list of type *Object[]*. The combining of different strategies for the mapping of stored procedure result sets is undefined.

StoredProcedureParameter metadata needs to be provided for all parameters. Parameters must be specified in the order in which they occur in the parameter list of the stored procedure. If parameter names are used, the parameter name is used to bind the parameter value and to extract the output value (if the parameter is an INOUT or OUT parameter). If parameter names are not specified, it is assumed that positional parameters are used. The mixing of named and positional parameters is undefined.

3.10.17.2. Dynamically-specified Stored Procedure Queries

If the stored procedure is not defined using metadata, parameter and result set information must be provided dynamically.

All parameters of a dynamically-specified stored procedure query must be registered using the *registerStoredProcedureParameter* method of the *StoredProcedureQuery* interface.

Result set mapping information can be provided by means of the *createStoredProcedureQuery* method.

3.10.17.3. Stored Procedure Query Execution

Stored procedure query execution can be controlled as described below.

The *setParameter* methods are used to set the values of all required IN and INOUT parameters. It is not required to set the values of stored procedure parameters for which default values have been defined by the stored procedure.

When *getResultList* and *getSingleResult* are called on a *StoredProcedureQuery* object, the persistence provider will call *execute* on an unexecuted stored procedure query before processing *getResultList* or *getSingleResult*.

When *executeUpdate* is called on a *StoredProcedureQuery* object, the persistence provider will call

execute on an unexecuted stored procedure query followed by *getUpdateCount*. The results of *executeUpdate* will be those of *getUpdateCount*.

The *execute* method supports both the simple case where scalar results are passed back only via INOUT and OUT parameters as well as the most general case (multiple result sets and/or update counts, possibly also in combination with output parameter values).

The *execute* method returns *true* if the first result is a result set, and *false* if it is an update count or there are no results other than through INOUT and OUT parameters, if any.

If the *execute* method returns *true*, the pending result set can be obtained by calling *getResultList* or *getSingleResult*. The *hasMoreResults* method can then be used to test for further results.

If *execute* or *hasMoreResults* returns *false*, the *getUpdateCount* method can be called to obtain the pending result if it is an update count. The *getUpdateCount* method will return either the update count (zero or greater) or -1 if there is no update count (i.e., either the next result is a result set or there is no next update count).

For portability, results that correspond to JDBC result sets and update counts need to be processed before the values of any INOUT or OUT parameters are extracted.

After results returned through *getResultList* and *getUpdateCount* have been exhausted, results returned through INOUT and OUT parameters can be retrieved.

The *getOutputParameterValue* methods are used to retrieve the values passed back from the procedure through INOUT and OUT parameters.

When using *REF_CURSOR* parameters for result sets, the update counts should be exhausted before calling *getResultList* to retrieve the result set. Alternatively, the *REF_CURSOR* result set can be retrieved through *getOutputParameterValue*. Result set mappings will be applied to results corresponding to *REF_CURSOR* parameters in the order the *REF_CURSOR* parameters were registered with the query.

In the simplest case, where results are returned only via INOUT and OUT parameters, *execute* can be followed immediately by calls to *getOutputParameterValue*.

3.11. Summary of Exceptions

The following is a summary of the exceptions defined by this specification:

PersistenceException

The *PersistenceException* is thrown by the persistence provider when a problem occurs. It may be thrown to report that the invoked operation could not complete because of an unexpected error (e.g., failure of the persistence provider to open a database connection).

All other exceptions defined by this specification are subclasses of the *PersistenceException*. All instances of *PersistenceException* except for instances of *NoResultException*,

NonUniqueResultException, LockTimeoutException , and *QueryTimeoutException* will cause the current transaction, if one is active and the persistence context has been joined to it, to be marked for rollback.

TransactionRequiredException

The *TransactionRequiredException* is thrown by the persistence provider when a transaction is required but is not active.

OptimisticLockException

The *OptimisticLockException* is thrown by the persistence provider when an optimistic locking conflict occurs. This exception may be thrown as part of an API call, at flush, or at commit time. The current transaction, if one is active, will be marked for rollback.

PessimisticLockException

The *PessimisticLockException* is thrown by the persistence provider when a pessimistic locking conflict occurs. The current transaction will be marked for rollback. Typically the *PessimisticLockException* occurs because the database transaction has been rolled back due to deadlock or because the database uses transaction-level rollback when a pessimistic lock cannot be granted.

LockTimeoutException

The *LockTimeoutException* is thrown by the persistence provider when a pessimistic locking conflict occurs that does not result in transaction rollback. Typically this occurs because the database uses statement-level rollback when a pessimistic lock cannot be granted (and there is no deadlock). The *LockTimeoutException* does not cause the current transaction to be marked for rollback.

RollbackException

The *RollbackException* is thrown by the persistence provider when *EntityTransaction.commit* fails. __

EntityExistsException

The *EntityExistsException* may thrown by the persistence provider when the *persist* operation is invoked and the entity already exists. The *EntityExistsException* may be thrown when the persist operation is invoked, or the *EntityExistsException* or another *PersistenceException* may be thrown at commit time. The current transaction, if one is active and the persistence context has been joined to it, will be marked for rollback.

EntityNotFoundException

The *EntityNotFoundException* is thrown by the persistence provider when an entity reference obtained by *getReference* is accessed but the entity does not exist. It is thrown by the *refresh* operation when the entity no longer exists in the database. It is also thrown by the *lock* operation when pessimistic locking is used and the entity no longer exists in the database. The current transaction, if one is active and the persistence context has been joined to it, will be marked for rollback.

NoResultException

The *NoResultException* is thrown by the persistence provider when *Query.getSingleResult* or *TypedQuery.getSingleResult* is invoked and there is no result to return. This exception will not cause the current transaction, if one is active, to be marked for rollback.

NonUniqueResultException

The *NonUniqueResultException* is thrown by the persistence provider when *Query.getSingleResult* or *TypedQuery.getSingleResult* is invoked and there is more than one result from the query. This exception will not cause the current transaction, if one is active, to be marked for rollback.

QueryTimeoutException

The *QueryTimeoutException* is thrown by the persistence provider when a query times out and only the statement is rolled back. The *QueryTimeoutException* does not cause the current transaction, if one is active, to be marked for rollback.

Chapter 4. Query Language

The Java Persistence query language is a string-based query language used to define queries over entities and their persistent state. It enables the application developer to specify the semantics of queries in a portable way, independent of the particular database schema in use in an enterprise environment. The full range of the language may be used in both static and dynamic queries.

This chapter provides the full definition of the Java Persistence query language.

4.1. Overview

The Java Persistence query language is a query specification language for string-based dynamic queries and static queries expressed through metadata. It is used to define queries over the persistent entities defined by this specification and their persistent state and relationships.

The Java Persistence query language 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 query language uses the abstract persistence schema of entities, including their embedded objects and relationships, for its data model, and it defines operators and expressions based on this data model. It uses a SQL-like syntax to select objects or values based on abstract schema types and relationships. It is possible to parse and validate queries before entities are deployed.

The term abstract persistence schema refers to the persistent schema abstraction (persistent entities, their state, and their relationships) over which Java Persistence queries operate. Queries over this persistent schema abstraction are translated into queries that are executed over the database schema to which entities are mapped.

Queries may be defined in metadata annotations or the 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. Path expressions 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.

4.2. Statement Types

A Java Persistence query language 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, a query language statement is defined as:

```
QL_statement :: = select_statement | update_statement | delete_statement
```

Any Java Persistence query language statement may be constructed dynamically or may be statically defined in a metadata annotation or XML descriptor element.

All statement types may have parameters.

4.2.1. Select Statements

A 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, a 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 4.10.

4.3. Abstract Schema Types and Query Domains

The Java Persistence query language is a typed language, and every expression 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 attributes evaluate, and the types of literals.

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

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

- For every non-relationship persistent field or get accessor method (for a persistent property) of the 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.
- For every persistent relationship field or get accessor method (for a persistent relationship property) of the 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).

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

The domain of a query consists of the abstract schema types of all entities and embeddables that are defined in the same persistence unit.

The domain of a query may be restricted by the *navigability* of the relationships of the entity and associated embeddable classes on which it is based. The association fields of an entity's or embeddable'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 query strings by their entity names. The entity 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. Entity names are scoped within the persistence unit and must be unique within the persistence unit.

4.3.2. Example

This example assumes that the application developer provides several entity classes, representing orders, products, and line items, and an embeddable address class representing shipping addresses and billing addresses. The abstract schema types for the entities are *Order*, *Product*, and *LineItem* respectively. There is a one-to-many relationship between *Order* and *LineItem*. The entity *LineItem* is related to *Product* in a many-to-one relationship. The classes are logically in the same persistence unit, as shown in Figure 1.

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
```

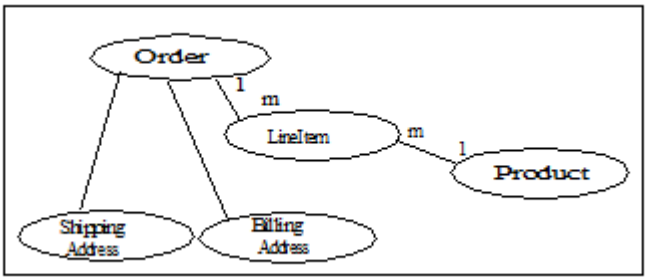

Figure 1. Abstract Persistence Schema of Several Entities with Defined in the Same Persistence Unit.

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 reserved identifiers, such as DISTINCT, FROM, AS, JOIN, WHERE, and FALSE appear in upper case in this example, reserved identifiers are case insensitive. [57: This chapter uses the convention that reserved identifiers appear in upper case in the examples and BNF for the language.]

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 schema 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. A query 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 entity 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 a 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. (See [a4792].)

Identification variables designate instances of a particular 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 ::= entity_name [AS] identification_variable

join ::= join_spec join_association_path_expression [AS] identification_variable
[join_condition]

fetch_join ::= join_spec FETCH join_association_path_expression

join_association_path_expression ::=
    join_collection_valued_path_expression |
    join_single_valued_path_expression |
    TREAT(join_collection_valued_path_expression AS subtype) |
    TREAT(join_single_valued_path_expression AS subtype)

join_collection_valued_path_expression ::=
identification_variable.{single_valued_embeddable_object_field.}*collection_valued_field

join_single_valued_path_expression ::=
identification_variable.{single_valued_embeddable_object_field.}*single_valued_object_fie
ld

join_spec ::= [LEFT [OUTER] | INNER] JOIN

join_condition ::= ON conditional_expression

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 the Java Persistence query language.

The following are reserved identifiers: ABS, ALL, AND, ANY, AS, ASC, AVG, BETWEEN, BIT_LENGTH [58: BIT_LENGTH, CHAR_LENGTH, CHARACTER_LENGTH, POSITION, and UNKNOWN are not currently

used: they are reserved for future use.], BOTH, BY, CASE, CHAR_LENGTH, CHARACTER_LENGTH, CLASS, COALESCE, CONCAT, COUNT, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, DELETE, DESC, DISTINCT, ELSE, EMPTY, END, ENTRY, ESCAPE, EXISTS, FALSE, FETCH, FROM, FUNCTION, GROUP, HAVING, IN, INDEX, INNER, IS, JOIN, KEY, LEADING, LEFT, LENGTH, LIKE, LOCATE, LOWER, MAX, MEMBER, MIN, MOD, NEW, NOT, NULL, NULLIF, OBJECT, OF, ON, OR, ORDER, OUTER, POSITION, SELECT, SET, SIZE, SOME, SQRT, SUBSTRING, SUM, THEN, TRAILING, TREAT, TRIM, TRUE, TYPE, UNKNOWN, UPDATE, UPPER, VALUE, WHEN, WHERE.

Reserved identifiers are case insensitive. Reserved identifiers must not be used as identification variables or result variables (see Section 4.8).

It is recommended that SQL key words other than those listed above not be used as identification variables in queries because they may be used as 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 a 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 entity in the same persistence unit.

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 can range over an entity, embeddable, or basic abstract schema type. An identification variable designates an instance of an abstract schema type or an element of a collection of abstract schema type instances.

Note that for identification variables referring to an instance of an association or collection represented as a *java.util.Map*, the identification variable is of the abstract schema type of the map

value.

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.

All identification variables used in the SELECT, WHERE, ORDER BY, GROUP BY, or HAVING clause of a SELECT or DELETE statement must be declared in the FROM clause. 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 these clauses. 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.

An identification variable is scoped to the query (or subquery) in which it is defined and is also visible to any subqueries within that query scope that do not define an identification variable of the same name.

4.4.3. Range Variable Declarations

The syntax for declaring an identification variable as a range variable is similar to that of SQL; optionally, it uses the AS keyword. A range variable designates an entity abstract schema type. [59: A range variable must not designate an embeddable class abstract schema type.]

```
range_variable_declaration ::= entity_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. The type of a path expression that navigates to an association field may be specified as a subtype of the declared type of the association field by means of the TREAT operator. See Section 4.4.9.

An identification variable qualified by the KEY, VALUE, or ENTRY operator is a path expression. The KEY, VALUE, and ENTRY operators may only be applied to identification variables that correspond to map-valued associations or map-valued element collections. The type of the path expression is the type computed as the result of the operation; that is, the abstract schema type of the field that is the value of the KEY, VALUE, or ENTRY operator (the map key, map value, or map entry respectively). [60: Note that use of VALUE is optional, as an identification variable referring to an association of type *java.util.Map* is of the abstract schema type of the map value. (See Section 4.4.2.)]

In the following query, photos is a map from photo label to filename.

```
SELECT i.name, VALUE(p)
FROM Item i JOIN i.photos p
WHERE KEY(p) LIKE '%egret'
```

In the above query the identification variable *p* designates an abstract schema type corresponding to the map *value*. The results of *VALUE(p)* and *KEY(p)* are the map value and the map key associated with *p*, respectively. The following query is equivalent:

```
SELECT i.name, p
FROM Item i JOIN i.photos p
WHERE KEY(p) LIKE '%egret'
```

A path expression using the KEY or VALUE operator can be further composed. A path expression using the ENTRY operator is terminal. It cannot be further composed and can only appear in the SELECT list of a query.

The syntax for qualified identification variables is as follows.

```
qualified_identification_variable ::=
    map_field_identification_variable |
    ENTRY(identification_variable)

map_field_identification_variable ::=
    KEY(identification_variable) |
    VALUE(identification_variable)
```

Depending on navigability, a path expression that leads to an association field or to a field whose type is an embeddable class 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).

In the following example, *contactInfo* denotes an embeddable class consisting of an address and set of phones. *Phone* is an entity.

```
SELECT p.vendor
FROM Employee e JOIN e.contactInfo.phones p
WHERE e.contactInfo.address.zipcode = '95054'
```

Path expression navigability is composed using “inner join” semantics. That is, if the value of a non-terminal 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 following query is equivalent to the query above:

```
SELECT p.vendor
FROM Employee e JOIN e.contactInfo c JOIN c.phones p
WHERE e.contactInfo.address.zipcode = '95054'
```

4.4.4.1. Path Expression Syntax

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

An identification variable used in a *single_valued_object_path_expression* or in a *collection_valued_path_expression* may be an unqualified identification variable or an identification variable to which the KEY or VALUE function has been applied.

```
general_identification_variable ::=
    identification_variable |
    map_field_identification_variable
```

The type of an entity-valued path expression or an entity-valued subpath of a path expression used in a WHERE clause may be specified as a subtype of the corresponding declared type by means of the TREAT operator. See Section 4.4.9.

```
general_subpath ::= simple_subpath | treated_subpath{.single_valued_object_field}*

simple_subpath ::=
    general_identification_variable |
    general_identification_variable{.single_valued_object_field}*

treated_subpath ::= TREAT(general_subpath AS subtype)

single_valued_path_expression ::=
    qualified_identification_variable |
    TREAT(qualified_identification_variable AS subtype) |
    state_field_path_expression |
    single_valued_object_path_expression

state_field_path_expression ::= general_subpath.state_field

state_valued_path_expression ::= state_field_path_expression |
general_identification_variable

single_valued_object_path_expression ::= general_subpath.single_valued_object_field

collection_valued_path_expression ::= general_subpath.collection_valued_field
```

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

A *state _field* is designated by the name of an entity or embeddable class state field that corresponds to a basic type.

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

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 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 JOIN o.lineItems l
```

It is illegal to use a *collection_valued_path_expression* other than in the FROM clause of a query except

in an *empty_collection_comparison_expression*, in a *collection_member_expression*, or as an argument to the SIZE operator. See Section 4.6.12, Section 4.6.13, and Section 4.6.17.2.2.

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 relationships.

The syntax for explicit join operations is as follows:

```
join ::= join_spec join_association_path_expression [AS] identification_variable
[join_condition]

fetch_join ::= join_spec FETCH join_association_path_expression

join_association_path_expression ::=
    join_collection_valued_path_expression |
    join_single_valued_path_expression |
    TREAT(join_collection_valued_path_expression _AS_ subtype) |
    TREAT(join_single_valued_path_expression AS subtype)

join_collection_valued_path_expression ::=

identification_variable.{single_valued_embeddable_object_field.}*collection_valued_field

join_single_valued_path_expression ::=

identification_variable.{single_valued_embeddable_object_field.}*single_valued_object_fie
ld

join_spec ::= [LEFT [OUTER] | INNER] JOIN

join_condition ::= ON conditional_expression
```

The inner and outer join operation types described in Section 4.4.5.1, Section 4.4.5.2, and Section 4.4.5.3 are supported.

4.4.5.1. Inner Joins (Relationship Joins)

The syntax for the inner join operation is

```
[INNER] JOIN join_association_path_expression [AS] identification_variable
[join_condition]
```

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 [4]. 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
```

The query below joins over *Employee*, *ContactInfo* and *Phone*. *ContactInfo* is an embeddable class that consists of an address and set of phones. *Phone* is an entity.

```
SELECT p.vendor
FROM Employee e JOIN e.contactInfo c JOIN c.phones p
WHERE c.address.zipcode = '95054'
```

A join condition may be specified for an inner join. This is equivalent to specification of the same condition in the WHERE clause.

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 join_association_path_expression [AS] identification_variable
[join_condition]
```

An outer join without a specified join condition has an implicit join condition over the foreign key relationship corresponding to the join_association_path_expression. It would typically be mapped to a SQL outer join with an ON condition on the foreign key relationship as in the queries below:

Java Persistence query language:

```
SELECT s.name, COUNT(p)
FROM Suppliers s LEFT JOIN s.products p
GROUP BY s.name
```

SQL:

```
SELECT s.name, COUNT(p.id)
FROM Suppliers s LEFT JOIN Products p
    ON s.id = p.supplierId
GROUP By s.name
```

An outer join with an explicit ON condition would cause an additional specified join condition to be added to the generated SQL:

Java Persistence query language:

```
SELECT s.name, COUNT(p)
FROM Suppliers s LEFT JOIN s.products p
    ON p.status = 'inStock'
GROUP BY s.name
```

SQL:

```
SELECT s.name, COUNT(p.id)
FROM Suppliers s LEFT JOIN Products p
    ON s.id = p.supplierId AND p.status = 'inStock'
GROUP BY s.name
```

Note that the result of this query will be different from that of the following query:

```
SELECT s.name, COUNT(p)
FROM Suppliers s LEFT JOIN s.products p
WHERE p.status = 'inStock'
GROUP BY s.name
```

The result of the latter query will exclude suppliers who have no products in stock whereas the former query will include them.

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 as described below.

4.4.5.3. Fetch Joins

A FETCH JOIN enables the fetching of an association or element collection as a side effect of the execution of a query.

The syntax for a fetch join is

```
fetch_join ::= [LEFT [OUTER] | INNER] JOIN FETCH join_association_path_expression
```

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

The following query returns a set of departments. As a side effect, the associated employees for those departments are also retrieved, even though they are not part of the explicit query result. The initialization of the persistent state or relationship fields or properties of the objects that are retrieved as a result of a fetch join is determined by the metadata for that class—in this example, the *Employee* entity class.

```
SELECT d
FROM Department d LEFT JOIN FETCH d.employees
WHERE d.deptno = 1
```

A fetch join has the same join semantics as the corresponding inner or outer join, except that the related objects specified on the right-hand side of the join operation are not returned in the query result or otherwise referenced in the query. Hence, for example, if department 1 has five employees, the above query returns five references to the department 1 entity.

The FETCH JOIN construct must not be used in the FROM clause of a subquery.

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.

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 or embeddable class 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
WHERE l.product.productType = 'office_supplies'
```

can 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. FROM Clause and SQL

The Java Persistence query language 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 JOIN o.lineItems l JOIN l.product p
```

4.4.8. Polymorphism

Java Persistence queries are automatically polymorphic. The FROM clause of a query designates not only instances of the specific entity class(es) to which it explicitly refers but instances of subclasses of those classes as well. The instances returned by a query thus include instances of the subclasses that satisfy the query criteria.

Non-polymorphic queries or queries whose polymorphism is restricted can be specified using entity type expressions in the WHERE clause to restrict the domain of the query. See Section 4.6.17.5.

4.4.9. Downcasting

The use of the TREAT operator is supported for downcasting within path expressions in the FROM and WHERE clauses. Use of the TREAT operator allows access to subclass-specific state.

If during query execution the first argument to the TREAT operator is not a subtype (proper or improper) of the target type, the path is considered to have no value, and does not participate in the determination of the result. That is, in the case of a join, the referenced object does not participate in the result, and in the case of a restriction, the associated predicate is false. Use of the TREAT operator therefore also has the effect of filtering on the specified type (and its subtypes) as well as performing the downcast. If the target type is not a subtype (proper or improper) of the static type of the first argument, the query is invalid.

Examples:

```
SELECT b.name, b.ISBN
FROM Order o JOIN TREAT(o.product AS Book) b

SELECT e FROM Employee e JOIN TREAT(e.projects AS LargeProject) lp
WHERE lp.budget > 1000

SELECT e FROM Employee e JOIN e.projects p
WHERE TREAT(p AS LargeProject).budget > 1000
    OR TREAT(p AS SmallProject).name LIKE 'Persist%'
    OR p.description LIKE "cost overrun"

SELECT e FROM Employee e
WHERE TREAT(e AS Exempt).vacationDays > 10
    OR TREAT(e AS Contractor).hours > 100
```

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 language constructs that can be used in a conditional expression of the WHERE clause, the HAVING clause, or in an ON condition.

State fields that are mapped in serialized form or as lobs cannot be portably used in conditional [61: 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'. String literals in queries, like Java *String* literals, use unicode character encoding. The use of Java escape notation is not supported in query string literals.

Exact numeric literals support the use of Java integer literal syntax as well as SQL exact numeric literal syntax.

Approximate literals support the use Java floating point literal syntax as well as SQL approximate numeric literal syntax.

Appropriate suffixes can be used to indicate the specific type of a numeric literal in accordance with the Java Language Specification. Support for the use of hexadecimal and octal numeric literals is not

required by this specification.

Enum literals support the use of Java enum literal syntax. The fully qualified enum class name must be specified.

The JDBC escape syntax may be used for the specification of date, time, and timestamp literals. For example:

```
SELECT o
FROM Customer c JOIN c.orders o
WHERE c.name = 'Smith'
    AND o.submissionDate < {d '2008-12-31'}
```

The portability of this syntax for date, time, and timestamp literals is dependent upon the JDBC driver in use. Persistence providers are not required to translate from this syntax into the native syntax of the database or driver.

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

Entity type literals are specified by entity names—for example: *Customer*.

Although 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 a 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,* in a *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 must not be mixed in a single query.

Input parameters can only be used in the WHERE clause or HAVING clause of a query or as the new value for an update item in the SET clause of an update statement.

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 4.11.

All input parameters must be single-valued, except in IN expressions (see Section 4.6.9), which support the use of collection-valued input parameters.

The API for the binding of query parameters is described in Chapter 3.

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.
- The same parameter can be used more than once in the query string.
- The ordering of the use of parameters within the query string need not conform to the order of the positional parameters.

4.6.4.2. Named Parameters

A named parameter is denoted by an identifier that is prefixed by the ":" symbol. It follows the rules for identifiers defined in [a4760]. Named parameters are case sensitive.

Example:

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

The same named parameter can be used more than once in the query string.

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, boolean literals, and boolean input parameters.

The scalar expressions described in Section 4.6.17 can be used in conditional expressions.

Aggregate functions can only be used in conditional expressions in a HAVING clause. See Section 4.7.

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 |
    in_expression |
    like_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]*, *[NOT] EXISTS*
- Logical operators:
  - *NOT*
  - *AND*
  - *OR*

The following sections describe operators used in specific expressions.

4.6.7. Comparison Expressions

The syntax for the use of comparison expressions in a conditional expression is as follows [62: Note that queries that contain subqueries on both sides of a comparison operation will not be portable across all databases.]:

```
comparison_expression ::=
    string_expression comparison_operator {string_expression | all_or_any_expression} |
    boolean_expression {= | <>} {boolean_expression | all_or_any_expression} |
    enum_expression {= | <>} {enum_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} |
    entity_type_expression {= | <>} entity_type_expression}

comparison_operator ::= = | > | >= | < | <= | <>
```

Examples:

```
item.cost * 1.08 <= 100.00
CONCAT(person.lastName, ', ', person.firstName)) = 'Jones, Sam'
TYPE(e) = ExemptEmployee
```

4.6.8. Between Expressions

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

```
 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
```

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 4.11.

Examples:

```
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
```

In the following example, *transactionHistory* is a list of credit card transactions defined using an order column.

```
SELECT t
FROM CreditCard c JOIN c.transactionHistory t
WHERE c.holder.name = 'John Doe' AND INDEX(t) BETWEEN 0 AND 9
```

4.6.9. In Expressions

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

```
in_expression ::=
    {state_valued_path_expression | type_discriminator} [NOT] IN
        {(in_item {, in_item}*) | (subquery) | collection_valued_input_parameter}
in_item ::= literal | single_valued_input_parameter
```

The state_valued_path_expression must have a string, numeric, date, time, timestamp, or enum value.

The literal and/or input parameter values must be *like* the same abstract schema type of the *state_valued_path_expression* in type. (See Section 4.12).

The results of the subquery must be *like* the same abstract schema type of the state_valued_path_expression in type. Subqueries are discussed in Section 4.6.16.

Example 1:

```
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')
```

Example 2:

```
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_valued_path_expression or *in_item* in an IN or NOT IN expression is *NULL* or unknown, the value of the expression is unknown.

Note that use of a collection-valued input parameter will mean that a static query cannot be precompiled.

4.6.10. Like Expressions

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

```
like_expression ::=
    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* _. [63: Refer to [2] for a more precise characterization of these rules.]

Examples:

- *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.

4.6.11. Null Comparison Expressions

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

```
null_comparison_expression ::=
    {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.

Null comparisons over instances of embeddable class types are not supported. Support for comparisons over embeddables may be added in a future release of this specification.

4.6.12. Empty Collection Comparison Expressions

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

```
empty_collection_comparison_expression ::=
    collection_valued_path_expression IS [NOT] EMPTY
```

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.

4.6.13. Collection Member Expressions

The syntax for the use of the comparison operator MEMBER OF [64: The use of the reserved word OF is optional in this expression.] in an *collection_member_expression* is as follows:

```
collection_member_expression ::=
    entity_or_value_expression [NOT] MEMBER [OF] collection_valued_path_expression
entity_or_value_expression ::=
    single_valued_object_path_expression |
    state_valued_path_expression |
    simple_entity_or_value_expression
simple_entity_or_value_expression ::=
    identification_variable |
    input_parameter |
    literal
```

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

Expressions that evaluate to embeddable types are not supported in collection member expressions. Support for use of embeddables in collection member expressions may be added in a future release of this specification.

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 *entity_or_value_expression* in the collection member expression is *NULL* or unknown, the value of the collection member expression is unknown.

Example:

```
SELECT p
FROM Person p
WHERE 'Joe' MEMBER OF p.nicknames
```

4.6.14. 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)
```

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.

4.6.15. All or Any Expressions

An ALL conditional expression is a predicate over a subquery that is true if the comparison operation is true for all values in the result of the subquery or the result of the subquery is empty. An ALL conditional expression is false if the result of the comparison is false for at least one value of the result of the subquery, and is unknown if neither true nor false.

An ANY conditional expression is a predicate over a subquery that is true if the comparison operation is true for some value in the result of the subquery. An ANY conditional expression is false if the result of the subquery is empty or if the comparison operation is false for every value in the result of the subquery, and is unknown if neither true nor false. The keyword SOME is synonymous with ANY.

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 4.12.

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)
```

The result of this query consists of all employees whose salaries exceed the salaries of all managers in their department.

4.6.16. Subqueries

Subqueries may be used in the WHERE or HAVING clause. [65: 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.]

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
subquery_from_clause ::=
    FROM subselect_identification_variable_declaration
        {, subselect_identification_variable_declaration |
            collection_member_declaration}*
subselect_identification_variable_declaration ::=
    identification_variable_declaration |
    derived_path_expression [AS] identification_variable {join}* |
    derived_collection_member_declaration
simple_select_expression ::=
    single_valued_path_expression |
    scalar_expression |
    aggregate_expression |
    identification_variable
derived_path_expression ::=
    general_derived_path.single_valued_object_field |
    general_derived_path.collection_valued_field
general_derived_path ::=
    simple_derived_path |
    treated_derived_path{.single_valued_object_field}*
simple_derived_path ::= superquery_identification_variable{.single_valued_object_field}*
treated_derived_path ::= TREAT(general_derived_path AS subtype)
derived_collection_member_declaration ::=
    IN
superquery_identification_variable.{single_valued_object_field.}*collection_valued_field
```

Examples:

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

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 examples using numeric comparisons.

```
SELECT c
FROM Customer c
WHERE (SELECT AVG(o.price) FROM c.orders o) > 100

SELECT goodCustomer
FROM Customer goodCustomer
WHERE goodCustomer.balanceOwed < (
    SELECT AVG(c.balanceOwed)/2.0 FROM Customer c)
```

4.6.17. Scalar Expressions

Numeric, string, datetime, case, and entity type expressions result in scalar values.

Scalar expressions may be used in the SELECT clause of a query as well as in the WHERE [66: Note that expressions involving aggregate operators must not be used in the WHERE clause.] and HAVING clauses.

```
scalar_expression::=
    arithmetic_expression |
    string_expression |
    enum_expression |
    datetime_expression |
    boolean_expression |
    case_expression |
    entity_type_expression
```

4.6.17.1. Arithmetic Expressions

The arithmetic operators are:

- +, - unary
- *, / multiplication and division
- +, - addition and subtraction

Arithmetic operations use numeric promotion.

Arithmetic functions are described in Section 4.6.17.2.2.

4.6.17.2. Built-in String, Arithmetic, and Datetime Functional Expressions

The Java Persistence query language includes the built-in functions described in Section 4.6.17.2.1, Section 4.6.17.2.2, Section 4.6.17.2.3, which may be used in the SELECT, WHERE or HAVING clause of a query. The invocation of predefined database functions and user-defined database functions is described in Section 4.6.17.3.

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

String Functions

```
functions_returning_strings ::=
    CONCAT(string_expression, string_expression {, string_expression}*) |
    SUBSTRING(string_expression,
              arithmetic_expression [, arithmetic_expression]) |
    TRIM([[trim_specification] [trim_character] FROM] string_expression) |
    LOWER(string_expression) |
    UPPER(string_expression)
trim_specification ::= LEADING | TRAILING | BOTH

functions_returning_numerics ::=
    LENGTH(string_expression) |
    LOCATE(string_expression, string_expression[, 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 third argument is optional. If it is not specified, the substring from the start position to the end of the string is returned. 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 will be 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*) [67: 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.]. If a trim specification is not provided, it defaults to BOTH. The TRIM function returns the trimmed string.

The LOWER and UPPER functions convert a string to lower and upper case, respectively, with regard to the locale of the database. 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. [68: 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 LENGTH function returns the length of the string in characters as an integer.

Arithmetic Functions

```
functions_returning_numerics ::=
    ABS(arithmetic_expression) |
    SQRT(arithmetic_expression) |
    MOD(arithmetic_expression, arithmetic_expression) |
    SIZE(collection_valued_path_expression) |
    INDEX(identification_variable)
```

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.

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

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.

The INDEX function returns an integer value corresponding to the position of its argument in an ordered list. The INDEX function can only be applied to identification variables denoting types for which an order column has been specified.

In the following example, *studentWaitlist* is a list of students for which an order column has been specified:

```
SELECT w.name
FROM Course c JOIN c.studentWaitlist w
WHERE c.name = 'Calculus'
AND INDEX(w) = 0
```

Datetime Functions

```
functions_returning_datetime :=
    CURRENT_DATE |
    CURRENT_TIME |
    CURRENT_TIMESTAMP
```

The datetime functions return the value of current date, time, and timestamp on the database server.

4.6.17.3. Invocation of Predefined and User-defined Database Functions

The invocation of functions other than the built-in functions of the Java Persistence query language is supported by means of the *function_invocation* syntax. This includes the invocation of predefined database functions and user-defined database functions.

```
function_invocation ::= FUNCTION(function_name {, function_arg}*)

function_arg ::=
    literal |
    state_valued_path_expression |
    input_parameter |
    scalar_expression
```

The *function_name* argument is a string that denotes the database function that is to be invoked. The arguments must be suitable for the database function that is to be invoked. The result of the function must be suitable for the invocation context.

The function may be a database-defined function or a user-defined function. The function may be a scalar function or an aggregate function.

Applications that use the *function_invocation* syntax will not be portable across databases.

Example:

```
SELECT c
FROM Customer c
WHERE FUNCTION('hasGoodCredit', c.balance, c.creditLimit)
```

4.6.17.4. Case Expressions

The following forms of case expressions are supported: general case expressions, simple case expressions, coalesce expressions, and nullif expressions. [69: Note that not all databases support the use of SQL case expressions. The use of case expressions may result in queries that are not portable to such databases.]

```
case_expression ::=
    general_case_expression |
    simple_case_expression |
    coalesce_expression |
    nullif_expression

general_case_expression ::=
    CASE when_clause {when_clause}* ELSE scalar_expression END
when_clause ::= WHEN conditional_expression THEN scalar_expression

simple_case_expression ::=
    CASE case_operand simple_when_clause {simple_when_clause}*
    ELSE scalar_expression
    END
case_operand ::= state_valued_path_expression | type_discriminator
simple_when_clause ::= WHEN scalar_expression THEN scalar_expression

coalesce_expression ::= COALESCE(scalar_expression {, scalar_expression}+)

nullif_expression ::= NULLIF(scalar_expression, scalar_expression)
```

Examples:

```
UPDATE Employee e
SET e.salary =
    CASE WHEN e.rating = 1 THEN e.salary * 1.1
         WHEN e.rating = 2 THEN e.salary * 1.05
         ELSE e.salary * 1.01
    END

UPDATE Employee e
SET e.salary =
    CASE e.rating WHEN 1 THEN e.salary * 1.1
                  WHEN 2 THEN e.salary * 1.05
                  ELSE e.salary * 1.01
    END

SELECT e.name,
    CASE TYPE(e) WHEN Exempt THEN 'Exempt'
                 WHEN Contractor THEN 'Contractor'
                 WHEN Intern THEN 'Intern'
                 ELSE 'NonExempt'
    END
FROM Employee e
WHERE e.dept.name = 'Engineering'

SELECT e.name,
       f.name,
       CONCAT(CASE WHEN f.annualMiles > 50000 THEN 'Platinum '
                   WHEN f.annualMiles > 25000 THEN 'Gold '
                   ELSE ''
              END,
       'Frequent Flyer')
FROM Employee e JOIN e.frequentFlierPlan f
```

4.6.17.5. Entity Type Expressions

An entity type expression can be used to restrict query polymorphism. The TYPE operator returns the exact type of the argument.

The syntax of an entity type expression is as follows:

```
entity_type_expression ::=
    type_discriminator |
    entity_type_literal |
    input_parameter
type_discriminator ::=
    TYPE(general_identification_variable |
         single_valued_object_path_expression |
         input_parameter)
```

An entity_type_literal is designated by the entity name.

The Java class of the entity is used as an input parameter to specify the entity type.

Examples:

```
SELECT e
FROM Employee e
WHERE TYPE(e) IN (Exempt, Contractor)

SELECT e
FROM Employee e
WHERE TYPE(e) IN (:empType1, :empType2)

SELECT e
FROM Employee e
WHERE TYPE(e) IN :empTypes

SELECT TYPE(e)
FROM Employee e
WHERE TYPE(e) <> Exempt
```

4.7. GROUP BY, HAVING

The GROUP BY construct enables the aggregation of result 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 ::= single_valued_path_expression | identification_variable

having_clause ::= HAVING conditional_expression
```

If a 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.

The requirements for the SELECT clause when GROUP BY is used follow those of SQL: namely, any item that appears in the SELECT clause (other than as an aggregate function or 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.

Grouping by an entity is permitted. In this case, the entity must contain no serialized state fields or lob-valued state fields that are eagerly fetched. Grouping by an entity that contains serialized state fields or lob-valued state fields is not portable, since the implementation is permitted to eagerly fetch fields or properties that have been specified as LAZY.

Grouping by embeddables is not supported.

The HAVING clause is used to filter over the groups, and can contain aggregate functions over attributes included in the groups and/or functions or other query language operators over the attributes that are used for grouping. It is not required that an aggregate function used in the HAVING clause also be used in the SELECT clause.

If there is no GROUP BY clause and the HAVING clause is used, the result is treated as a single group, and the select list can only consist of aggregate functions. The use of HAVING in the absence of GROUP BY is not required to be supported by an implementation of this specification. Portable applications should not rely on HAVING without the use of GROUP BY.

Examples:

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

SELECT c.country, COUNT(c)
FROM Customer c
GROUP BY c.country
HAVING COUNT(c) > 30

SELECT c, COUNT(o)
FROM Customer c JOIN c.orders o
GROUP BY c
HAVING COUNT(o) >= 5
```

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 can contain one or more of the following elements: an identification variable that ranges over an abstract schema type, a single-valued path expression, a scalar expression, an aggregate expression, a constructor expression.

The SELECT clause has the following syntax:

```
select_clause ::= SELECT [DISTINCT] select_item {, select_item}*
select_item ::= select_expression [[AS] result_variable]
select_expression ::=
    single_valued_path_expression |
    scalar_expression |
    aggregate_expression |
    identification_variable |
    OBJECT(identification_variable) |
    constructor_expression
constructor_expression ::=
    NEW constructor_name (constructor_item {, constructor_item}*)
constructor_item ::=
    single_valued_path_expression |
    scalar_expression |
    aggregate_expression |
    identification_variable
aggregate_expression ::=
    {AVG | MAX | MIN | SUM} ([DISTINCT] state_valued_path_expression) |
     COUNT ([DISTINCT] identification_variable | state_valued_path_expression |
         single_valued_object_path_expression) |
     function_invocation
```

For example:

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

In the following example, *videoInventory* is a Map from the entity *Movie* to the number of copies in stock:

```
SELECT v.location.street, KEY(i).title, VALUE(i)
FROM VideoStore v JOIN v.videoInventory i
WHERE v.location.zipcode = '94301' AND VALUE(i) > 0
```

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.

The result of DISTINCT over embeddable objects or map *entry* results is undefined.

Standalone identification variables in the SELECT clause may optionally be qualified by the OBJECT operator. [70: Note that the keyword OBJECT is not required. It is preferred that it be omitted for new queries.] The SELECT clause must not use the OBJECT operator to qualify path expressions.

A *result_variable* may be used to name a *select_item* in the query result. [71: This can be used, for example, to refer to a select expression in the ORDER BY clause.]

Example:

```
SELECT c, COUNT(l) AS itemCount
FROM Customer c JOIN c.Orders o JOIN o.lineItems l
WHERE c.address.state = 'CA'
GROUP BY c
ORDER BY itemCount
```

4.8.1. Result Type of the SELECT Clause

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 a scalar expression, the result of an aggregate function, the result of a construction operation, or some sequence of these.

The result type of the SELECT clause is defined by the the result types of the select expressions contained in it. When multiple select expressions are used in the SELECT clause, the elements in this result correspond in order to the order of their specification in the SELECT clause and in type to the result types of each of the select expressions.

The type of the result of a select_expression is as follows:

- The result type of an identification_variable is the type of the entity object or embeddable object to which the identification variable corresponds. The type of an identification_variable that refers to an entity abstract schema type is the type of the entity to which that identification variable corresponds or a subtype as determined by the object/relational mapping.
- The result type of a single_valued_path_expression that is a state_field_path_expression is the same type as the corresponding state field of the entity or embeddable class. If the state field of the entity is a primitive type, the result type is the corresponding object type.
- The result type of a single_valued_path_expression that is a single_valued_object_path_expression is the type of the entity object or embeddable object to which the path expression corresponds. A single_valued_object_path_expression that results in an entity object will result in an entity of the type of the relationship field or the subtype of the relationship field of the entity object as determined by the object/relational mapping.
- The result type of a single_valued_path_expression that is an *identification_variable* to which the KEY or VALUE function has been applied is determined by the type of the map key or value respectively, as defined by the above rules.
- The result type of a single_valued_path_expression that is an *identification_variable* to which the ENTRY function has been applied is *java.util.Map.Entry*, where the key and value types of the map entry are determined by the above rules as applied to the map key and map value respectively.
- The result type of a *scalar_expression* is the type of the scalar value to which the expression evaluates. The result type of a numeric *scalar_expression* is defined in Section 4.8.6.
- The result type of an *entity_type_expression* scalar expression is the Java class to which the resulting abstract schema type corresponds.
- The result type of aggregate_expression is defined in Section 4.8.5.
- The result type of a constructor_expression is the type of the class for which the constructor is defined. The types of the arguments to the constructor are defined by the above rules.

4.8.2. Constructor Expressions in the SELECT Clause

A constructor may be used in the SELECT list to return an instance of a Java class. 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 as the constructor name in the SELECT NEW clause, the resulting entity instances will be in either the new or the detached state, depending on whether a primary key is retrieved for the constructed object.

If a *single_valued_path_expression* or *identification_variable* that is an argument to the constructor references an entity, the resulting entity instance referenced by that *single_valued_path_expression* or *identification_variable* will be in the managed state.

For example,

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

4.8.3. Null Values in the Query Result

If the result of a query corresponds to an 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.

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

4.8.4. Embeddables in the Query Result

If the result of a query corresponds to an identification variable or state field whose value is an embeddable, the embeddable instance returned by the query will not be in the managed state (i.e., it will not be part of the state of any managed entity).

In the following example, the *Address* instances returned by the query will reference *Phone* instances. While the *Phone* instances will be managed, the *Address* instances referenced by the *addr* result variable will not be. Modifications to these embeddable instances will have no effect on persistent state.

```
@Entity
public class Employee {
    @Id
    int id;

    Address address;

    // ...
}

@Embeddable
public class Address {
    String street;

    // ...

    @OneToOne
    Phone phone; // fetch=EAGER
}

@Entity
public class Phone {
    @Id
    int id;

    // ...

    @OneToOne(mappedBy="address.phone")
    Employee emp; // fetch=EAGER
}
```

```
SELECT e.address AS addr
FROM Employee e
```

4.8.5. Aggregate Functions in the SELECT Clause

The result of a 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 a query: AVG, COUNT, MAX, MIN, SUM, aggregate functions defined in the database.

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:

- 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.

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

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

The use of DISTINCT with COUNT is not supported for arguments of embeddable types or map entry types.

The invocation of aggregate database functions, including user defined functions, is supported by means of the FUNCTION operator. See Section 4.6.17.3.

4.8.5.1. Examples

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'
```

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
```

4.8.6. Numeric Expressions in the SELECT Clause

The type of a numeric expression in the query result is determined as follows:

- An operand that corresponds to a persistent state field is of the same type as that persistent state field.
- An operand that corresponds to one of arithmetic functions described in Section 4.6.17.2.2 is of the type defined by Section 4.6.17.2.2.
- An operand that corresponds to one of an aggregate functions described in Section 4.8.5 is of the type defined by Section 4.8.5.

The result of a case expression, coalesce expression, nullif expression, or arithmetic expression (+, -, *, /) is determined by applying the following rule to its operands [73: In the case of a general or simple case expression, these are the scalar expressions of the THEN and ELSE clauses.].

- If there is an operand of type Double or double, the result of the operation is of type Double;
- otherwise, if there is an operand of type Float or float, the result of the operation is of type Float;
- otherwise, if there is an operand of type BigDecimal, the result of the operation is of type BigDecimal;
- otherwise, if there is an operand of type BigInteger, the result of the operation is of type BigInteger, unless the operator is / (division), in which case the numeric result type is not further defined;
- otherwise, if there is an operand of type Long or long, the result of the operation is of type Long, unless the operator is / (division), in which case the numeric result type is not further defined;

- otherwise, if there is an operand of integral type, the result of the operation is of type Integer, unless the operator is / (division), in which case the numeric result type is not further defined.

Users should note that the semantics of the SQL division operation are not standard across databases. In particular, when both operands are of integral types, the result of the division operation will be an integral type in some databases, and an non-integral type in others. Portable applications should not assume a particular result type.

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 | general_identification_variable | result_variable}
        [ASC | DESC]
```

An orderby_item must be one of the following:

1. A *state_field_path_expression* that evaluates to an orderable state field of an entity or embeddable class abstract schema type designated in the SELECT clause by one of the following:
   - a *general_identification_variable*
   - a *single_valued_object_path_expression*
2. A *state_field_path_expression* that evaluates to the same state field of the same entity or embeddable abstract schema type as a *state_field_path_expression* in the SELECT clause
3. A general_identification_variable that evaluates to the same map field of the same entity or embeddable abstract schema type as a general_identification_variable in the SELECT clause
4. A *result_variable* that refers to an orderable item in the SELECT clause for which the same *result_variable* has been specified. This may be the result of an *aggregate_expression* , a *scalar_expression*, or a *state_field_path_expression* in the SELECT clause.

For example, the four queries below are legal.

```
SELECT o
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA'
ORDER BY o.quantity DESC, 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

SELECT o.quantity, o.cost*1.08 AS taxedCost, a.zipcode
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA' AND a.county = 'Santa Clara'
ORDER BY o.quantity, taxedCost, a.zipcode

SELECT AVG(o.quantity) as q, a.zipcode
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA'
GROUP BY a.zipcode
ORDER BY q DESC
```

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 for the associated *orderby_item*; 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 execution method if the ORDER

BY clause is used.

4.10. 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.

The syntax of these operations is as follows:

```
update_statement ::= update_clause [where_clause]
update_clause ::= UPDATE entity_name [[AS] identification_variable]
                  SET update_item {, update_item}*
update_item ::= [identification_variable.]{single_valued_embeddable_object_field.}*
    {state_field | single_valued_object_field} = new_value
new_value ::=
    scalar_expression |
    simple_entity_expression |
    NULL

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

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

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 field to which it is assigned.

Bulk update maps directly to a database update operation, bypassing optimistic locking checks. Portable applications must manually update the value of the version column, if desired, and/or manually validate the value of the version column.

The persistence context is not synchronized with the result of the bulk update or delete.

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 transaction in a new persistence context or before fetching or accessing entities 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

UPDATE Employee e
SET e.address.building = 22
WHERE e.address.building = 14
    AND e.address.city = 'Santa Clara'
    AND e.project = 'Java EE'
```

4.11. Null Values

When the target of a reference does not exist in the database, its value is regarded as *NULL*. SQL *NULL* semantics [2] 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 object 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

The Java Persistence query language 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 query comparisons involving the empty string and NULL value.

4.12. Equality and Comparison Semantics

Only the values of *like* types are permitted 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 wrapped 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 the arithmetic operators and comparison operators are permitted to be applied to state fields and input parameters of the wrapped 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.

Only equality/inequality comparisons over enums are required to be supported.

Comparisons over instances of embeddable class or map entry types are not supported.

4.13. Examples

The following examples illustrate the syntax and semantics of the Java Persistence query language. These examples are based on the example presented in Section 4.3.2.

4.13.1. 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
```

4.13.2. Queries with Relationships

Find all orders that have line items:

```
SELECT DISTINCT o
FROM Order o JOIN 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.

```
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* type 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 4.12. 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.

4.13.3. 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 JOIN 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.

4.14. BNF

BNF notation summary:

- { ... } grouping

- [...] optional constructs
- * zero or more
- + one or more
- | alternates

The following is the BNF for the Java Persistence query language.

```
QL_statement ::= 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 ::= entity_name [AS] identification_variable
join ::= join_spec join_association_path_expression [AS] identification_variable
    [join_condition]
fetch_join ::= join_spec FETCH join_association_path_expression
join_spec ::= [LEFT [OUTER] | INNER] JOIN
join_condition ::= ON conditional_expression
join_association_path_expression ::=
    join_collection_valued_path_expression |
    join_single_valued_path_expression |
    TREAT(join_collection_valued_path_expression AS subtype) |
    TREAT(join_single_valued_path_expression AS subtype)
join_collection_valued_path_expression ::=
    identification_variable.{single_valued_embeddable_object_field.}*
collection_valued_field
join_single_valued_path_expression ::=
    identification_variable.{single_valued_embeddable_object_field.}*
single_valued_object_field
collection_member_declaration ::=
    IN (collection_valued_path_expression) [AS] identification_variable
qualified_identification_variable ::=
    map_field_identification_variable |
    ENTRY(identification_variable)
map_field_identification_variable ::=
    KEY(identification_variable) |
    VALUE(identification_variable)
single_valued_path_expression ::=
    qualified_identification_variable |
    TREAT(qualified_identification_variable AS subtype) |
    state_field_path_expression |
    single_valued_object_path_expression
```

```
general_identification_variable ::=
    identification_variable |
    map_field_identification_variable
general_subpath ::= simple_subpath | treated_subpath{.single_valued_object_field}*
simple_subpath ::=
    general_identification_variable |
    general_identification_variable{.single_valued_object_field}*
treated_subpath ::= TREAT(general_subpath AS subtype)
state_field_path_expression ::= general_subpath.state_field
state_valued_path_expression ::=
    state_field_path_expression | general_identification_variable
single_valued_object_path_expression ::=
    general_subpath.single_valued_object_field
collection_valued_path_expression ::= general_subpath.{collection_valued_field}
update_clause ::= UPDATE entity_name [[AS] identification_variable]
    SET update_item {, update_item}*
update_item ::= [identification_variable.]{single_valued_embeddable_object_field.}*
    {state_field | single_valued_object_field} = new_value
new_value ::=
    scalar_expression |
    simple_entity_expression |
    NULL
delete_clause ::= DELETE FROM entity_name [[AS] identification_variable]
select_clause ::= SELECT [DISTINCT] select_item {, select_item}*
select_item ::= select_expression [[AS] result_variable]
select_expression ::=
    single_valued_path_expression |
    scalar_expression |
    aggregate_expression |
    identification_variable |
    OBJECT(identification_variable) |
    constructor_expression
constructor_expression ::=
    NEW constructor_name (constructor_item {, constructor_item}*)
constructor_item ::=
    single_valued_path_expression |
    scalar_expression |
    aggregate_expression |
    identification_variable
aggregate_expression ::=
    {AVG | MAX | MIN | SUM} ([DISTINCT] state_valued_path_expression) |
    COUNT ([DISTINCT] identification_variable | state_valued_path_expression |
        single_valued_object_path_expression) |
    function_invocation
where_clause ::= WHERE conditional_expression
groupby_clause ::= GROUP BY groupby_item {, groupby_item}*
groupby_item ::= single_valued_path_expression | identification_variable
having_clause ::= HAVING conditional_expression
```

```
orderby_clause ::= ORDER BY orderby_item {, orderby_item}*
orderby_item ::=
    state_field_path_expression |
    general_identification_variable |
    result_variable
    [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 |
collection_member_declaration}*
subselect_identification_variable_declaration ::=
    identification_variable_declaration |
    derived_path_expression [AS] identification_variable {join}* |
    derived_collection_member_declaration
derived_path_expression ::=
    general_derived_path.single_valued_object_field |
    general_derived_path.collection_valued_field
general_derived_path ::=
    simple_derived_path |
    treated_derived_path{.single_valued_object_field}*
simple_derived_path ::= superquery_identification_variable{.single_valued_object_field}*
treated_derived_path ::= TREAT(general_derived_path AS subtype)
derived_collection_member_declaration ::=
    IN
superquery_identification_variable.{single_valued_object_field.}*collection_valued_field
simple_select_clause ::= SELECT [DISTINCT] simple_select_expression
simple_select_expression::=
    single_valued_path_expression |
    scalar_expression |
    aggregate_expression |
    identification_variable
scalar_expression ::=
    arithmetic_expression |
    string_expression |
    enum_expression |
    datetime_expression |
    boolean_expression |
    case_expression |
    entity_type_expression
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 |
```

```
    in_expression |
    like_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_valued_path_expression | type_discriminator} [NOT] IN
        {(in_item{, in_item}*) | (subquery) | collection_valued_input_parameter}
in_item ::= literal | single_valued_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_or_value_expression
    [NOT] MEMBER [OF] collection_valued_path_expression
entity_or_value_expression ::=
    single_valued_object_path_expression |
    state_field_path_expression |
    simple_entity_or_value_expression
simple_entity_or_value_expression ::=
    identification_variable |
    input_parameter |
    literal
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} |
    enum_expression {= | <>} {enum_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} |
    entity_type_expression {= | <>} entity_type_expression}
comparison_operator ::= = | > | >= | < | <= | <>
arithmetic_expression ::=
    arithmetic_term | arithmetic_expression {+ | -} arithmetic_term
arithmetic_term ::= arithmetic_factor | arithmetic_term {* | /} arithmetic_factor
arithmetic_factor ::= [{+ | -}] arithmetic_primary
```

```
arithmetic_primary ::=
    state_valued_path_expression |
    numeric_literal |
    (arithmetic_expression) |
    input_parameter |
    functions_returning_numerics |
    aggregate_expression |
    case_expression |
    function_invocation |
    (subquery)
string_expression ::=
    state_valued_path_expression |
    string_literal |
    input_parameter |
    functions_returning_strings |
    aggregate_expression |
    case_expression |
    function_invocation |
    (subquery)
datetime_expression ::=
    state_valued_path_expression |
    input_parameter |
    functions_returning_datetime |
    aggregate_expression |
    case_expression |
    function_invocation |
    date_time_timestamp_literal |
    (subquery)
boolean_expression ::=
    state_valued_path_expression |
    boolean_literal |
    input_parameter |
    case_expression |
    function_invocation |
    (subquery)
enum_expression ::=
    state_valued_path_expression |
    enum_literal |
    input_parameter |
    case_expression |
    (subquery)
entity_expression ::= single_valued_object_path_expression | simple_entity_expression
simple_entity_expression ::= identification_variable | input_parameter
entity_type_expression ::=
    type_discriminator |
    entity_type_literal |
    input_parameter
type_discriminator ::=
```

```
    TYPE(general_identification_variable |
        single_valued_object_path_expression |
        input_parameter)
functions_returning_numerics ::=
    LENGTH(string_expression) |
    LOCATE(string_expression, string_expression[, arithmetic_expression]) |
    ABS(arithmetic_expression) |
    SQRT(arithmetic_expression) |
    MOD(arithmetic_expression, arithmetic_expression) |
    SIZE(collection_valued_path_expression) |
    INDEX(identification_variable)
functions_returning_datetime ::=
    CURRENT_DATE |
    CURRENT_TIME |
    CURRENT_TIMESTAMP
functions_returning_strings ::=
    CONCAT(string_expression, string_expression{, string_expression}*) |
    SUBSTRING(string_expression, arithmetic_expression[, arithmetic_expression]) |
    TRIM([[trim_specification] [trim_character] FROM] string_expression) |
    LOWER(string_expression) |
    UPPER(string_expression)
trim_specification ::= LEADING | TRAILING | BOTH
function_invocation ::= FUNCTION(function_name{, function_arg}*)
function_arg ::=
    literal |
    state_valued_path_expression |
    input_parameter |
    scalar_expression
_case_expression ::=_
    general_case_expression |
    simple_case_expression |
    coalesce_expression |
    nullif_expression
general_case_expression::= CASE when_clause {when_clause}* ELSE scalar_expression END
when_clause ::= WHEN conditional_expression THEN scalar_expression
simple_case_expression ::=
    CASE case_operand simple_when_clause {simple_when_clause}*
    ELSE scalar_expression
    END
case_operand ::= state_valued_path_expression | type_discriminator
simple_when_clause ::= WHEN scalar_expression THEN scalar_expression
coalesce_expression ::= COALESCE(scalar_expression{, scalar_expression}+)
nullif_expression::= NULLIF(scalar_expression, scalar_expression)
```

Chapter 5. Metamodel API

This specification provides a set of interfaces for dynamically accessing the metamodel corresponding to the managed classes of a persistence unit.

5.1. Metamodel API Interfaces

The *jakarta.persistence.metamodel* interfaces provide __ for dynamically accessing the metamodel of the persistent state and relationships of the managed classes of a persistence unit.

The metamodel can be accessed through the *EntityManagerFactory* or *EntityManager getMetamodel* methods.

The metamodel API may be extended to cover object/relational mapping information in a future release of this specification.

5.1.1. Metamodel Interface

```
package jakarta.persistence.metamodel;

import java.util.Set;

/**
 * Provides access to the metamodel of persistent
 * entities in the persistence unit.
 *
 * @since 2.0
 */
public interface Metamodel {

    /**
     *  Return the metamodel entity type representing the entity.
     *  @param cls  the type of the represented entity
     *  @return the metamodel entity type
     *  @throws IllegalArgumentException if not an entity
     */
    <X> EntityType<X> entity(Class<X> cls);

    /**
     *  Return the metamodel managed type representing the
     *  entity, mapped superclass, or embeddable class.
     *  @param cls  the type of the represented managed class
     *  @return the metamodel managed type
     *  @throws IllegalArgumentException if not a managed class
     */
```

```
    <X> ManagedType<X> managedType(Class<X> cls);

    /**
     *  Return the metamodel embeddable type representing the
     *  embeddable class.
     *  @param cls  the type of the represented embeddable class
     *  @return the metamodel embeddable type
     *  @throws IllegalArgumentException if not an embeddable class
     */
    <X> EmbeddableType<X> embeddable(Class<X> cls);

    /**
     *  Return the metamodel managed types.
     *  @return the metamodel managed types
     */
    Set<ManagedType<?>> getManagedTypes();

    /**
     * Return the metamodel entity types.
     * @return the metamodel entity types
     */
    Set<EntityType<?>> getEntities();

    /**
     * Return the metamodel embeddable types.  Returns empty set
     * if there are no embeddable types.
     * @return the metamodel embeddable types
     */
    Set<EmbeddableType<?>> getEmbeddables();
}
```

5.1.2. Type Interface

```
package jakarta.persistence.metamodel;

/**
 * Instances of the type <code>Type</code> represent persistent object
 * or attribute types.
 *
 * @param <X>  The type of the represented object or attribute
 *
 * @since 2.0
 */
public interface Type<X> {

       public static enum PersistenceType {

       /** Entity */
           ENTITY,

       /** Embeddable class */
       EMBEDDABLE,

       /** Mapped superclass */
       MAPPED_SUPERCLASS,

       /** Basic type */
       BASIC
       }

    /**
     *  Return the persistence type.
     *  @return persistence type
     */
    PersistenceType getPersistenceType();

    /**
     *  Return the represented Java type.
     *  @return Java type
     */
    Class<X> getJavaType();
}
```

5.1.3. ManagedType Interface

```
package jakarta.persistence.metamodel;

import java.util.Set;
```

```
/**
 *  Instances of the type <code>ManagedType</code> represent entity, mapped
 *  superclass, and embeddable types.
 *
 *  @param <X> The represented type.
 *
 *  @since 2.0
 *
 */
public interface ManagedType<X> extends Type<X> {

    /**
     *  Return the attributes of the managed type.
     *  @return attributes of the managed type
     */
     Set<Attribute<? super X, ?>> getAttributes();

    /**
     *  Return the attributes declared by the managed type.
     *  Returns empty set if the managed type has no declared
     *  attributes.
     *  @return declared attributes of the managed type
     */
     Set<Attribute<X, ?>> getDeclaredAttributes();

    /**
     *  Return the single-valued attribute of the managed
     *  type that corresponds to the specified name and Java type.
     *  @param name  the name of the represented attribute
     *  @param type  the type of the represented attribute
     *  @return single-valued attribute with given name and type
     *  @throws IllegalArgumentException if attribute of the given
     *          name and type is not present in the managed type
     */
    <Y> SingularAttribute<? super X, Y> getSingularAttribute(String name, Class<Y> type);

    /**
     *  Return the single-valued attribute declared by the
     *  managed type that corresponds to the specified name and
     *  Java type.
     *  @param name  the name of the represented attribute
     *  @param type  the type of the represented attribute
     *  @return declared single-valued attribute of the given
     *          name and type
     *  @throws IllegalArgumentException if attribute of the given
     *          name and type is not declared in the managed type
     */
```

```
    <Y> SingularAttribute<X, Y> getDeclaredSingularAttribute(String name, Class<Y> type);

    /**
     *  Return the single-valued attributes of the managed type.
     *  Returns empty set if the managed type has no single-valued
     *  attributes.
     *  @return single-valued attributes
     */
    Set<SingularAttribute<? super X, ?>> getSingularAttributes();

    /**
     *  Return the single-valued attributes declared by the managed
     *  type.
     *  Returns empty set if the managed type has no declared
     *  single-valued attributes.
     *  @return declared single-valued attributes
     */
    Set<SingularAttribute<X, ?>> getDeclaredSingularAttributes();

    /**
     *  Return the Collection-valued attribute of the managed type
     *  that corresponds to the specified name and Java element type.
     *  @param name  the name of the represented attribute
     *  @param elementType  the element type of the represented
     *                      attribute
     *  @return CollectionAttribute of the given name and element
     *          type
     *  @throws IllegalArgumentException if attribute of the given
     *          name and type is not present in the managed type
     */
    <E> CollectionAttribute<? super X, E> getCollection(String name, Class<E>
elementType);

    /**
     *  Return the Collection-valued attribute declared by the
     *  managed type that corresponds to the specified name and Java
     *  element type.
     *  @param name  the name of the represented attribute
     *  @param elementType  the element type of the represented
     *                      attribute
     *  @return declared <code>CollectionAttribute</code> of the given name and
     *          element type
     *  @throws IllegalArgumentException if attribute of the given
     *          name and type is not declared in the managed type
     */
    <E> CollectionAttribute<X, E> getDeclaredCollection(String name, Class<E>
elementType);
```

```
    /**
     *  Return the Set-valued attribute of the managed type that
     *  corresponds to the specified name and Java element type.
     *  @param name  the name of the represented attribute
     *  @param elementType  the element type of the represented
     *                      attribute
     *  @return SetAttribute of the given name and element type
     *  @throws IllegalArgumentException if attribute of the given
     *          name and type is not present in the managed type
     */
    <E> SetAttribute<? super X, E> getSet(String name, Class<E> elementType);

    /**
     *  Return the Set-valued attribute declared by the managed type
     *  that corresponds to the specified name and Java element type.
     *  @param name  the name of the represented attribute
     *  @param elementType  the element type of the represented
     *                      attribute
     *  @return declared SetAttribute of the given name and
     *          element type
     *  @throws IllegalArgumentException if attribute of the given
     *          name and type is not declared in the managed type
     */
    <E> SetAttribute<X, E> getDeclaredSet(String name, Class<E> elementType);

    /**
     *  Return the List-valued attribute of the managed type that
     *  corresponds to the specified name and Java element type.
     *  @param name  the name of the represented attribute
     *  @param elementType  the element type of the represented
     *                      attribute
     *  @return ListAttribute of the given name and element type
     *  @throws IllegalArgumentException if attribute of the given
     *          name and type is not present in the managed type
     */
    <E> ListAttribute<? super X, E> getList(String name, Class<E> elementType);

    /**
     *  Return the List-valued attribute declared by the managed
     *  type that corresponds to the specified name and Java
     *  element type.
     *  @param name  the name of the represented attribute
     *  @param elementType  the element type of the represented
     *                      attribute
     *  @return declared ListAttribute of the given name and
     *          element type
     *  @throws IllegalArgumentException if attribute of the given
     *          name and type is not declared in the managed type
```

```
     */
    <E> ListAttribute<X, E> getDeclaredList(String name, Class<E> elementType);

    /**
     *  Return the Map-valued attribute of the managed type that
     *  corresponds to the specified name and Java key and value
     *  types.
     *  @param name  the name of the represented attribute
     *  @param keyType  the key type of the represented attribute
     *  @param valueType  the value type of the represented attribute
     *  @return MapAttribute of the given name and key and value
     *  types
     *  @throws IllegalArgumentException if attribute of the given
     *          name and type is not present in the managed type
     */
    <K, V> MapAttribute<? super X, K, V> getMap(String name,
                                                Class<K> keyType,
                                                Class<V> valueType);

    /**
     *  Return the Map-valued attribute declared by the managed
     *  type that corresponds to the specified name and Java key
     *  and value types.
     *  @param name  the name of the represented attribute
     *  @param keyType  the key type of the represented attribute
     *  @param valueType  the value type of the represented attribute
     *  @return declared MapAttribute of the given name and key
     *          and value types
     *  @throws IllegalArgumentException if attribute of the given
     *          name and type is not declared in the managed type
     */
    <K, V> MapAttribute<X, K, V> getDeclaredMap(String name,
                                                Class<K> keyType,
                                                Class<V> valueType);

    /**
     *  Return all multi-valued attributes (Collection-, Set-,
     *  List-, and Map-valued attributes) of the managed type.
     *  Returns empty set if the managed type has no multi-valued
     *  attributes.
     *  @return Collection-, Set-, List-, and Map-valued attributes
     */
    Set<PluralAttribute<? super X, ?, ?>> getPluralAttributes();

    /**
     *  Return all multi-valued attributes (Collection-, Set-,
     *  List-, and Map-valued attributes) declared by the
     *  managed type.
```

```
     *  Returns empty set if the managed type has no declared
     *  multi-valued attributes.
     *  @return declared Collection-, Set-, List-, and Map-valued
     *          attributes
     */
    Set<PluralAttribute<X, ?, ?>> getDeclaredPluralAttributes();


//String-based:

    /**
     *  Return the attribute of the managed
     *  type that corresponds to the specified name.
     *  @param name  the name of the represented attribute
     *  @return attribute with given name
     *  @throws IllegalArgumentException if attribute of the given
     *          name is not present in the managed type
     */
    Attribute<? super X, ?> getAttribute(String name);

    /**
     *  Return the attribute declared by the managed
     *  type that corresponds to the specified name.
     *  @param name  the name of the represented attribute
     *  @return attribute with given name
     *  @throws IllegalArgumentException if attribute of the given
     *          name is not declared in the managed type
     */
    Attribute<X, ?> getDeclaredAttribute(String name);

    /**
     *  Return the single-valued attribute of the managed type that
     *  corresponds to the specified name.
     *  @param name  the name of the represented attribute
     *  @return single-valued attribute with the given name
     *  @throws IllegalArgumentException if attribute of the given
     *          name is not present in the managed type
     */
    SingularAttribute<? super X, ?> getSingularAttribute(String name);

    /**
     *  Return the single-valued attribute declared by the managed
     *  type that corresponds to the specified name.
     *  @param name  the name of the represented attribute
     *  @return declared single-valued attribute of the given
     *          name
     *  @throws IllegalArgumentException if attribute of the given
     *          name is not declared in the managed type
```

```
     */
    SingularAttribute<X, ?> getDeclaredSingularAttribute(String name);

    /**
     *  Return the Collection-valued attribute of the managed type
     *  that corresponds to the specified name.
     *  @param name  the name of the represented attribute
     *  @return CollectionAttribute of the given name
     *  @throws IllegalArgumentException if attribute of the given
     *          name is not present in the managed type
     */
    CollectionAttribute<? super X, ?> getCollection(String name);

    /**
     *  Return the Collection-valued attribute declared by the
     *  managed type that corresponds to the specified name.
     *  @param name  the name of the represented attribute
     *  @return declared CollectionAttribute of the given name
     *  @throws IllegalArgumentException if attribute of the given
     *          name is not declared in the managed type
     */
    CollectionAttribute<X, ?> getDeclaredCollection(String name);

    /**
     *  Return the Set-valued attribute of the managed type that
     *  corresponds to the specified name.
     *  @param name  the name of the represented attribute
     *  @return SetAttribute of the given name
     *  @throws IllegalArgumentException if attribute of the given
     *          name is not present in the managed type
     */
    SetAttribute<? super X, ?> getSet(String name);

    /**
     *  Return the Set-valued attribute declared by the managed type
     *  that corresponds to the specified name.
     *  @param name  the name of the represented attribute
     *  @return declared SetAttribute of the given name
     *  @throws IllegalArgumentException if attribute of the given
     *          name is not declared in the managed type
     */
    SetAttribute<X, ?> getDeclaredSet(String name);

    /**
     *  Return the List-valued attribute of the managed type that
     *  corresponds to the specified name.
     *  @param name  the name of the represented attribute
     *  @return ListAttribute of the given name
```

```
     *  @throws IllegalArgumentException if attribute of the given
     *          name is not present in the managed type
     */
    ListAttribute<? super X, ?> getList(String name);

    /**
     *  Return the List-valued attribute declared by the managed
     *  type that corresponds to the specified name.
     *  @param name  the name of the represented attribute
     *  @return declared ListAttribute of the given name
     *  @throws IllegalArgumentException if attribute of the given
     *          name is not declared in the managed type
     */
    ListAttribute<X, ?> getDeclaredList(String name);

    /**
     *  Return the Map-valued attribute of the managed type that
     *  corresponds to the specified name.
     *  @param name  the name of the represented attribute
     *  @return MapAttribute of the given name
     *  @throws IllegalArgumentException if attribute of the given
     *          name is not present in the managed type
     */
    MapAttribute<? super X, ?, ?> getMap(String name);

    /**
     *  Return the Map-valued attribute declared by the managed
     *  type that corresponds to the specified name.
     *  @param name  the name of the represented attribute
     *  @return declared MapAttribute of the given name
     *  @throws IllegalArgumentException if attribute of the given
     *          name is not declared in the managed type
     */
    MapAttribute<X, ?, ?> getDeclaredMap(String name);
}
```

5.1.4. IdentifiableType Interface

```
package jakarta.persistence.metamodel;

import java.util.Set;

/**
 *  Instances of the type <code>IdentifiableType</code> represent entity or
 *  mapped superclass types.
 *
```

```
 *  @param <X> The represented entity or mapped superclass type.
 *
 *  @since 2.0
 *
 */
public interface IdentifiableType<X> extends ManagedType<X> {

    /**
     *  Return the attribute that corresponds to the id attribute of
     *  the entity or mapped superclass.
     *  @param type  the type of the represented id attribute
     *  @return id attribute
     *  @throws IllegalArgumentException if id attribute of the given
     *          type is not present in the identifiable type or if
     *          the identifiable type has an id class
     */
    <Y> SingularAttribute<? super X, Y> getId(Class<Y> type);

    /**
     *  Return the attribute that corresponds to the id attribute
     *  declared by the entity or mapped superclass.
     *  @param type  the type of the represented declared
     *               id attribute
     *  @return declared id attribute
     *  @throws IllegalArgumentException if id attribute of the given
     *          type is not declared in the identifiable type or if
     *          the identifiable type has an id class
     */
    <Y> SingularAttribute<X, Y> getDeclaredId(Class<Y> type);

    /**
     *  Return the attribute that corresponds to the version
     *  attribute of the entity or mapped superclass.
     *  @param type  the type of the represented version attribute
     *  @return version attribute
     *  @throws IllegalArgumentException if version attribute of the
     *          given type is not present in the identifiable type
     */
    <Y> SingularAttribute<? super X, Y> getVersion(Class<Y> type);

    /**
     *  Return the attribute that corresponds to the version
     *  attribute declared by the entity or mapped superclass.
     *  @param type  the type of the represented declared version
     *               attribute
     *  @return declared version attribute
     *  @throws IllegalArgumentException if version attribute of the
     *          type is not declared in the identifiable type
```

```
     */
    <Y> SingularAttribute<X, Y> getDeclaredVersion(Class<Y> type);

    /**
     *  Return the identifiable type that corresponds to the most
     *  specific mapped superclass or entity extended by the entity
     *  or mapped superclass.
     *  @return supertype of identifiable type or null if no
     *          such supertype
     */
    IdentifiableType<? super X> getSupertype();

    /**
     *  Whether the identifiable type has a single id attribute.
     *  Returns true for a simple id or embedded id; returns false
     *  for an idclass.
     *  @return boolean indicating whether the identifiable
     *          type has a single id attribute
     */
    boolean hasSingleIdAttribute();

    /**
     *  Whether the identifiable type has a version attribute.
     *  @return boolean indicating whether the identifiable
     *          type has a version attribute
     */
    boolean hasVersionAttribute();

    /**
     *   Return the attributes corresponding to the id class of the
     *   identifiable type.
     *   @return id attributes
     *   @throws IllegalArgumentException if the identifiable type
     *           does not have an id class
     */
     Set<SingularAttribute<? super X, ?>> getIdClassAttributes();

    /**
     *  Return the type that represents the type of the id.
     *  @return type of id
     */
    Type<?> getIdType();
}
```

5.1.5. EntityType Interface

```
package jakarta.persistence.metamodel;

/**
 *  Instances of the type <code>EntityType</code> represent entity types.
 *
 *  @param <X> The represented entity type.
 *
 * @since 2.0
 *
 */
public interface EntityType<X>
            extends IdentifiableType<X>, Bindable<X>{

    /**
     *  Return the entity name.
     *  @return entity name
     */
    String getName();
}
```

5.1.6. EmbeddableType Interface

```
package jakarta.persistence.metamodel;

/**
 *  Instances of the type <code>EmbeddableType</code> represent embeddable types.
 *
 *  @param <X> The represented type.
 *
 *  @since 2.0
 *
 */
public interface EmbeddableType<X> extends ManagedType<X> {}
```

5.1.7. MappedSuperclassType Interface

```
package jakarta.persistence.metamodel;

/**
 *  Instances of the type <code>MappedSuperclassType</code> represent mapped
 *  superclass types.
 *
 *  @param <X> The represented entity type
 *  @since 2.0
 */
public interface MappedSuperclassType<X> extends IdentifiableType<X> {}
```

5.1.8. BasicType Interface

```
package jakarta.persistence.metamodel;

/**
 * Instances of the type <code>BasicType</code> represent basic types (including
 * temporal and enumerated types).
 *
 * @param <X> The type of the represented basic type
 *
 * @since 2.0
 */
public interface BasicType<X> extends Type<X> {}
```

5.1.9. Bindable Interface

```
package jakarta.persistence.metamodel;

import jakarta.persistence.criteria.Path;

/**
 * Instances of the type <code>Bindable</code> represent object or attribute types
 * that can be bound into a {@link Path Path}.
 *
 * @param <T>  The type of the represented object or attribute
 *
 * @since 2.0
 *
 */
public interface Bindable<T> {

    public static enum BindableType {

        /** Single-valued attribute type */
        SINGULAR_ATTRIBUTE,

        /** Multi-valued attribute type */
        PLURAL_ATTRIBUTE,

        /** Entity type */
        ENTITY_TYPE
    }

    /**
     *  Return the bindable type of the represented object.
     *  @return bindable type
     */
    BindableType getBindableType();

    /**
     * Return the Java type of the represented object.
     * If the bindable type of the object is <code>PLURAL_ATTRIBUTE</code>,
     * the Java element type is returned. If the bindable type is
     * <code>SINGULAR_ATTRIBUTE</code> or <code>ENTITY_TYPE</code>,
     * the Java type of the
     * represented entity or attribute is returned.
     * @return Java type
     */
    Class<T> getBindableJavaType();
}
```

5.1.10. Attribute Interface

```
package jakarta.persistence.metamodel;

/**
 * Represents an attribute of a Java type.
 *
 * @param <X> The represented type that contains the attribute
 * @param <Y> The type of the represented attribute
 *
 * @since 2.0
 */
public interface Attribute<X, Y> {

    public static enum PersistentAttributeType {

         /** Many-to-one association */
         MANY_TO_ONE,

         /** One-to-one association */
         ONE_TO_ONE,

         /** Basic attribute */
         BASIC,

         /** Embeddable class attribute */
         EMBEDDED,

         /** Many-to-many association */
         MANY_TO_MANY,

         /** One-to-many association */
         ONE_TO_MANY,

         /** Element collection */
         ELEMENT_COLLECTION
    }

    /**
     * Return the name of the attribute.
     * @return name
     */
    String getName();

    /**
     *  Return the persistent attribute type for the attribute.
     *  @return persistent attribute type
```

```
     */
    PersistentAttributeType getPersistentAttributeType();

    /**
     *  Return the managed type representing the type in which
     *  the attribute was declared.
     *  @return declaring type
     */
    ManagedType<X> getDeclaringType();

    /**
     *  Return the Java type of the represented attribute.
     *  @return Java type
     */
    Class<Y> getJavaType();

    /**
     *  Return the <code>java.lang.reflect.Member</code> for the represented
     *  attribute.
     *  @return corresponding <code>java.lang.reflect.Member</code>
     */
    java.lang.reflect.Member getJavaMember();

    /**
     *  Is the attribute an association.
     *  @return boolean indicating whether the attribute
     *          corresponds to an association
     */
    boolean isAssociation();

    /**
     *  Is the attribute collection-valued (represents a Collection,
     *  Set, List, or Map).
     *  @return boolean indicating whether the attribute is
     *          collection-valued
     */
    boolean isCollection();
}
```

5.1.11. SingularAttribute Interface

```
package jakarta.persistence.metamodel;

/**
 * Instances of the type <code>SingularAttribute</code> represents persistent
 * single-valued properties or fields.
 *
 * @param <X> The type containing the represented attribute
 * @param <T> The type of the represented attribute
 *
 * @since 2.0
 */
public interface SingularAttribute<X, T>
        extends Attribute<X, T>, Bindable<T> {

    /**
     *  Is the attribute an id attribute.  This method will return
     *  true if the attribute is an attribute that corresponds to
     *  a simple id, an embedded id, or an attribute of an id class.
     *  @return boolean indicating whether the attribute is an id
     */
    boolean isId();

    /**
     *  Is the attribute a version attribute.
     *  @return boolean indicating whether the attribute is
     *          a version attribute
     */
    boolean isVersion();

    /**
     *  Can the attribute be null.
     *  @return boolean indicating whether the attribute can
     *          be null
     */
    boolean isOptional();

    /**
     * Return the type that represents the type of the attribute.
     * @return type of attribute
     */
    Type<T> getType();
}
```

5.1.12. PluralAttribute Interface

```
package jakarta.persistence.metamodel;

/**
 * Instances of the type <code>PluralAttribute</code> represent
 * persistent collection-valued attributes.
 *
 * @param <X> The type the represented collection belongs to
 * @param <C> The type of the represented collection
 * @param <E> The element type of the represented collection
 *
 * @since 2.0
 */
public interface PluralAttribute<X, C, E>
        extends Attribute<X, C>, Bindable<E> {

    public static enum CollectionType {

        /** Collection-valued attribute */
        COLLECTION,

        /** Set-valued attribute */
        SET,

        /** List-valued attribute */
        LIST,

        /** Map-valued attribute */
        MAP
    }

    /**
     * Return the collection type.
     * @return collection type
     */
    CollectionType getCollectionType();

    /**
     * Return the type representing the element type of the
     * collection.
     * @return element type
     */
    Type<E> getElementType();
}
```

5.1.13. CollectionAttribute Interface

```
package jakarta.persistence.metamodel;

/**
 * Instances of the type <code>CollectionAttribute</code> represent persistent
 * <code>java.util.Collection</code>-valued attributes.
 *
 * @param <X> The type the represented Collection belongs to
 * @param <E> The element type of the represented Collection
 *
 * @since 2.0
 *
 */
public interface CollectionAttribute<X, E>
    extends PluralAttribute<X, java.util.Collection<E>, E> {}
```

5.1.14. SetAttribute Interface

```
package jakarta.persistence.metamodel;

/**
 * Instances of the type <code>SetAttribute</code> represent
 * persistent <code>java.util.Set</code>-valued attributes.
 *
 * @param <X> The type the represented Set belongs to
 * @param <E> The element type of the represented Set
 *
 * @since 2.0
 */
public interface SetAttribute<X, E>
    extends PluralAttribute<X, java.util.Set<E>, E> {}
```

5.1.15. ListAttribute Interface

```
package jakarta.persistence.metamodel;

/**
 * Instances of the type <code>ListAttribute</code> represent persistent
 * <code>javax.util.List</code>-valued attributes.
 *
 * @param <X> The type the represented List belongs to
 * @param <E> The element type of the represented List
 *
 * @since 2.0
 *
 */
public interface ListAttribute<X, E>
        extends PluralAttribute<X, java.util.List<E>, E> {}
```

5.1.16. MapAttribute Interface

```
package jakarta.persistence.metamodel;

/**
 * Instances of the type <code>MapAttribute</code> represent
 * persistent <code>java.util.Map</code>-valued attributes.
 *
 * @param <X> The type the represented Map belongs to
 * @param <K> The type of the key of the represented Map
 * @param <V> The type of the value of the represented Map
 *
 * @since 2.0
 *
 */
public interface MapAttribute<X, K, V>
    extends PluralAttribute<X, java.util.Map<K, V>, V> {

    /**
     * Return the Java type of the map key.
     * @return Java key type
     */
    Class<K> getKeyJavaType();

    /**
     * Return the type representing the key type of the map.
     * @return type representing key type
     */
    Type<K> getKeyType();
}
```

5.1.17. StaticMetamodel Annotation

```
package jakarta.persistence.metamodel;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * The <code>StaticMetamodel</code> annotation specifies that the class
 * is a metamodel class that represents the entity, mapped
 * superclass, or embeddable class designated by the value
 * element.
 *
 * @since 2.0
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface StaticMetamodel {

    /**
     * Class being modelled by the annotated class.
     */
    Class<?> value();
}
```

Chapter 6. Criteria API

The Java Persistence Criteria API is used to define queries through the construction of object-based query definition objects, rather than use of the string-based approach of the Java Persistence query language described in Chapter 4.

This chapter provides the full definition of the Criteria API.

6.1. Overview

The Java Persistence Criteria API, like the Java Persistence query language is based on the abstract persistence schema of entities, their embedded objects, and their relationships as its data model. This abstract persistence schema is materialized in the form of metamodel objects over which the Criteria API operates. The semantics of criteria queries are designed to reflect those of Java Persistence query language queries.

The syntax of the Criteria API is designed to allow the construction of an object-based query "graph", whose nodes correspond to the semantic query elements.

Java language variables can be used to reference individual nodes in a criteria query object as it is constructed and/or modified. Such variables, when used to refer to the entities and embeddable types that constitute the query domain, play a role analogous to that of the identification variables of the Java Persistence query language.

These concepts are further described in the sections that follow. The metamodel on which criteria queries are based is presented in Chapter 5. The static metamodel classes that can be used in constructing strongly-typed criteria queries are described in Section 6.2. The *javax.persistence.criteria* interfaces are presented in Section 6.3. Sections Section 6.4 through Section 6.8 describe the construction and modification of criteria query objects. Additional requirements on the persistence provider are described in Section 6.9.

6.2. Metamodel

Java Persistence criteria queries are based on a metamodel of the managed classes of the persistence unit. Static metamodel classes corresponding to the managed classes of the persistence unit can be generated by means of an annotation processor or can be created by the application developer, or the metamodel can be accessed dynamically by use of the *javax.persistence.metamodel.Metamodel* interface. The *getMetamodel* method of the *EntityManagerFactory* or *EntityManager* interface can be used to obtain a *Metamodel* instance.

6.2.1. Static Metamodel Classes

In the typical case, an annotation processor is expected to be used to produce static metamodel classes corresponding to the entities, mapped superclasses, and embeddable classes in the persistence unit. A

static metamodel class models the persistent state and relationships of the corresponding managed class. For portability, an annotation processor should generate a canonical metamodel as defined below.

6.2.1.1. Canonical Metamodel

This specification defines as follows a canonical metamodel and the structure of canonical metamodel classes.

For every managed class in the persistence unit, a corresponding metamodel class is produced as follows:

- For each managed class *X* in package *p*, a metamodel class *X_* in package *p* is created. [74: We expect that the option of different packages will be provided in a future release of this specification.]
- The name of the metamodel class is derived from the name of the managed class by appending " _ " to the name of the managed class.
- The metamodel class *X_* must be annotated with the *javax.persistence.StaticMetamodel* annotation [75: If the class was generated, the *javax.annotation.Generated* annotation should be used to annotate the class. The use of any annotations other than *StaticMetamodel* and *Generated* on static metamodel classes is undefined.].
- If class *X* extends another class *S*, where *S* is the most derived managed class (i.e., entity or mapped superclass) extended by *X*, then class *X must extend class _S, where _S_* is the metamodel class created for *S*.
- For every persistent non-collection-valued attribute *y* declared by class *X*, where the type of *y* is *Y*, the metamodel class must contain a declaration as follows:

```
public static volatile SingularAttribute<X, Y> y;
```

- For every persistent collection-valued attribute *z* declared by class *X*, where the element type of *z* is *Z*, the metamodel class must contain a declaration as follows:
  - if the collection type of *z* is *java.util.Collection*, then

```
public static volatile CollectionAttribute<X, Z> z;
```

  - if the collection type of *z* is *java.util.Set*, then

```
public static volatile SetAttribute<X, Z> z;
```

  - if the collection type of *z* is *java.util.List*, then

```
public static volatile ListAttribute<X, Z> z;
```

- if the collection type of *z* is *java.util.Map*, then

```
public static volatile MapAttribute<X, K, Z> z;
```

where *K* is the type of the key of the map in class *X*

Import statements must be included for the needed *javax.persistence.metamodel* types as appropriate (e.g., *javax.persistence.metamodel.SingularAttribute*, *javax.persistence.metamodel.CollectionAttribute*, *javax.persistence.metamodel.SetAttribute*, *javax.persistence.metamodel.ListAttribute*, *javax.persistence.metamodel.MapAttribute*) and all classes *X*, *Y*, *Z*, and *K*.

Implementations of this specification are not required to support the use of non-canonical metamodel classes. Applications that use non-canonical metamodel classes will not be portable.

6.2.1.2. Example

Assume the *Order* entity below.

```
package com.example;

import java.util.Set;
import java.math.BigDecimal;

@Entity
public class Order {
    @Id
    Integer orderId;

    @ManyToOne
    Customer customer;

    @OneToMany
    Set<Item> lineItems;

    Address shippingAddress;

    BigDecimal totalCost;

    // ...
}
```

The corresponding canonical metamodel class, *Order*, is as follows:

```
package com.example;

import java.math.BigDecimal;
import jakarta.persistence.metamodel.SingularAttribute;
import jakarta.persistence.metamodel.SetAttribute;
import jakarta.persistence.metamodel.StaticMetamodel;

@StaticMetamodel(Order.class)
public class Order {
    public static volatile SingularAttribute<Order, Integer> orderId;
    public static volatile SingularAttribute<Order, Customer> customer;
    public static volatile SetAttribute<Order, Item> lineItems;
    public static volatile SingularAttribute<Order, Address> shippingAddress;
    public static volatile SingularAttribute<Order, BigDecimal> totalCost;
}
```

6.2.2. Bootstrapping

When the entity manager factory for a persistence unit is created, it is the responsibility of the persistence provider to initialize the state of the metamodel classes of the persistence unit. Any generated metamodel classes must be accessible on the classpath.

Persistence providers must support the use of canonical metamodel classes. Persistence providers may, but are not required to, support the use of non-canonical metamodel classes.

6.3. Criteria API Interfaces

6.3.1. CriteriaBuilder Interface

```
package jakarta.persistence.criteria;

import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.Collection;
import java.util.Map;
import java.util.Set;
import jakarta.persistence.Tuple;

/**
 * Used to construct criteria queries, compound selections,
 * expressions, predicates, orderings.
 *
```

```
 * <p> Note that <code>Predicate</code> is used instead of
<code>Expression&#060;Boolean&#062;</code>
 * in this API in order to work around the fact that Java
 * generics are not compatible with varags.
 *
 * @since 2.0
 */
public interface CriteriaBuilder {

    /**
     *  Create a <code>CriteriaQuery</code> object.
     *  @return criteria query object
     */
    CriteriaQuery<Object> createQuery();

    /**
     *  Create a <code>CriteriaQuery</code> object with the specified result
     *  type.
     *  @param resultClass  type of the query result
     *  @return criteria query object
     */
    <T> CriteriaQuery<T> createQuery(Class<T> resultClass);

    /**
     *  Create a <code>CriteriaQuery</code> object that returns a tuple of
     *  objects as its result.
     *  @return criteria query object
     */
    CriteriaQuery<Tuple> createTupleQuery();

    // methods to construct queries for bulk updates and deletes:

    /**
     *  Create a <code>CriteriaUpdate</code> query object to perform a bulk update
operation.
     *  @param targetEntity  target type for update operation
     *  @return the query object
     *  @since 2.1
     */
    <T> CriteriaUpdate<T> createCriteriaUpdate(Class<T> targetEntity);

    /**
     *  Create a <code>CriteriaDelete</code> query object to perform a bulk delete
operation.
     *  @param targetEntity  target type for delete operation
     *  @return the query object
     *  @since 2.1
     */
```

```
    <T> CriteriaDelete<T> createCriteriaDelete(Class<T> targetEntity);


    // selection construction methods:

    /**
     * Create a selection item corresponding to a constructor.
     * This method is used to specify a constructor that will be
     * applied to the results of the query execution. If the
     * constructor is for an entity class, the resulting entities
     * will be in the new state after the query is executed.
     * @param resultClass  class whose instance is to be constructed
     * @param selections  arguments to the constructor
     * @return compound selection item
     * @throws IllegalArgumentException if an argument is a
     *         tuple- or array-valued selection item
     */
    <Y> CompoundSelection<Y> construct(Class<Y> resultClass, Selection<?>... selections);

    /**
     * Create a tuple-valued selection item.
     * @param selections  selection items
     * @return tuple-valued compound selection
     * @throws IllegalArgumentException if an argument is a
     *         tuple- or array-valued selection item
     */
    CompoundSelection<Tuple> tuple(Selection<?>... selections);

    /**
     * Create an array-valued selection item.
     * @param selections  selection items
     * @return array-valued compound selection
     * @throws IllegalArgumentException if an argument is a
     *         tuple- or array-valued selection item
     */
    CompoundSelection<Object[]> array(Selection<?>... selections);


    //ordering:

    /**
     * Create an ordering by the ascending value of the expression.
     * @param x  expression used to define the ordering
     * @return ascending ordering corresponding to the expression
     */
    Order asc(Expression<?> x);

    /**
```

```
     * Create an ordering by the descending value of the expression.
     * @param x  expression used to define the ordering
     * @return descending ordering corresponding to the expression
     */
    Order desc(Expression<?> x);


    //aggregate functions:

    /**
     * Create an aggregate expression applying the avg operation.
     * @param x  expression representing input value to avg operation
     * @return avg expression
     */
    <N extends Number> Expression<Double> avg(Expression<N> x);

    /**
     * Create an aggregate expression applying the sum operation.
     * @param x  expression representing input value to sum operation
     * @return sum expression
     */
    <N extends Number> Expression<N> sum(Expression<N> x);

    /**
     * Create an aggregate expression applying the sum operation to an
     * Integer-valued expression, returning a Long result.
     * @param x  expression representing input value to sum operation
     * @return sum expression
     */
    Expression<Long> sumAsLong(Expression<Integer> x);

    /**
     * Create an aggregate expression applying the sum operation to a
     * Float-valued expression, returning a Double result.
     * @param x  expression representing input value to sum operation
     * @return sum expression
     */
    Expression<Double> sumAsDouble(Expression<Float> x);

    /**
     * Create an aggregate expression applying the numerical max
     * operation.
     * @param x  expression representing input value to max operation
     * @return max expression
     */
    <N extends Number> Expression<N> max(Expression<N> x);

    /**
```

```
     * Create an aggregate expression applying the numerical min
     * operation.
     * @param x  expression representing input value to min operation
     * @return min expression
     */
    <N extends Number> Expression<N> min(Expression<N> x);

    /**
     * Create an aggregate expression for finding the greatest of
     * the values (strings, dates, etc).
     * @param x  expression representing input value to greatest
     *           operation
     * @return greatest expression
     */
    <X extends Comparable<? super X>> Expression<X> greatest(Expression<X> x);

    /**
     * Create an aggregate expression for finding the least of
     * the values (strings, dates, etc).
     * @param x  expression representing input value to least
     *           operation
     * @return least expression
     */
    <X extends Comparable<? super X>> Expression<X> least(Expression<X> x);

    /**
     * Create an aggregate expression applying the count operation.
     * @param x  expression representing input value to count
     *           operation
     * @return count expression
     */
    Expression<Long> count(Expression<?> x);

    /**
     * Create an aggregate expression applying the count distinct
     * operation.
     * @param x  expression representing input value to
     *        count distinct operation
     * @return count distinct expression
     */
    Expression<Long> countDistinct(Expression<?> x);



    //subqueries:

    /**
     * Create a predicate testing the existence of a subquery result.
```

```
     * @param subquery  subquery whose result is to be tested
     * @return exists predicate
     */
    Predicate exists(Subquery<?> subquery);

    /**
     * Create an all expression over the subquery results.
     * @param subquery  subquery
     * @return all expression
     */
    <Y> Expression<Y> all(Subquery<Y> subquery);

    /**
     * Create a some expression over the subquery results.
     * This expression is equivalent to an <code>any</code> expression.
     * @param subquery  subquery
     * @return some expression
     */
    <Y> Expression<Y> some(Subquery<Y> subquery);

    /**
     * Create an any expression over the subquery results.
     * This expression is equivalent to a <code>some</code> expression.
     * @param subquery  subquery
     * @return any expression
     */
    <Y> Expression<Y> any(Subquery<Y> subquery);


    //boolean functions:

    /**
     * Create a conjunction of the given boolean expressions.
     * @param x  boolean expression
     * @param y  boolean expression
     * @return and predicate
     */
    Predicate and(Expression<Boolean> x, Expression<Boolean> y);

    /**
     * Create a conjunction of the given restriction predicates.
     * A conjunction of zero predicates is true.
     * @param restrictions  zero or more restriction predicates
     * @return and predicate
     */
    Predicate and(Predicate... restrictions);

    /**
```

```
     * Create a disjunction of the given boolean expressions.
     * @param x  boolean expression
     * @param y  boolean expression
     * @return or predicate
     */
    Predicate or(Expression<Boolean> x, Expression<Boolean> y);

    /**
     * Create a disjunction of the given restriction predicates.
     * A disjunction of zero predicates is false.
     * @param restrictions  zero or more restriction predicates
     * @return or predicate
     */
    Predicate or(Predicate... restrictions);

    /**
     * Create a negation of the given restriction.
     * @param restriction  restriction expression
     * @return not predicate
     */
    Predicate not(Expression<Boolean> restriction);

    /**
     * Create a conjunction (with zero conjuncts).
     * A conjunction with zero conjuncts is true.
     * @return and predicate
     */
    Predicate conjunction();

    /**
     * Create a disjunction (with zero disjuncts).
     * A disjunction with zero disjuncts is false.
     * @return or predicate
     */
    Predicate disjunction();


    //turn Expression<Boolean> into a Predicate
    //useful for use with varargs methods

    /**
     * Create a predicate testing for a true value.
     * @param x  expression to be tested
     * @return predicate
     */
    Predicate isTrue(Expression<Boolean> x);

    /**
```

```
     * Create a predicate testing for a false value.
     * @param x  expression to be tested
     * @return predicate
     */
    Predicate isFalse(Expression<Boolean> x);


    //null tests:

    /**
     * Create a predicate to test whether the expression is null.
     * @param x expression
     * @return is-null predicate
     */
    Predicate isNull(Expression<?> x);

    /**
     * Create a predicate to test whether the expression is not null.
     * @param x expression
     * @return is-not-null predicate
     */
    Predicate isNotNull(Expression<?> x);

    //equality:

    /**
     * Create a predicate for testing the arguments for equality.
     * @param x  expression
     * @param y  expression
     * @return equality predicate
     */
    Predicate equal(Expression<?> x, Expression<?> y);

    /**
     * Create a predicate for testing the arguments for equality.
     * @param x  expression
     * @param y  object
     * @return equality predicate
     */
    Predicate equal(Expression<?> x, Object y);

    /**
     * Create a predicate for testing the arguments for inequality.
     * @param x  expression
     * @param y  expression
     * @return inequality predicate
     */
    Predicate notEqual(Expression<?> x, Expression<?> y);
```

```
    /**
     * Create a predicate for testing the arguments for inequality.
     * @param x  expression
     * @param y  object
     * @return inequality predicate
     */
    Predicate notEqual(Expression<?> x, Object y);


    //comparisons for generic (non-numeric) operands:

    /**
     * Create a predicate for testing whether the first argument is
     * greater than the second.
     * @param x  expression
     * @param y  expression
     * @return greater-than predicate
     */
    <Y extends Comparable<? super Y>> Predicate greaterThan(Expression<? extends Y> x,
Expression<? extends Y> y);

    /**
     * Create a predicate for testing whether the first argument is
     * greater than the second.
     * @param x  expression
     * @param y  value
     * @return greater-than predicate
     */
    <Y extends Comparable<? super Y>> Predicate greaterThan(Expression<? extends Y> x, Y
y);

    /**
     * Create a predicate for testing whether the first argument is
     * greater than or equal to the second.
     * @param x  expression
     * @param y  expression
     * @return greater-than-or-equal predicate
     */
    <Y extends Comparable<? super Y>> Predicate greaterThanOrEqualTo(Expression<? extends
Y> x, Expression<? extends Y> y);

    /**
     * Create a predicate for testing whether the first argument is
     * greater than or equal to the second.
     * @param x  expression
     * @param y  value
     * @return greater-than-or-equal predicate
```

```
     */
    <Y extends Comparable<? super Y>> Predicate greaterThanOrEqualTo(Expression<? extends
Y> x, Y y);

    /**
     * Create a predicate for testing whether the first argument is
     * less than the second.
     * @param x  expression
     * @param y  expression
     * @return less-than predicate
     */
    <Y extends Comparable<? super Y>> Predicate lessThan(Expression<? extends Y> x,
Expression<? extends Y> y);

    /**
     * Create a predicate for testing whether the first argument is
     * less than the second.
     * @param x  expression
     * @param y  value
     * @return less-than predicate
     */
    <Y extends Comparable<? super Y>> Predicate lessThan(Expression<? extends Y> x, Y y);

    /**
     * Create a predicate for testing whether the first argument is
     * less than or equal to the second.
     * @param x  expression
     * @param y  expression
     * @return less-than-or-equal predicate
     */
    <Y extends Comparable<? super Y>> Predicate lessThanOrEqualTo(Expression<? extends Y>
x, Expression<? extends Y> y);

    /**
     * Create a predicate for testing whether the first argument is
     * less than or equal to the second.
     * @param x  expression
     * @param y  value
     * @return less-than-or-equal predicate
     */
    <Y extends Comparable<? super Y>> Predicate lessThanOrEqualTo(Expression<? extends Y>
x, Y y);

    /**
     * Create a predicate for testing whether the first argument is
     * between the second and third arguments in value.
     * @param v  expression
     * @param x  expression
```

```
     * @param y  expression
     * @return between predicate
     */
    <Y extends Comparable<? super Y>> Predicate between(Expression<? extends Y> v,
Expression<? extends Y> x, Expression<? extends Y> y);

    /**
     * Create a predicate for testing whether the first argument is
     * between the second and third arguments in value.
     * @param v  expression
     * @param x  value
     * @param y  value
     * @return between predicate
     */
    <Y extends Comparable<? super Y>> Predicate between(Expression<? extends Y> v, Y x, Y
y);


    //comparisons for numeric operands:

    /**
     * Create a predicate for testing whether the first argument is
     * greater than the second.
     * @param x  expression
     * @param y  expression
     * @return greater-than predicate
     */
    Predicate gt(Expression<? extends Number> x, Expression<? extends Number> y);

    /**
     * Create a predicate for testing whether the first argument is
     * greater than the second.
     * @param x  expression
     * @param y  value
     * @return greater-than predicate
     */
    Predicate gt(Expression<? extends Number> x, Number y);

    /**
     * Create a predicate for testing whether the first argument is
     * greater than or equal to the second.
     * @param x  expression
     * @param y  expression
     * @return greater-than-or-equal predicate
     */
    Predicate ge(Expression<? extends Number> x, Expression<? extends Number> y);

    /**
```

```
     * Create a predicate for testing whether the first argument is
     * greater than or equal to the second.
     * @param x  expression
     * @param y  value
     * @return greater-than-or-equal predicate
     */
    Predicate ge(Expression<? extends Number> x, Number y);

    /**
     * Create a predicate for testing whether the first argument is
     * less than the second.
     * @param x  expression
     * @param y  expression
     * @return less-than predicate
     */
    Predicate lt(Expression<? extends Number> x, Expression<? extends Number> y);

    /**
     * Create a predicate for testing whether the first argument is
     * less than the second.
     * @param x  expression
     * @param y  value
     * @return less-than predicate
     */
    Predicate lt(Expression<? extends Number> x, Number y);

    /**
     * Create a predicate for testing whether the first argument is
     * less than or equal to the second.
     * @param x  expression
     * @param y  expression
     * @return less-than-or-equal predicate
     */
    Predicate le(Expression<? extends Number> x, Expression<? extends Number> y);

    /**
     * Create a predicate for testing whether the first argument is
     * less than or equal to the second.
     * @param x  expression
     * @param y  value
     * @return less-than-or-equal predicate
     */
    Predicate le(Expression<? extends Number> x, Number y);


    //numerical operations:

    /**
```

```
     * Create an expression that returns the arithmetic negation
     * of its argument.
     * @param x expression
     * @return arithmetic negation
     */
    <N extends Number> Expression<N> neg(Expression<N> x);

    /**
     * Create an expression that returns the absolute value
     * of its argument.
     * @param x expression
     * @return absolute value
     */
    <N extends Number> Expression<N> abs(Expression<N> x);

    /**
     * Create an expression that returns the sum
     * of its arguments.
     * @param x expression
     * @param y expression
     * @return sum
     */
    <N extends Number> Expression<N> sum(Expression<? extends N> x, Expression<? extends
N> y);

    /**
     * Create an expression that returns the sum
     * of its arguments.
     * @param x expression
     * @param y value
     * @return sum
     */
    <N extends Number> Expression<N> sum(Expression<? extends N> x, N y);

    /**
     * Create an expression that returns the sum
     * of its arguments.
     * @param x value
     * @param y expression
     * @return sum
     */
    <N extends Number> Expression<N> sum(N x, Expression<? extends N> y);

    /**
     * Create an expression that returns the product
     * of its arguments.
     * @param x expression
     * @param y expression
```

```
     * @return product
     */
    <N extends Number> Expression<N> prod(Expression<? extends N> x, Expression<? extends
N> y);

    /**
     * Create an expression that returns the product
     * of its arguments.
     * @param x expression
     * @param y value
     * @return product
     */
    <N extends Number> Expression<N> prod(Expression<? extends N> x, N y);

    /**
     * Create an expression that returns the product
     * of its arguments.
     * @param x value
     * @param y expression
     * @return product
     */
    <N extends Number> Expression<N> prod(N x, Expression<? extends N> y);

    /**
     * Create an expression that returns the difference
     * between its arguments.
     * @param x expression
     * @param y expression
     * @return difference
     */
    <N extends Number> Expression<N> diff(Expression<? extends N> x, Expression<? extends
N> y);

    /**
     * Create an expression that returns the difference
     * between its arguments.
     * @param x expression
     * @param y value
     * @return difference
     */
    <N extends Number> Expression<N> diff(Expression<? extends N> x, N y);

    /**
     * Create an expression that returns the difference
     * between its arguments.
     * @param x value
     * @param y expression
     * @return difference
```

```
     */
    <N extends Number> Expression<N> diff(N x, Expression<? extends N> y);

    /**
     * Create an expression that returns the quotient
     * of its arguments.
     * @param x expression
     * @param y expression
     * @return quotient
     */
    Expression<Number> quot(Expression<? extends Number> x, Expression<? extends Number>
y);

    /**
     * Create an expression that returns the quotient
     * of its arguments.
     * @param x expression
     * @param y value
     * @return quotient
     */
    Expression<Number> quot(Expression<? extends Number> x, Number y);

    /**
     * Create an expression that returns the quotient
     * of its arguments.
     * @param x value
     * @param y expression
     * @return quotient
     */
    Expression<Number> quot(Number x, Expression<? extends Number> y);

    /**
     * Create an expression that returns the modulus
     * of its arguments.
     * @param x expression
     * @param y expression
     * @return modulus
     */
    Expression<Integer> mod(Expression<Integer> x, Expression<Integer> y);

    /**
     * Create an expression that returns the modulus
     * of its arguments.
     * @param x expression
     * @param y value
     * @return modulus
     */
    Expression<Integer> mod(Expression<Integer> x, Integer y);
```

```
    /**
     * Create an expression that returns the modulus
     * of its arguments.
     * @param x value
     * @param y expression
     * @return modulus
     */
    Expression<Integer> mod(Integer x, Expression<Integer> y);

    /**
     * Create an expression that returns the square root
     * of its argument.
     * @param x expression
     * @return square root
     */
    Expression<Double> sqrt(Expression<? extends Number> x);


    //typecasts:

    /**
     * Typecast.  Returns same expression object.
     * @param number  numeric expression
     * @return Expression&#060;Long&#062;
     */
    Expression<Long> toLong(Expression<? extends Number> number);

    /**
     * Typecast.  Returns same expression object.
     * @param number  numeric expression
     * @return Expression&#060;Integer&#062;
     */
    Expression<Integer> toInteger(Expression<? extends Number> number);

    /**
     * Typecast. Returns same expression object.
     * @param number  numeric expression
     * @return Expression&#060;Float&#062;
     */
    Expression<Float> toFloat(Expression<? extends Number> number);

    /**
     * Typecast.  Returns same expression object.
     * @param number  numeric expression
     * @return Expression&#060;Double&#062;
     */
    Expression<Double> toDouble(Expression<? extends Number> number);
```

```
    /**
     * Typecast.  Returns same expression object.
     * @param number  numeric expression
     * @return Expression&#060;BigDecimal&#062;
     */
    Expression<BigDecimal> toBigDecimal(Expression<? extends Number> number);

    /**
     * Typecast.  Returns same expression object.
     * @param number  numeric expression
     * @return Expression&#060;BigInteger&#062;
     */
    Expression<BigInteger> toBigInteger(Expression<? extends Number> number);

    /**
     * Typecast.  Returns same expression object.
     * @param character expression
     * @return Expression&#060;String&#062;
     */
    Expression<String> toString(Expression<Character> character);


    //literals:

    /**
     * Create an expression for a literal.
     * @param value  value represented by the expression
     * @return expression literal
     * @throws IllegalArgumentException if value is null
     */
    <T> Expression<T> literal(T value);

    /**
     * Create an expression for a null literal with the given type.
     * @param resultClass  type of the null literal
     * @return null expression literal
     */
    <T> Expression<T> nullLiteral(Class<T> resultClass);

    //parameters:

    /**
     * Create a parameter expression.
     * @param paramClass parameter class
     * @return parameter expression
     */
    <T> ParameterExpression<T> parameter(Class<T> paramClass);
```

```
    /**
     * Create a parameter expression with the given name.
     * @param paramClass parameter class
     * @param name  name that can be used to refer to
     *              the parameter
     * @return parameter expression
     */
    <T> ParameterExpression<T> parameter(Class<T> paramClass, String name);


    //collection operations:

    /**
     *  Create a predicate that tests whether a collection is empty.
     *  @param collection expression
     *  @return is-empty predicate
     */
    <C extends Collection<?>> Predicate isEmpty(Expression<C> collection);

    /**
     *  Create a predicate that tests whether a collection is
     *  not empty.
     *  @param collection expression
     *  @return is-not-empty predicate
     */
    <C extends Collection<?>> Predicate isNotEmpty(Expression<C> collection);

    /**
     * Create an expression that tests the size of a collection.
     * @param collection expression
     * @return size expression
     */
    <C extends java.util.Collection<?>> Expression<Integer> size(Expression<C>
collection);

    /**
     * Create an expression that tests the size of a collection.
     * @param collection collection
     * @return size expression
     */
    <C extends Collection<?>> Expression<Integer> size(C collection);

    /**
     *  Create a predicate that tests whether an element is
     *  a member of a collection.
     *  If the collection is empty, the predicate will be false.
     *  @param elem element expression
```

```
     *  @param collection expression
     *  @return is-member predicate
     */
    <E, C extends Collection<E>> Predicate isMember(Expression<E> elem, Expression<C>
collection);

    /**
     *  Create a predicate that tests whether an element is
     *  a member of a collection.
     *  If the collection is empty, the predicate will be false.
     *  @param elem element
     *  @param collection expression
     *  @return is-member predicate
     */
    <E, C extends Collection<E>> Predicate isMember(E elem, Expression<C> collection);

    /**
     *  Create a predicate that tests whether an element is
     *  not a member of a collection.
     *  If the collection is empty, the predicate will be true.
     *  @param elem element expression
     *  @param collection expression
     *  @return is-not-member predicate
     */
    <E, C extends Collection<E>> Predicate isNotMember(Expression<E> elem, Expression<C>
collection);

    /**
     *  Create a predicate that tests whether an element is
     *  not a member of a collection.
     *  If the collection is empty, the predicate will be true.
     *  @param elem element
     *  @param collection expression
     *  @return is-not-member predicate
     */
    <E, C extends Collection<E>> Predicate isNotMember(E elem, Expression<C> collection);


    //get the values and keys collections of the Map, which may then
    //be passed to size(), isMember(), isEmpty(), etc

    /**
     * Create an expression that returns the values of a map.
     * @param map  map
     * @return collection expression
     */
    <V, M extends Map<?, V>> Expression<Collection<V>> values(M map);
```

```
    /**
     * Create an expression that returns the keys of a map.
     * @param map  map
     * @return set expression
     */
    <K, M extends Map<K, ?>> Expression<Set<K>> keys(M map);


    //string functions:

    /**
     * Create a predicate for testing whether the expression
     * satisfies the given pattern.
     * @param x  string expression
     * @param pattern  string expression
     * @return like predicate
     */
    Predicate like(Expression<String> x, Expression<String> pattern);

    /**
     * Create a predicate for testing whether the expression
     * satisfies the given pattern.
     * @param x  string expression
     * @param pattern  string
     * @return like predicate
     */
    Predicate like(Expression<String> x, String pattern);

    /**
     * Create a predicate for testing whether the expression
     * satisfies the given pattern.
     * @param x  string expression
     * @param pattern  string expression
     * @param escapeChar  escape character expression
     * @return like predicate
     */
    Predicate like(Expression<String> x, Expression<String> pattern, Expression<
Character> escapeChar);

    /**
     * Create a predicate for testing whether the expression
     * satisfies the given pattern.
     * @param x  string expression
     * @param pattern  string expression
     * @param escapeChar  escape character
     * @return like predicate
     */
    Predicate like(Expression<String> x, Expression<String> pattern, char escapeChar);
```

```
    /**
     * Create a predicate for testing whether the expression
     * satisfies the given pattern.
     * @param x  string expression
     * @param pattern  string
     * @param escapeChar  escape character expression
     * @return like predicate
     */
    Predicate like(Expression<String> x, String pattern, Expression<Character>
escapeChar);

    /**
     * Create a predicate for testing whether the expression
     * satisfies the given pattern.
     * @param x  string expression
     * @param pattern  string
     * @param escapeChar  escape character
     * @return like predicate
     */
    Predicate like(Expression<String> x, String pattern, char escapeChar);

    /**
     * Create a predicate for testing whether the expression
     * does not satisfy the given pattern.
     * @param x  string expression
     * @param pattern  string expression
     * @return not-like predicate
     */
    Predicate notLike(Expression<String> x, Expression<String> pattern);

    /**
     * Create a predicate for testing whether the expression
     * does not satisfy the given pattern.
     * @param x  string expression
     * @param pattern  string
     * @return not-like predicate
     */
    Predicate notLike(Expression<String> x, String pattern);

    /**
     * Create a predicate for testing whether the expression
     * does not satisfy the given pattern.
     * @param x  string expression
     * @param pattern  string expression
     * @param escapeChar  escape character expression
     * @return not-like predicate
     */
```

```
    Predicate notLike(Expression<String> x, Expression<String> pattern, Expression
<Character> escapeChar);

    /**
     * Create a predicate for testing whether the expression
     * does not satisfy the given pattern.
     * @param x  string expression
     * @param pattern  string expression
     * @param escapeChar  escape character
     * @return not-like predicate
     */
    Predicate notLike(Expression<String> x, Expression<String> pattern, char escapeChar);

    /**
     * Create a predicate for testing whether the expression
     * does not satisfy the given pattern.
     * @param x  string expression
     * @param pattern  string
     * @param escapeChar  escape character expression
     * @return not-like predicate
     */
    Predicate notLike(Expression<String> x, String pattern, Expression<Character>
escapeChar);

   /**
     * Create a predicate for testing whether the expression
     * does not satisfy the given pattern.
     * @param x  string expression
     * @param pattern  string
     * @param escapeChar  escape character
     * @return not-like predicate
     */
    Predicate notLike(Expression<String> x, String pattern, char escapeChar);

    /**
     *  Create an expression for string concatenation.
     *  @param x  string expression
     *  @param y  string expression
     *  @return expression corresponding to concatenation
     */
    Expression<String> concat(Expression<String> x, Expression<String> y);

    /**
     *  Create an expression for string concatenation.
     *  @param x  string expression
     *  @param y  string
     *  @return expression corresponding to concatenation
     */
```

```
    Expression<String> concat(Expression<String> x, String y);

    /**
     *  Create an expression for string concatenation.
     *  @param x  string
     *  @param y  string expression
     *  @return expression corresponding to concatenation
     */
    Expression<String> concat(String x, Expression<String> y);

    /**
     *  Create an expression for substring extraction.
     *  Extracts a substring starting at the specified position
     *  through to end of the string.
     *  First position is 1.
     *  @param x  string expression
     *  @param from  start position expression
     *  @return expression corresponding to substring extraction
     */
    Expression<String> substring(Expression<String> x, Expression<Integer> from);

    /**
     *  Create an expression for substring extraction.
     *  Extracts a substring starting at the specified position
     *  through to end of the string.
     *  First position is 1.
     *  @param x  string expression
     *  @param from  start position
     *  @return expression corresponding to substring extraction
     */
    Expression<String> substring(Expression<String> x, int from);

    /**
     *  Create an expression for substring extraction.
     *  Extracts a substring of given length starting at the
     *  specified position.
     *  First position is 1.
     *  @param x  string expression
     *  @param from  start position expression
     *  @param len  length expression
     *  @return expression corresponding to substring extraction
     */
    Expression<String> substring(Expression<String> x, Expression<Integer> from,
Expression<Integer> len);

    /**
     *  Create an expression for substring extraction.
     *  Extracts a substring of given length starting at the
```

```
     *  specified position.
     *  First position is 1.
     *  @param x  string expression
     *  @param from  start position
     *  @param len  length
     *  @return expression corresponding to substring extraction
     */
    Expression<String> substring(Expression<String> x, int from, int len);

    /**
     *  Used to specify how strings are trimmed.
     */
    public static enum Trimspec {

        /**
         * Trim from leading end.
         */
        LEADING,

        /**
         * Trim from trailing end.
         */
        TRAILING,

        /**
         * Trim from both ends.
         */
        BOTH
    }

    /**
     * Create expression to trim blanks from both ends of
     * a string.
     * @param x  expression for string to trim
     * @return trim expression
     */
    Expression<String> trim(Expression<String> x);

    /**
     * Create expression to trim blanks from a string.
     * @param ts  trim specification
     * @param x  expression for string to trim
     * @return trim expression
     */
    Expression<String> trim(Trimspec ts, Expression<String> x);

    /**
     * Create expression to trim character from both ends of
```

```
     * a string.
     * @param t  expression for character to be trimmed
     * @param x  expression for string to trim
     * @return trim expression
     */
    Expression<String> trim(Expression<Character> t, Expression<String> x);

    /**
     * Create expression to trim character from a string.
     * @param ts  trim specification
     * @param t  expression for character to be trimmed
     * @param x  expression for string to trim
     * @return trim expression
     */
    Expression<String> trim(Trimspec ts, Expression<Character> t, Expression<String> x);

    /**
     * Create expression to trim character from both ends of
     * a string.
     * @param t  character to be trimmed
     * @param x  expression for string to trim
     * @return trim expression
     */
    Expression<String> trim(char t, Expression<String> x);

    /**
     * Create expression to trim character from a string.
     * @param ts  trim specification
     * @param t  character to be trimmed
     * @param x  expression for string to trim
     * @return trim expression
     */
    Expression<String> trim(Trimspec ts, char t, Expression<String> x);

    /**
     * Create expression for converting a string to lowercase.
     * @param x  string expression
     * @return expression to convert to lowercase
     */
    Expression<String> lower(Expression<String> x);

    /**
     * Create expression for converting a string to uppercase.
     * @param x  string expression
     * @return expression to convert to uppercase
     */
    Expression<String> upper(Expression<String> x);
```

```
    /**
     * Create expression to return length of a string.
     * @param x  string expression
     * @return length expression
     */
    Expression<Integer> length(Expression<String> x);


    /**
     * Create expression to locate the position of one string
     * within another, returning position of first character
     * if found.
     * The first position in a string is denoted by 1.  If the
     * string to be located is not found, 0 is returned.
     * @param x  expression for string to be searched
     * @param pattern  expression for string to be located
     * @return expression corresponding to position
     */
    Expression<Integer> locate(Expression<String> x, Expression<String> pattern);

    /**
     * Create expression to locate the position of one string
     * within another, returning position of first character
     * if found.
     * The first position in a string is denoted by 1.  If the
     * string to be located is not found, 0 is returned.
     * @param x  expression for string to be searched
     * @param pattern  string to be located
     * @return expression corresponding to position
     */
    Expression<Integer> locate(Expression<String> x, String pattern);

    /**
     * Create expression to locate the position of one string
     * within another, returning position of first character
     * if found.
     * The first position in a string is denoted by 1.  If the
     * string to be located is not found, 0 is returned.
     * @param x  expression for string to be searched
     * @param pattern  expression for string to be located
     * @param from  expression for position at which to start search
     * @return expression corresponding to position
     */
    Expression<Integer> locate(Expression<String> x, Expression<String> pattern,
Expression<Integer> from);

    /**
     * Create expression to locate the position of one string
```

```
     * within another, returning position of first character
     * if found.
     * The first position in a string is denoted by 1.  If the
     * string to be located is not found, 0 is returned.
     * @param x  expression for string to be searched
     * @param pattern  string to be located
     * @param from  position at which to start search
     * @return expression corresponding to position
     */
    Expression<Integer> locate(Expression<String> x, String pattern, int from);


    // Date/time/timestamp functions:

    /**
     *  Create expression to return current date.
     *  @return expression for current date
     */
    Expression<java.sql.Date> currentDate();

    /**
     *  Create expression to return current timestamp.
     *  @return expression for current timestamp
     */
    Expression<java.sql.Timestamp> currentTimestamp();

    /**
     *  Create expression to return current time.
     *  @return expression for current time
     */
    Expression<java.sql.Time> currentTime();


    //in builders:

    /**
     *  Interface used to build in predicates.
     */
    public static interface In<T> extends Predicate {

         /**
          * Return the expression to be tested against the
          * list of values.
          * @return expression
          */
         Expression<T> getExpression();

         /**
```

```
         *  Add to list of values to be tested against.
         *  @param value value
         *  @return in predicate
         */
        In<T> value(T value);

        /**
         *  Add to list of values to be tested against.
         *  @param value expression
         *  @return in predicate
         */
        In<T> value(Expression<? extends T> value);
    }

    /**
     *  Create predicate to test whether given expression
     *  is contained in a list of values.
     *  @param  expression to be tested against list of values
     *  @return  in predicate
     */
    <T> In<T> in(Expression<? extends T> expression);


    // coalesce, nullif:

    /**
     * Create an expression that returns null if all its arguments
     * evaluate to null, and the value of the first non-null argument
     * otherwise.
     * @param x expression
     * @param y expression
     * @return coalesce expression
     */
    <Y> Expression<Y> coalesce(Expression<? extends Y> x, Expression<? extends Y> y);

    /**
     * Create an expression that returns null if all its arguments
     * evaluate to null, and the value of the first non-null argument
     * otherwise.
     * @param x expression
     * @param y value
     * @return coalesce expression
     */
    <Y> Expression<Y> coalesce(Expression<? extends Y> x, Y y);

    /**
     * Create an expression that tests whether its argument are
     * equal, returning null if they are and the value of the
```

```
     * first expression if they are not.
     * @param x expression
     * @param y expression
     * @return nullif expression
     */
    <Y> Expression<Y> nullif(Expression<Y> x, Expression<?> y);

    /**
     * Create an expression that tests whether its argument are
     * equal, returning null if they are and the value of the
     * first expression if they are not.
     * @param x expression
     * @param y value
     * @return nullif expression
     */
    <Y> Expression<Y> nullif(Expression<Y> x, Y y);


    // coalesce builder:

    /**
     *  Interface used to build coalesce expressions.
     *
     * A coalesce expression is equivalent to a case expression
     * that returns null if all its arguments evaluate to null,
     * and the value of its first non-null argument otherwise.
     */
    public static interface Coalesce<T> extends Expression<T> {

         /**
          * Add an argument to the coalesce expression.
          * @param value  value
          * @return coalesce expression
          */
         Coalesce<T> value(T value);

         /**
          * Add an argument to the coalesce expression.
          * @param value expression
          * @return coalesce expression
          */
         Coalesce<T> value(Expression<? extends T> value);
    }

    /**
     * Create a coalesce expression.
     * @return coalesce expression
     */
```

```
    <T> Coalesce<T> coalesce();


    //case builders:

    /**
     *  Interface used to build simple case expressions.
     *  Case conditions are evaluated in the order in which
     *  they are specified.
     */
    public static interface SimpleCase<C,R> extends Expression<R> {

        /**
         * Return the expression to be tested against the
         * conditions.
         * @return expression
         */
        Expression<C> getExpression();

        /**
         * Add a when/then clause to the case expression.
         * @param condition  "when" condition
         * @param result  "then" result value
         * @return simple case expression
         */
        SimpleCase<C, R> when(C condition, R result);

        /**
         * Add a when/then clause to the case expression.
         * @param condition  "when" condition
         * @param result  "then" result expression
         * @return simple case expression
         */
        SimpleCase<C, R> when(C condition, Expression<? extends R> result);

        /**
         * Add an "else" clause to the case expression.
         * @param result  "else" result
         * @return expression
         */
        Expression<R> otherwise(R result);

        /**
         * Add an "else" clause to the case expression.
         * @param result  "else" result expression
         * @return expression
         */
        Expression<R> otherwise(Expression<? extends R> result);
```

```
    }

    /**
     *  Create a simple case expression.
     *  @param expression  to be tested against the case conditions
     *  @return simple case expression
     */
    <C, R> SimpleCase<C,R> selectCase(Expression<? extends C> expression);


    /**
     *  Interface used to build general case expressions.
     *  Case conditions are evaluated in the order in which
     *  they are specified.
     */
    public static interface Case<R> extends Expression<R> {

        /**
         * Add a when/then clause to the case expression.
         * @param condition  "when" condition
         * @param result  "then" result value
         * @return general case expression
         */
        Case<R> when(Expression<Boolean> condition, R result);

        /**
         * Add a when/then clause to the case expression.
         * @param condition  "when" condition
         * @param result  "then" result expression
         * @return general case expression
         */
        Case<R> when(Expression<Boolean> condition, Expression<? extends R> result);

        /**
         * Add an "else" clause to the case expression.
         * @param result  "else" result
         * @return expression
         */
        Expression<R> otherwise(R result);

        /**
         * Add an "else" clause to the case expression.
         * @param result  "else" result expression
         * @return expression
         */
        Expression<R> otherwise(Expression<? extends R> result);
    }
```

```
    /**
     *  Create a general case expression.
     *  @return general case expression
     */
    <R> Case<R> selectCase();

    /**
     * Create an expression for the execution of a database
     * function.
     * @param name  function name
     * @param type  expected result type
     * @param args  function arguments
     * @return expression
     */
   <T> Expression<T> function(String name, Class<T> type,
Expression<?>... args);


    // methods for downcasting:

    /**
     *  Downcast Join object to the specified type.
     *  @param join  Join object
     *  @param type type to be downcast to
     *  @return  Join object of the specified type
     *  @since 2.1
     */
    <X, T, V extends T> Join<X, V> treat(Join<X, T> join, Class<V> type);

    /**
     *  Downcast CollectionJoin object to the specified type.
     *  @param join  CollectionJoin object
     *  @param type type to be downcast to
     *  @return  CollectionJoin object of the specified type
     *  @since 2.1
     */
    <X, T, E extends T> CollectionJoin<X, E> treat(CollectionJoin<X, T> join, Class<E>
type);

    /**
     *  Downcast SetJoin object to the specified type.
     *  @param join  SetJoin object
     *  @param type type to be downcast to
     *  @return  SetJoin object of the specified type
     *  @since 2.1
     */
    <X, T, E extends T> SetJoin<X, E> treat(SetJoin<X, T> join, Class<E> type);
```

```
    /**
     *  Downcast ListJoin object to the specified type.
     *  @param join  ListJoin object
     *  @param type type to be downcast to
     *  @return  ListJoin object of the specified type
     *  @since 2.1
     */
    <X, T, E extends T> ListJoin<X, E> treat(ListJoin<X, T> join, Class<E> type);

    /**
     *  Downcast MapJoin object to the specified type.
     *  @param join  MapJoin object
     *  @param type type to be downcast to
     *  @return  MapJoin object of the specified type
     *  @since 2.1
     */
    <X, K, T, V extends T> MapJoin<X, K, V> treat(MapJoin<X, K, T> join, Class<V> type);


    /**
     *  Downcast Path object to the specified type.
     *  @param path  path
     *  @param type type to be downcast to
     *  @return  Path object of the specified type
     *  @since 2.1
     */
    <X, T extends X> Path<T> treat(Path<X> path, Class<T> type);

    /**
     *  Downcast Root object to the specified type.
     *  @param root  root
     *  @param type type to be downcast to
     *  @return  Root object of the specified type
     *  @since 2.1
     */
    <X, T extends X> Root<T> treat(Root<X> root, Class<T> type);

}
```

6.3.2. CommonAbstractCriteria Interface

```
package jakarta.persistence.criteria;

/**
 * The <code>CommonAbstractCriteria</code> interface defines functionality
 * that is common to both top-level criteria queries and subqueries as
 * well as to update and delete criteria operations.
 * It is not intended to be used directly in query construction.
 *
 * <p> Note that criteria queries and criteria update and delete operations
 * are typed differently.
 * Criteria queries are typed according to the query result type.
 * Update and delete operations are typed according to the target of the
 * update or delete.
 *
 * @since 2.1
 */
public interface CommonAbstractCriteria {

    /**
     * Create a subquery of the query.
     * @param type  the subquery result type
     * @return subquery
     */
    <U> Subquery<U> subquery(Class<U> type);

    /**
     * Return the predicate that corresponds to the where clause
     * restriction(s), or null if no restrictions have been
     * specified.
     * @return where clause predicate
     */
    Predicate getRestriction();

}
```

6.3.3. AbstractQuery Interface

```
package jakarta.persistence.criteria;

import java.util.List;
import java.util.Set;
import jakarta.persistence.metamodel.EntityType;

/**
 * The <code>AbstractQuery</code> interface defines functionality that is common
```

```
 * to both top-level queries and subqueries.
 * It is not intended to be used directly in query construction.
 *
 * <p> All queries must have:
 *         a set of root entities (which may in turn own joins).
 * <p> All queries may have:
 *         a conjunction of restrictions.
 *
 * @param <T>  the type of the result
 *
 * @since 2.0
 */
public interface AbstractQuery<T> extends CommonAbstractCriteria {

    /**
     * Create and add a query root corresponding to the given entity,
     * forming a cartesian product with any existing roots.
     * @param entityClass  the entity class
     * @return query root corresponding to the given entity
     */
    <X> Root<X> from(Class<X> entityClass);

    /**
     * Create and add a query root corresponding to the given entity,
     * forming a cartesian product with any existing roots.
     * @param entity  metamodel entity representing the entity
     *                of type X
     * @return query root corresponding to the given entity
     */
    <X> Root<X> from(EntityType<X> entity);

    /**
     * Modify the query to restrict the query results according
     * to the specified boolean expression.
     * Replaces the previously added restriction(s), if any.
     * @param restriction  a simple or compound boolean expression
     * @return the modified query
     */
    AbstractQuery<T> where(Expression<Boolean> restriction);

    /**
     * Modify the query to restrict the query results according
     * to the conjunction of the specified restriction predicates.
     * Replaces the previously added restriction(s), if any.
     * If no restrictions are specified, any previously added
     * restrictions are simply removed.
     * @param restrictions  zero or more restriction predicates
     * @return the modified query
```

```
     */
    AbstractQuery<T> where(Predicate... restrictions);

    /**
     * Specify the expressions that are used to form groups over
     * the query results.
     * Replaces the previous specified grouping expressions, if any.
     * If no grouping expressions are specified, any previously
     * added grouping expressions are simply removed.
     * @param grouping  zero or more grouping expressions
     * @return the modified query
     */
    AbstractQuery<T> groupBy(Expression<?>... grouping);

    /**
     * Specify the expressions that are used to form groups over
     * the query results.
     * Replaces the previous specified grouping expressions, if any.
     * If no grouping expressions are specified, any previously
     * added grouping expressions are simply removed.
     * @param grouping  list of zero or more grouping expressions
     * @return the modified query
     */
    AbstractQuery<T> groupBy(List<Expression<?>> grouping);

    /**
     * Specify a restriction over the groups of the query.
     * Replaces the previous having restriction(s), if any.
     * @param restriction  a simple or compound boolean expression
     * @return the modified query
     */
    AbstractQuery<T> having(Expression<Boolean> restriction);

    /**
     * Specify restrictions over the groups of the query
     * according the conjunction of the specified restriction
     * predicates.
     * Replaces the previously having added restriction(s), if any.
     * If no restrictions are specified, any previously added
     * restrictions are simply removed.
     * @param restrictions  zero or more restriction predicates
     * @return the modified query
     */
    AbstractQuery<T> having(Predicate... restrictions);

    /**
     * Specify whether duplicate query results will be eliminated.
     * A true value will cause duplicates to be eliminated.
```

```
     * A false value will cause duplicates to be retained.
     * If distinct has not been specified, duplicate results must
     * be retained.
     * @param distinct  boolean value specifying whether duplicate
     *        results must be eliminated from the query result or
     *        whether they must be retained
     * @return the modified query
     */
    AbstractQuery<T> distinct(boolean distinct);

    /**
     * Return the query roots.  These are the roots that have
     * been defined for the <code>CriteriaQuery</code> or <code>Subquery</code> itself,
     * including any subquery roots defined as a result of
     * correlation. Returns empty set if no roots have been defined.
     * Modifications to the set do not affect the query.
     * @return the set of query roots
     */
    Set<Root<?>> getRoots();

    /**
     *  Return the selection of the query, or null if no selection
     *  has been set.
     *  @return selection item
     */
    Selection<T> getSelection();

    /**
     * Return a list of the grouping expressions.  Returns empty
     * list if no grouping expressions have been specified.
     * Modifications to the list do not affect the query.
     * @return the list of grouping expressions
     */
    List<Expression<?>> getGroupList();

    /**
     * Return the predicate that corresponds to the restriction(s)
     * over the grouping items, or null if no restrictions have
     * been specified.
     * @return having clause predicate
     */
    Predicate getGroupRestriction();

    /**
     * Return whether duplicate query results must be eliminated or
     * retained.
     * @return boolean indicating whether duplicate query results
     *         must be eliminated
```

```
     */
    boolean isDistinct();

    /**
     * Return the result type of the query or subquery.  If a result
     * type was specified as an argument to the
     * <code>createQuery</code> or <code>subquery</code> method, that
     * type will be returned.  If the query was created using the
     * <code>createTupleQuery</code> method, the result type is
     * <code>Tuple</code>.  Otherwise, the result type is
     * <code>Object</code>.
     * @return result type
     */
    Class<T> getResultType();
}
```

6.3.4. CriteriaQuery Interface

```
package jakarta.persistence.criteria;

import java.util.List;
import java.util.Set;

/**
 * The <code>CriteriaQuery</code> interface defines functionality that is specific
 * to top-level queries.
 *
 * @param <T>  the type of the defined result
 *
 * @since 2.0
 */
public interface CriteriaQuery<T> extends AbstractQuery<T> {

    /**
     * Specify the item that is to be returned in the query result.
     * Replaces the previously specified selection(s), if any.
     *
     * <p> Note: Applications using the string-based API may need to
     * specify the type of the select item when it results from
     * a get or join operation and the query result type is
     * specified.
     *
     * <pre>
     *     For example:
     *
     *     CriteriaQuery&#060;String&#062; q = cb.createQuery(String.class);
```

```
     *     Root&#060;Order&#062; order = q.from(Order.class);
     *     q.select(order.get("shippingAddress").&#060;String&#062;get("state"));
     *
     *     CriteriaQuery&#060;Product&#062; q2 = cb.createQuery(Product.class);
     *     q2.select(q2.from(Order.class)
     *                 .join("items")
     *                 .&#060;Item,Product&#062;join("product"));
     *
     * </pre>
     * @param selection  selection specifying the item that
     *        is to be returned in the query result
     * @return the modified query
     * @throws IllegalArgumentException if the selection is
     *         a compound selection and more than one selection
     *         item has the same assigned alias
     */
    CriteriaQuery<T> select(Selection<? extends T> selection);

    /**
     * Specify the selection items that are to be returned in the
     * query result.
     * Replaces the previously specified selection(s), if any.
     *
     * The type of the result of the query execution depends on
     * the specification of the type of the criteria query object
     * created as well as the arguments to the multiselect method.
     * <p> An argument to the multiselect method must not be a tuple-
     * or array-valued compound selection item.
     *
     * <p>The semantics of this method are as follows:
     * <ul>
     * <li>
     * If the type of the criteria query is
     * <code>CriteriaQuery&#060;Tuple&#062;</code> (i.e., a criteria
     * query object created by either the
     * <code>createTupleQuery</code> method or by passing a
     * <code>Tuple</code> class argument to the
     * <code>createQuery</code> method), a <code>Tuple</code> object
     * corresponding to the arguments of the <code>multiselect</code>
     * method, in the specified order, will be instantiated and
     * returned for each row that results from the query execution.
     *
     * <li> If the type of the criteria query is <code>CriteriaQuery&#060;X&#062;</code>
for
     * some user-defined class X (i.e., a criteria query object
     * created by passing a X class argument to the <code>createQuery</code>
     * method), the arguments to the <code>multiselect</code> method will be
     * passed to the X constructor and an instance of type X will be
```

```
     * returned for each row.
     *
     * <li> If the type of the criteria query is
<code>CriteriaQuery&#060;X[]&#062;</code> for
     * some class X, an instance of type X[] will be returned for
     * each row.   The elements of the array will correspond to the
     * arguments of the <code>multiselect</code> method, in the
     * specified order.
     *
     * <li> If the type of the criteria query is
<code>CriteriaQuery&#060;Object&#062;</code>
     * or if the criteria query was created without specifying a
     * type, and only a single argument is passed to the <code>multiselect</code>
     * method, an instance of type <code>Object</code> will be returned for
     * each row.
     *
     * <li> If the type of the criteria query is
<code>CriteriaQuery&#060;Object&#062;</code>
     * or if the criteria query was created without specifying a
     * type, and more than one argument is passed to the <code>multiselect</code>
     * method, an instance of type <code>Object[]</code> will be instantiated
     * and returned for each row.  The elements of the array will
     * correspond to the arguments to the <code> multiselect</code>  method,
     * in the specified order.
     * </ul>
     *
     * @param selections  selection items corresponding to the
     *        results to be returned by the query
     * @return the modified query
     * @throws IllegalArgumentException if a selection item is
     *         not valid or if more than one selection item has
     *         the same assigned alias
     */
    CriteriaQuery<T> multiselect(Selection<?>... selections);


    /**
     * Specify the selection items that are to be returned in the
     * query result.
     * Replaces the previously specified selection(s), if any.
     *
     * <p> The type of the result of the query execution depends on
     * the specification of the type of the criteria query object
     * created as well as the argument to the <code>multiselect</code> method.
     * An element of the list passed to the <code>multiselect</code> method
     * must not be a tuple- or array-valued compound selection item.
     *
     * <p> The semantics of this method are as follows:
```

```
     * <ul>
     * <li> If the type of the criteria query is
<code>CriteriaQuery&#060;Tuple&#062;</code>
     * (i.e., a criteria query object created by either the
     * <code>createTupleQuery</code> method or by passing a <code>Tuple</code> class
argument
     * to the <code>createQuery</code> method), a <code>Tuple</code> object corresponding
to
     * the elements of the list passed to the <code>multiselect</code> method,
     * in the specified order, will be instantiated and returned for each
     * row that results from the query execution.
     *
     * <li> If the type of the criteria query is <code>CriteriaQuery&#060;X&#062;</code>
for
     * some user-defined class X (i.e., a criteria query object
     * created by passing a X class argument to the <code>createQuery</code>
     * method), the elements of the list passed to the <code>multiselect</code>
     * method will be passed to the X constructor and an instance
     * of type X will be returned for each row.
     *
     * <li> If the type of the criteria query is
<code>CriteriaQuery&#060;X[]&#062;</code> for
     * some class X, an instance of type X[] will be returned for
     * each row.   The elements of the array will correspond to the
     * elements of the list passed to the <code>multiselect</code> method,
     * in the specified order.
     *
     * <li> If the type of the criteria query is
<code>CriteriaQuery&#060;Object&#062;</code>
     * or if the criteria query was created without specifying a
     * type, and the list passed to the <code>multiselect</code> method contains
     * only a single element, an instance of type <code>Object</code> will be
     * returned for each row.
     *
     * <li> If the type of the criteria query is
<code>CriteriaQuery&#060;Object&#062;</code>
     * or if the criteria query was created without specifying a
     * type, and the list passed to the <code>multiselect</code> method contains
     * more than one element, an instance of type <code>Object[]</code> will be
     * instantiated and returned for each row.  The elements of the
     * array will correspond to the elements of the list passed to
     * the <code>multiselect</code> method, in the specified order.
     * </ul>
     *
     * @param selectionList  list of selection items corresponding
     *        to the results to be returned by the query
     * @return the modified query
     * @throws IllegalArgumentException if a selection item is
```

```
     *         not valid or if more than one selection item has
     *         the same assigned alias
     */
    CriteriaQuery<T> multiselect(List<Selection<?>> selectionList);

    /**
     * Modify the query to restrict the query result according
     * to the specified boolean expression.
     * Replaces the previously added restriction(s), if any.
     * This method only overrides the return type of the
     * corresponding <code>AbstractQuery</code> method.
     * @param restriction  a simple or compound boolean expression
     * @return the modified query
     */
    CriteriaQuery<T> where(Expression<Boolean> restriction);

    /**
     * Modify the query to restrict the query result according
     * to the conjunction of the specified restriction predicates.
     * Replaces the previously added restriction(s), if any.
     * If no restrictions are specified, any previously added
     * restrictions are simply removed.
     * This method only overrides the return type of the
     * corresponding <code>AbstractQuery</code> method.
     * @param restrictions  zero or more restriction predicates
     * @return the modified query
     */
    CriteriaQuery<T> where(Predicate... restrictions);

    /**
     * Specify the expressions that are used to form groups over
     * the query results.
     * Replaces the previous specified grouping expressions, if any.
     * If no grouping expressions are specified, any previously
     * added grouping expressions are simply removed.
     * This method only overrides the return type of the
     * corresponding <code>AbstractQuery</code> method.
     * @param grouping  zero or more grouping expressions
     * @return the modified query
     */
    CriteriaQuery<T> groupBy(Expression<?>... grouping);

    /**
     * Specify the expressions that are used to form groups over
     * the query results.
     * Replaces the previous specified grouping expressions, if any.
     * If no grouping expressions are specified, any previously
     * added grouping expressions are simply removed.
```

```
     * This method only overrides the return type of the
     * corresponding <code>AbstractQuery</code> method.
     * @param grouping  list of zero or more grouping expressions
     * @return the modified query
     */
    CriteriaQuery<T> groupBy(List<Expression<?>> grouping);

    /**
     * Specify a restriction over the groups of the query.
     * Replaces the previous having restriction(s), if any.
     * This method only overrides the return type of the
     * corresponding <code>AbstractQuery</code> method.
     * @param restriction  a simple or compound boolean expression
     * @return the modified query
     */
    CriteriaQuery<T> having(Expression<Boolean> restriction);

    /**
     * Specify restrictions over the groups of the query
     * according the conjunction of the specified restriction
     * predicates.
     * Replaces the previously added having restriction(s), if any.
     * If no restrictions are specified, any previously added
     * restrictions are simply removed.
     * This method only overrides the return type of the
     * corresponding <code>AbstractQuery</code> method.
     * @param restrictions  zero or more restriction predicates
     * @return the modified query
     */
    CriteriaQuery<T> having(Predicate... restrictions);

    /**
     * Specify the ordering expressions that are used to
     * order the query results.
     * Replaces the previous ordering expressions, if any.
     * If no ordering expressions are specified, the previous
     * ordering, if any, is simply removed, and results will
     * be returned in no particular order.
     * The left-to-right sequence of the ordering expressions
     * determines the precedence, whereby the leftmost has highest
     * precedence.
     * @param o  zero or more ordering expressions
     * @return the modified query
     */
    CriteriaQuery<T> orderBy(Order... o);

    /**
     * Specify the ordering expressions that are used to
```

```
     * order the query results.
     * Replaces the previous ordering expressions, if any.
     * If no ordering expressions are specified, the previous
     * ordering, if any, is simply removed, and results will
     * be returned in no particular order.
     * The order of the ordering expressions in the list
     * determines the precedence, whereby the first element in the
     * list has highest precedence.
     * @param o  list of zero or more ordering expressions
     * @return the modified query
     */
    CriteriaQuery<T> orderBy(List<Order> o);

    /**
     * Specify whether duplicate query results will be eliminated.
     * A true value will cause duplicates to be eliminated.
     * A false value will cause duplicates to be retained.
     * If distinct has not been specified, duplicate results must
     * be retained.
     * This method only overrides the return type of the
     * corresponding <code>AbstractQuery</code> method.
     * @param distinct  boolean value specifying whether duplicate
     *        results must be eliminated from the query result or
     *        whether they must be retained
     * @return the modified query.
     */
    CriteriaQuery<T> distinct(boolean distinct);

    /**
     * Return the ordering expressions in order of precedence.
     * Returns empty list if no ordering expressions have been
     * specified.
     * Modifications to the list do not affect the query.
     * @return the list of ordering expressions
     */
    List<Order> getOrderList();

    /**
     * Return the parameters of the query.  Returns empty set if
     * there are no parameters.
     * Modifications to the set do not affect the query.
     * @return the query parameters
     */
    Set<ParameterExpression<?>> getParameters();
}
```

6.3.5. CriteriaUpdate Interface

```
package jakarta.persistence.criteria;

import jakarta.persistence.metamodel.SingularAttribute;
import jakarta.persistence.metamodel.EntityType;

/**
 * The <code>CriteriaUpdate</code> interface defines functionality for performing
 * bulk update operations using the Criteria API.
 *
 * <p>Criteria API bulk update operations map directly to database update
 * operations, bypassing any optimistic locking checks.  Portable
 * applications using bulk update operations must manually update the
 * value of the version column, if desired, and/or manually validate
 * the value of the version column.
 * The persistence context is not synchronized with the result of the
 * bulk update.
 *
 * <p> A <code>CriteriaUpdate</code> object must have a single root.
 *
 * @param <T>  the entity type that is the target of the update
 *
 * @since 2.1
 */
public interface CriteriaUpdate<T> extends CommonAbstractCriteria {

   /**
    * Create and add a query root corresponding to the entity
    * that is the target of the update.
    * A <code>CriteriaUpdate</code> object has a single root, the entity that
    * is being updated.
    * @param entityClass  the entity class
    * @return query root corresponding to the given entity
    */
   Root<T> from(Class<T> entityClass);

   /**
    * Create and add a query root corresponding to the entity
    * that is the target of the update.
    * A <code>CriteriaUpdate</code> object has a single root, the entity that
    * is being updated.
    * @param entity  metamodel entity representing the entity
    *                of type X
    * @return query root corresponding to the given entity
    */
   Root<T> from(EntityType<T> entity);
```

```
    /**
     * Return the query root.
     * @return the query root
     */
    Root<T> getRoot();

    /**
     * Update the value of the specified attribute.
     * @param attribute  attribute to be updated
     * @param value  new value
     * @return  the modified update query
     */
    <Y, X extends Y> CriteriaUpdate<T> set(SingularAttribute<? super T, Y> attribute, X
value);

    /**
     * Update the value of the specified attribute.
     * @param attribute  attribute to be updated
     * @param value  new value
     * @return  the modified update query
     */
    <Y> CriteriaUpdate<T> set(SingularAttribute<? super T, Y> attribute, Expression<?
extends Y> value);

    /**
     * Update the value of the specified attribute.
     * @param attribute  attribute to be updated
     * @param value  new value
     * @return  the modified update query
     */
    <Y, X extends Y> CriteriaUpdate<T> set(Path<Y> attribute, X value);

    /**
     * Update the value of the specified attribute.
     * @param attribute  attribute to be updated
     * @param value  new value
     * @return  the modified update query
     */
    <Y> CriteriaUpdate<T> set(Path<Y> attribute, Expression<? extends Y> value);

    /**
     * Update the value of the specified attribute.
     * @param attributeName  name of the attribute to be updated
     * @param value  new value
     * @return  the modified update query
     */
    CriteriaUpdate<T> set(String attributeName, Object value);
```

```
    /**
     * Modify the update query to restrict the target of the update
     * according to the specified boolean expression.
     * Replaces the previously added restriction(s), if any.
     * @param restriction  a simple or compound boolean expression
     * @return the modified update query
     */
   CriteriaUpdate<T> where(Expression<Boolean> restriction);

    /**
     * Modify the update query to restrict the target of the update
     * according to the conjunction of the specified restriction
     * predicates.
     * Replaces the previously added restriction(s), if any.
     * If no restrictions are specified, any previously added
     * restrictions are simply removed.
     * @param restrictions  zero or more restriction predicates
     * @return the modified update query
     */
   CriteriaUpdate<T> where(Predicate... restrictions);
}
```

6.3.6. CriteriaDelete Interface

```
package jakarta.persistence.criteria;

import jakarta.persistence.metamodel.EntityType;

/**
 * The <code>CriteriaDelete</code> interface defines functionality for performing
 * bulk delete operations using the Criteria API
 *
 * <p>Criteria API bulk delete operations map directly to database
 * delete operations.  The persistence context is not synchronized
 * with the result of the bulk delete.
 *
 * <p> A <code>CriteriaDelete</code> object must have a single root.
 *
 * @param <T>  the entity type that is the target of the delete
 *
 * @since 2.1
 */
public interface CriteriaDelete<T> extends CommonAbstractCriteria {
```

```
    /**
     * Create and add a query root corresponding to the entity
     * that is the target of the delete.
     * A <code>CriteriaDelete</code> object has a single root, the entity that
     * is being deleted.
     * @param entityClass  the entity class
     * @return query root corresponding to the given entity
     */
    Root<T> from(Class<T> entityClass);

    /**
     * Create and add a query root corresponding to the entity
     * that is the target of the delete.
     * A <code>CriteriaDelete</code> object has a single root, the entity that
     * is being deleted.
     * @param entity  metamodel entity representing the entity
     *                of type X
     * @return query root corresponding to the given entity
     */
    Root<T> from(EntityType<T> entity);

   /**
    * Return the query root.
    * @return the query root
    */
   Root<T> getRoot();

    /**
     * Modify the delete query to restrict the target of the deletion
     * according to the specified boolean expression.
     * Replaces the previously added restriction(s), if any.
     * @param restriction  a simple or compound boolean expression
     * @return the modified delete query
     */
   CriteriaDelete<T> where(Expression<Boolean> restriction);

    /**
     * Modify the delete query to restrict the target of the deletion
     * according to the conjunction of the specified restriction
     * predicates.
     * Replaces the previously added restriction(s), if any.
     * If no restrictions are specified, any previously added
     * restrictions are simply removed.
     * @param restrictions  zero or more restriction predicates
     * @return the modified delete query
     */
   CriteriaDelete<T> where(Predicate... restrictions);
```

```
}
```

6.3.7. Subquery Interface

```
package jakarta.persistence.criteria;

import java.util.List;
import java.util.Set;

/**
 * The <code>Subquery</code> interface defines functionality that is
 * specific to subqueries.
 *
 * A subquery has an expression as its selection item.
 *
 * @param <T> the type of the selection item.
 *
 * @since 2.0
 */
public interface Subquery<T> extends AbstractQuery<T>, Expression<T> {

    /**
     * Specify the item that is to be returned as the subquery
     * result.
     * Replaces the previously specified selection, if any.
     * @param expression  expression specifying the item that
     *        is to be returned as the subquery result
     * @return the modified subquery
     */
    Subquery<T> select(Expression<T> expression);

    /**
     * Modify the subquery to restrict the result according
     * to the specified boolean expression.
     * Replaces the previously added restriction(s), if any.
     * This method only overrides the return type of the
     * corresponding <code>AbstractQuery</code> method.
     * @param restriction  a simple or compound boolean expression
     * @return the modified subquery
     */
    Subquery<T> where(Expression<Boolean> restriction);

    /**
     * Modify the subquery to restrict the result according
     * to the conjunction of the specified restriction predicates.
     * Replaces the previously added restriction(s), if any.
```

```
     * If no restrictions are specified, any previously added
     * restrictions are simply removed.
     * This method only overrides the return type of the
     * corresponding <code>AbstractQuery</code> method.
     * @param restrictions  zero or more restriction predicates
     * @return the modified subquery
     */
    Subquery<T> where(Predicate... restrictions);

    /**
     * Specify the expressions that are used to form groups over
     * the subquery results.
     * Replaces the previous specified grouping expressions, if any.
     * If no grouping expressions are specified, any previously
     * added grouping expressions are simply removed.
     * This method only overrides the return type of the
     * corresponding <code>AbstractQuery</code> method.
     * @param grouping  zero or more grouping expressions
     * @return the modified subquery
     */
    Subquery<T> groupBy(Expression<?>... grouping);

    /**
     * Specify the expressions that are used to form groups over
     * the subquery results.
     * Replaces the previous specified grouping expressions, if any.
     * If no grouping expressions are specified, any previously
     * added grouping expressions are simply removed.
     * This method only overrides the return type of the
     * corresponding <code>AbstractQuery</code> method.
     * @param grouping  list of zero or more grouping expressions
     * @return the modified subquery
     */
    Subquery<T> groupBy(List<Expression<?>> grouping);

    /**
     * Specify a restriction over the groups of the subquery.
     * Replaces the previous having restriction(s), if any.
     * This method only overrides the return type of the
     * corresponding <code>AbstractQuery</code> method.
     * @param restriction  a simple or compound boolean expression
     * @return the modified subquery
     */
    Subquery<T> having(Expression<Boolean> restriction);

    /**
     * Specify restrictions over the groups of the subquery
     * according the conjunction of the specified restriction
```

```
     * predicates.
     * Replaces the previously added having restriction(s), if any.
     * If no restrictions are specified, any previously added
     * restrictions are simply removed.
     * This method only overrides the return type of the
     * corresponding <code>AbstractQuery</code> method.
     * @param restrictions  zero or more restriction predicates
     * @return the modified subquery
     */
    Subquery<T> having(Predicate... restrictions);

    /**
     * Specify whether duplicate query results will be eliminated.
     * A true value will cause duplicates to be eliminated.
     * A false value will cause duplicates to be retained.
     * If distinct has not been specified, duplicate results must
     * be retained.
     * This method only overrides the return type of the
     * corresponding <code>AbstractQuery</code> method.
     * @param distinct  boolean value specifying whether duplicate
     *        results must be eliminated from the subquery result or
     *        whether they must be retained
     * @return the modified subquery.
     */
    Subquery<T> distinct(boolean distinct);

    /**
     * Create a subquery root correlated to a root of the
     * enclosing query.
     * @param parentRoot  a root of the containing query
     * @return subquery root
     */
    <Y> Root<Y> correlate(Root<Y> parentRoot);

    /**
     * Create a subquery join object correlated to a join object
     * of the enclosing query.
     * @param parentJoin  join object of the containing query
     * @return subquery join
     */
    <X, Y> Join<X, Y> correlate(Join<X, Y> parentJoin);

    /**
     * Create a subquery collection join object correlated to a
     * collection join object of the enclosing query.
     * @param parentCollection  join object of the containing query
     * @return subquery join
     */
```

```
    <X, Y> CollectionJoin<X, Y> correlate(CollectionJoin<X, Y> parentCollection);

    /**
     * Create a subquery set join object correlated to a set join
     * object of the enclosing query.
     * @param parentSet  join object of the containing query
     * @return subquery join
     */
    <X, Y> SetJoin<X, Y> correlate(SetJoin<X, Y> parentSet);

    /**
     * Create a subquery list join object correlated to a list join
     * object of the enclosing query.
     * @param parentList join object of the containing query
     * @return subquery join
     */
    <X, Y> ListJoin<X, Y> correlate(ListJoin<X, Y> parentList);

    /**
     * Create a subquery map join object correlated to a map join
     * object of the enclosing query.
     * @param parentMap join object of the containing query
     * @return subquery join
     */
    <X, K, V> MapJoin<X, K, V> correlate(MapJoin<X, K, V> parentMap);

    /**
     * Return the query of which this is a subquery.
     * This must be a CriteriaQuery or a Subquery.
     * @return the enclosing query or subquery
     */
    AbstractQuery<?> getParent();

    /**
     * Return the query of which this is a subquery.
     * This may be a CriteriaQuery, CriteriaUpdate, CriteriaDelete,
     * or a Subquery.
     * @return the enclosing query or subquery
     * @since 2.1
     */
    CommonAbstractCriteria getContainingQuery();

    /**
     * Return the selection expression.
     * @return the item to be returned in the subquery result
     */
    Expression<T> getSelection();
```

```
    /**
     *  Return the correlated joins of the subquery.
     *  Returns empty set if the subquery has no correlated
     *  joins.
     *  Modifications to the set do not affect the query.
     *  @return the correlated joins of the subquery
     */
    Set<Join<?, ?>> getCorrelatedJoins();

}
```

6.3.8. Selection Interface

```
package jakarta.persistence.criteria;

import jakarta.persistence.TupleElement;
import java.util.List;

/**
 * The <code>Selection</code> interface defines an item that is to be
 * returned in a query result.
 *
 * @param <X> the type of the selection item
 *
 * @since 2.0
 */
public interface Selection<X> extends TupleElement<X> {

    /**
     * Assigns an alias to the selection item.
     * Once assigned, an alias cannot be changed or reassigned.
     * Returns the same selection item.
     * @param name  alias
     * @return selection item
     */
    Selection<X> alias(String name);

    /**
     * Whether the selection item is a compound selection.
     * @return boolean indicating whether the selection is a compound
     *         selection
     */
    boolean isCompoundSelection();

    /**
     * Return the selection items composing a compound selection.
     * Modifications to the list do not affect the query.
     * @return list of selection items
     * @throws IllegalStateException if selection is not a
     *         compound selection
     */
    List<Selection<?>> getCompoundSelectionItems();
}
```

6.3.9. CompoundSelection Interface

```
package jakarta.persistence.criteria;

/**
 * The <code>CompoundSelection</code> interface defines a compound selection item
 * (tuple, array, or result of constructor).
 *
 * @param <X> the type of the selection item
 *
 * @since 2.0
 */
public interface CompoundSelection<X> extends Selection<X> {}
```

6.3.10. Expression Interface

```
package jakarta.persistence.criteria;

import java.util.Collection;

/**
 * Type for query expressions.
 *
 * @param <T> the type of the expression
 *
 * @since 2.0
 */
public interface Expression<T> extends Selection<T> {

    /**
     *  Create a predicate to test whether the expression is null.
     *  @return predicate testing whether the expression is null
     */
    Predicate isNull();

    /**
     *  Create a predicate to test whether the expression is
     *  not null.
     *  @return predicate testing whether the expression is not null
     */
    Predicate isNotNull();

    /**
     * Create a predicate to test whether the expression is a member
     * of the argument list.
     * @param values  values to be tested against
     * @return predicate testing for membership
```

```
     */
    Predicate in(Object... values);

    /**
     * Create a predicate to test whether the expression is a member
     * of the argument list.
     * @param values  expressions to be tested against
     * @return predicate testing for membership
     */
    Predicate in(Expression<?>... values);

    /**
     * Create a predicate to test whether the expression is a member
     * of the collection.
     * @param values  collection of values to be tested against
     * @return predicate testing for membership
     */
    Predicate in(Collection<?> values);

    /**
     * Create a predicate to test whether the expression is a member
     * of the collection.
     * @param values expression corresponding to collection to be
     *        tested against
     * @return predicate testing for membership
     */
    Predicate in(Expression<Collection<?>> values);

    /**
     * Perform a typecast upon the expression, returning a new
     * expression object.
     * This method does not cause type conversion:
     * the runtime type is not changed.
     * Warning: may result in a runtime failure.
     * @param type  intended type of the expression
     * @return new expression of the given type
     */
    <X> Expression<X> as(Class<X> type);
}
```

6.3.11. Predicate Interface

```
package jakarta.persistence.criteria;

import java.util.List;
```

```
/**
 * The type of a simple or compound predicate: a conjunction or
 * disjunction of restrictions.
 * A simple predicate is considered to be a conjunction with a
 * single conjunct.
 *
 * @since 2.0
 */
public interface Predicate extends Expression<Boolean> {

        public static enum BooleanOperator {
              AND, OR
        }

    /**
     * Return the boolean operator for the predicate.
     * If the predicate is simple, this is <code>AND</code>.
     * @return boolean operator for the predicate
     */
    BooleanOperator getOperator();

    /**
     * Whether the predicate has been created from another
     * predicate by applying the <code>Predicate.not()</code> method
     * or the <code>CriteriaBuilder.not()</code> method.
     * @return boolean indicating if the predicate is
     *                 a negated predicate
     */
    boolean isNegated();

    /**
     * Return the top-level conjuncts or disjuncts of the predicate.
     * Returns empty list if there are no top-level conjuncts or
     * disjuncts of the predicate.
     * Modifications to the list do not affect the query.
     * @return list of boolean expressions forming the predicate
     */
    List<Expression<Boolean>> getExpressions();

    /**
     * Create a negation of the predicate.
     * @return negated predicate
     */
    Predicate not();

}
```

6.3.12. Path Interface

```
package jakarta.persistence.criteria;

import jakarta.persistence.metamodel.PluralAttribute;
import jakarta.persistence.metamodel.SingularAttribute;
import jakarta.persistence.metamodel.Bindable;
import jakarta.persistence.metamodel.MapAttribute;

/**
 * Represents a simple or compound attribute path from a
 * bound type or collection, and is a "primitive" expression.
 *
 * @param <X>  the type referenced by the path
 *
 * @since 2.0
 */
public interface Path<X> extends Expression<X> {

    /**
     * Return the bindable object that corresponds to the
     * path expression.
     * @return bindable object corresponding to the path
     */
    Bindable<X> getModel();

    /**
     *  Return the parent "node" in the path or null if no parent.
     *  @return parent
     */
    Path<?> getParentPath();

    /**
     *  Create a path corresponding to the referenced
     *  single-valued attribute.
     *  @param attribute single-valued attribute
     *  @return path corresponding to the referenced attribute
     */
    <Y> Path<Y> get(SingularAttribute<? super X, Y> attribute);

    /**
     *  Create a path corresponding to the referenced
     *  collection-valued attribute.
     *  @param collection collection-valued attribute
     *  @return expression corresponding to the referenced attribute
     */
    <E, C extends java.util.Collection<E>> Expression<C> get(PluralAttribute<X, C, E>
```

```
collection);

    /**
     *  Create a path corresponding to the referenced
     *  map-valued attribute.
     *  @param map map-valued attribute
     *  @return expression corresponding to the referenced attribute
     */
    <K, V, M extends java.util.Map<K, V>> Expression<M> get(MapAttribute<X, K, V> map);

    /**
     *  Create an expression corresponding to the type of the path.
     *  @return expression corresponding to the type of the path
     */
    Expression<Class<? extends X>> type();


    //String-based:

    /**
     *  Create a path corresponding to the referenced attribute.
     *
     *  <p> Note: Applications using the string-based API may need to
     *  specify the type resulting from the <code>get</code> operation in order
     *  to avoid the use of <code>Path</code> variables.
     *
     *  <pre>
     *     For example:
     *
     *     CriteriaQuery&#060;Person&#062; q = cb.createQuery(Person.class);
     *     Root&#060;Person&#062; p = q.from(Person.class);
     *     q.select(p)
     *      .where(cb.isMember("joe",
     *                         p.&#060;Set&#060;String&#062;&#062;get("nicknames")));
     *
     *     rather than:
     *
     *     CriteriaQuery&#060;Person&#062; q = cb.createQuery(Person.class);
     *     Root&#060;Person&#062; p = q.from(Person.class);
     *     Path&#060;Set&#060;String&#062;&#062; nicknames = p.get("nicknames");
     *     q.select(p)
     *      .where(cb.isMember("joe", nicknames));
     *  </pre>
     *
     *  @param attributeName  name of the attribute
     *  @return path corresponding to the referenced attribute
     *  @throws IllegalStateException if invoked on a path that
     *          corresponds to a basic type
```

```
     *  @throws IllegalArgumentException if attribute of the given
     *          name does not otherwise exist
     */
    <Y> Path<Y> get(String attributeName);
}
```

6.3.13. FetchParent Interface

```
package jakarta.persistence.criteria;

import jakarta.persistence.metamodel.PluralAttribute;
import jakarta.persistence.metamodel.SingularAttribute;

/**
 * Represents an element of the from clause which may
 * function as the parent of Fetches.
 *
 * @param <Z>  the source type
 * @param <X>  the target type
 *
 * @since 2.0
 */
public interface FetchParent<Z, X> {

    /**
     *  Return the fetch joins that have been made from this type.
     *  Returns empty set if no fetch joins have been made from
     *  this type.
     *  Modifications to the set do not affect the query.
     *  @return fetch joins made from this type
     */
    java.util.Set<Fetch<X, ?>> getFetches();

    /**
     *  Create a fetch join to the specified single-valued attribute
     *  using an inner join.
     *  @param attribute  target of the join
     *  @return the resulting fetch join
     */
    <Y> Fetch<X, Y> fetch(SingularAttribute<? super X, Y> attribute);

    /**
     *  Create a fetch join to the specified single-valued attribute
     *  using the given join type.
     *  @param attribute  target of the join
     *  @param jt  join type
```

```
     *  @return the resulting fetch join
     */
    <Y> Fetch<X, Y> fetch(SingularAttribute<? super X, Y> attribute, JoinType jt);

    /**
     *  Create a fetch join to the specified collection-valued
     *  attribute using an inner join.
     *  @param attribute  target of the join
     *  @return the resulting join
     */
    <Y> Fetch<X, Y> fetch(PluralAttribute<? super X, ?, Y> attribute);

    /**
     *  Create a fetch join to the specified collection-valued
     *  attribute using the given join type.
     *  @param attribute  target of the join
     *  @param jt  join type
     *  @return the resulting join
     */
    <Y> Fetch<X, Y> fetch(PluralAttribute<? super X, ?, Y> attribute, JoinType jt);


    //String-based:

    /**
     *  Create a fetch join to the specified attribute using an
     *  inner join.
     *  @param attributeName  name of the attribute for the
     *         target of the join
     *  @return the resulting fetch join
     *  @throws IllegalArgumentException if attribute of the given
     *          name does not exist
     */
    @SuppressWarnings("hiding")
    <X, Y> Fetch<X, Y> fetch(String attributeName);

    /**
     *  Create a fetch join to the specified attribute using
     *  the given join type.
     *  @param attributeName  name of the attribute for the
     *               target of the join
     *  @param jt  join type
     *  @return the resulting fetch join
     *  @throws IllegalArgumentException if attribute of the given
     *          name does not exist
     */
    @SuppressWarnings("hiding")
    <X, Y> Fetch<X, Y> fetch(String attributeName, JoinType jt);
```

```
}
```

6.3.14. Fetch Interface

```
package jakarta.persistence.criteria;

import jakarta.persistence.metamodel.Attribute;

/**
 * Represents a join-fetched association or attribute.
 *
 * @param <Z>  the source type of the fetch
 * @param <X>  the target type of the fetch
 *
 * @since 2.0
 */
public interface Fetch<Z, X> extends FetchParent<Z, X> {

    /**
     * Return the metamodel attribute corresponding to the
     * fetch join.
     * @return metamodel attribute for the join
     */
    Attribute<? super Z, ?> getAttribute();

    /**
     * Return the parent of the fetched item.
     * @return fetch parent
     */
    FetchParent<?, Z> getParent();

    /**
     * Return the join type used in the fetch join.
     * @return join type
     */
    JoinType getJoinType();
}
```

6.3.15. From Interface

```
package jakarta.persistence.criteria;

import jakarta.persistence.metamodel.SingularAttribute;
import jakarta.persistence.metamodel.CollectionAttribute;
import jakarta.persistence.metamodel.ListAttribute;
```

```
import jakarta.persistence.metamodel.MapAttribute;
import jakarta.persistence.metamodel.SetAttribute;
import java.util.Set;

/**
 * Represents a bound type, usually an entity that appears in
 * the from clause, but may also be an embeddable belonging to
 * an entity in the from clause.
 * <p> Serves as a factory for Joins of associations, embeddables, and
 * collections belonging to the type, and for Paths of attributes
 * belonging to the type.
 *
 * @param <Z>  the source type
 * @param <X>  the target type
 *
 * @since 2.0
 */
@SuppressWarnings("hiding")
public interface From<Z, X> extends Path<X>, FetchParent<Z, X> {

    /**
     *  Return the joins that have been made from this bound type.
     *  Returns empty set if no joins have been made from this
     *  bound type.
     *  Modifications to the set do not affect the query.
     *  @return joins made from this type
     */
    Set<Join<X, ?>> getJoins();

    /**
     *  Whether the <code>From</code> object has been obtained as a result of
     *  correlation (use of a <code>Subquery</code> <code>correlate</code>
     *  method).
     *  @return boolean indicating whether the object has been
     *          obtained through correlation
     */
    boolean isCorrelated();

    /**
     *  Returns the parent <code>From</code> object from which the correlated
     *  <code>From</code> object has been obtained through correlation (use
     *  of a <code>Subquery</code> <code>correlate</code> method).
     *  @return  the parent of the correlated From object
     *  @throws IllegalStateException if the From object has
     *          not been obtained through correlation
     */
    From<Z, X> getCorrelationParent();
```

```
    /**
     *  Create an inner join to the specified single-valued
     *  attribute.
     *  @param attribute  target of the join
     *  @return the resulting join
     */
    <Y> Join<X, Y> join(SingularAttribute<? super X, Y> attribute);

    /**
     *  Create a join to the specified single-valued attribute
     *  using the given join type.
     *  @param attribute  target of the join
     *  @param jt  join type
     *  @return the resulting join
     */
    <Y> Join<X, Y> join(SingularAttribute<? super X, Y> attribute, JoinType jt);

    /**
     *  Create an inner join to the specified Collection-valued
     *  attribute.
     *  @param collection  target of the join
     *  @return the resulting join
     */
    <Y> CollectionJoin<X, Y> join(CollectionAttribute<? super X, Y> collection);

    /**
     *  Create an inner join to the specified Set-valued attribute.
     *  @param set  target of the join
     *  @return the resulting join
     */
    <Y> SetJoin<X, Y> join(SetAttribute<? super X, Y> set);

    /**
     *  Create an inner join to the specified List-valued attribute.
     *  @param list  target of the join
     *  @return the resulting join
     */
    <Y> ListJoin<X, Y> join(ListAttribute<? super X, Y> list);

    /**
     *  Create an inner join to the specified Map-valued attribute.
     *  @param map  target of the join
     *  @return the resulting join
     */
    <K, V> MapJoin<X, K, V> join(MapAttribute<? super X, K, V> map);

    /**
     *  Create a join to the specified Collection-valued attribute
```

```
     *  using the given join type.
     *  @param collection  target of the join
     *  @param jt  join type
     *  @return the resulting join
     */
    <Y> CollectionJoin<X, Y> join(CollectionAttribute<? super X, Y> collection, JoinType
jt);

    /**
     *  Create a join to the specified Set-valued attribute using
     *  the given join type.
     *  @param set  target of the join
     *  @param jt  join type
     *  @return the resulting join
     */
    <Y> SetJoin<X, Y> join(SetAttribute<? super X, Y> set, JoinType jt);

    /**
     *  Create a join to the specified List-valued attribute using
     *  the given join type.
     *  @param list  target of the join
     *  @param jt  join type
     *  @return the resulting join
     */
    <Y> ListJoin<X, Y> join(ListAttribute<? super X, Y> list, JoinType jt);

    /**
     *  Create a join to the specified Map-valued attribute using
     *  the given join type.
     *  @param map  target of the join
     *  @param jt  join type
     *  @return the resulting join
     */
    <K, V> MapJoin<X, K, V> join(MapAttribute<? super X, K, V> map, JoinType jt);


    //String-based:

    /**
     *  Create an inner join to the specified attribute.
     *  @param attributeName  name of the attribute for the
     *               target of the join
     *  @return the resulting join
     *  @throws IllegalArgumentException if attribute of the given
     *          name does not exist
     */
    <X, Y> Join<X, Y> join(String attributeName);
```

```
    /**
     *  Create an inner join to the specified Collection-valued
     *  attribute.
     *  @param attributeName  name of the attribute for the
     *              target of the join
     *  @return the resulting join
     *  @throws IllegalArgumentException if attribute of the given
     *          name does not exist
     */
    <X, Y> CollectionJoin<X, Y> joinCollection(String attributeName);

    /**
     *  Create an inner join to the specified Set-valued attribute.
     *  @param attributeName  name of the attribute for the
     *              target of the join
     *  @return the resulting join
     *  @throws IllegalArgumentException if attribute of the given
     *          name does not exist
     */
    <X, Y> SetJoin<X, Y> joinSet(String attributeName);

    /**
     *  Create an inner join to the specified List-valued attribute.
     *  @param attributeName  name of the attribute for the
     *              target of the join
     *  @return the resulting join
     *  @throws IllegalArgumentException if attribute of the given
     *          name does not exist
     */
    <X, Y> ListJoin<X, Y> joinList(String attributeName);

    /**
     *  Create an inner join to the specified Map-valued attribute.
     *  @param attributeName  name of the attribute for the
     *              target of the join
     *  @return the resulting join
     *  @throws IllegalArgumentException if attribute of the given
     *          name does not exist
     */
    <X, K, V> MapJoin<X, K, V> joinMap(String attributeName);

    /**
     *  Create a join to the specified attribute using the given
     *  join type.
     *  @param attributeName  name of the attribute for the
     *              target of the join
     *  @param jt  join type
     *  @return the resulting join
```

```
     *  @throws IllegalArgumentException if attribute of the given
     *          name does not exist
     */
    <X, Y> Join<X, Y> join(String attributeName, JoinType jt);

    /**
     *  Create a join to the specified Collection-valued attribute
     *  using the given join type.
     *  @param attributeName  name of the attribute for the
     *               target of the join
     *  @param jt  join type
     *  @return the resulting join
     *  @throws IllegalArgumentException if attribute of the given
     *          name does not exist
     */
    <X, Y> CollectionJoin<X, Y> joinCollection(String attributeName, JoinType jt);

    /**
     *  Create a join to the specified Set-valued attribute using
     *  the given join type.
     *  @param attributeName  name of the attribute for the
     *               target of the join
     *  @param jt  join type
     *  @return the resulting join
     *  @throws IllegalArgumentException if attribute of the given
     *          name does not exist
     */
    <X, Y> SetJoin<X, Y> joinSet(String attributeName, JoinType jt);

    /**
     *  Create a join to the specified List-valued attribute using
     *  the given join type.
     *  @param attributeName  name of the attribute for the
     *               target of the join
     *  @param jt  join type
     *  @return the resulting join
     *  @throws IllegalArgumentException if attribute of the given
     *          name does not exist
     */
    <X, Y> ListJoin<X, Y> joinList(String attributeName, JoinType jt);

    /**
     *  Create a join to the specified Map-valued attribute using
     *  the given join type.
     *  @param attributeName  name of the attribute for the
     *               target of the join
     *  @param jt  join type
     *  @return the resulting join
```

```
     *  @throws IllegalArgumentException if attribute of the given
     *          name does not exist
     */
    <X, K, V> MapJoin<X, K, V> joinMap(String attributeName, JoinType jt);
}
```

6.3.16. Root Interface

```
package jakarta.persistence.criteria;

import jakarta.persistence.metamodel.EntityType;

/**
 * A root type in the from clause.
 * Query roots always reference entities.
 *
 * @param <X> the entity type referenced by the root
 *
 * @since 2.0
 */
public interface Root<X> extends From<X, X> {

    /**
     * Return the metamodel entity corresponding to the root.
     * @return metamodel entity corresponding to the root
     */
    EntityType<X> getModel();
}
```

6.3.17. Join Interface

```
package jakarta.persistence.criteria;

import jakarta.persistence.metamodel.Attribute;

/**
 * A join to an entity, embeddable, or basic type.
 *
 * @param <Z> the source type of the join
 * @param <X> the target type of the join
 *
 * @since 2.0
 */
public interface Join<Z, X> extends From<Z, X> {
```

```
    /**
     *  Modify the join to restrict the result according to the
     *  specified ON condition and return the join object.
     *  Replaces the previous ON condition, if any.
     *  @param restriction  a simple or compound boolean expression
     *  @return the modified join object
     *  @since 2.1
     */
    Join<Z, X> on(Expression<Boolean> restriction);

    /**
     *  Modify the join to restrict the result according to the
     *  specified ON condition and return the join object.
     *  Replaces the previous ON condition, if any.
     *  @param restrictions  zero or more restriction predicates
     *  @return the modified join object
     *  @since 2.1
     */
    Join<Z, X> on(Predicate... restrictions);

    /**
     *  Return the predicate that corresponds to the ON
     *  restriction(s) on the join, or null if no ON condition
     *  has been specified.
     *  @return the ON restriction predicate
     *  @since 2.1
     */
    Predicate getOn();

    /**
     * Return the metamodel attribute corresponding to the join.
     * @return metamodel attribute corresponding to the join
     */
    Attribute<? super Z, ?> getAttribute();

    /**
     * Return the parent of the join.
     * @return join parent
     */
    From<?, Z> getParent();

    /**
     * Return the join type.
     * @return join type
     */
    JoinType getJoinType();
}
```

6.3.18. JoinType

```
package jakarta.persistence.criteria;

/**
 * Defines the three types of joins.
 *
 * Right outer joins and right outer fetch joins are not required
 * to be supported. Applications that use <code>RIGHT</code> join
 * types will not be portable.
 *
 * @since 2.0
 */
public enum JoinType {

    /** Inner join. */
    INNER,

    /** Left outer join. */
    LEFT,

    /** Right outer join. */
    RIGHT
}
```

6.3.19. PluralJoin Interface

```
package jakarta.persistence.criteria;

import jakarta.persistence.metamodel.PluralAttribute;

/**
 * The <code>PluralJoin</code> interface defines functionality
 * that is common to joins to all collection types.  It is
 * not intended to be used directly in query construction.
 *
 * @param <Z> the source type
 * @param <C> the collection type
 * @param <E> the element type of the collection
 *
 * @since 2.0
 */
public interface PluralJoin<Z, C, E> extends Join<Z, E> {

    /**
     * Return the metamodel representation for the collection-valued
     * attribute corresponding to the join.
     * @return metamodel collection-valued attribute corresponding
     *         to the target of the join
     */
    PluralAttribute<? super Z, C, E> getModel();
}
```

6.3.20. CollectionJoin Interface

```
package jakarta.persistence.criteria;

import java.util.Collection;
import jakarta.persistence.metamodel.CollectionAttribute;

/**
 * The <code>CollectionJoin</code> interface is the type of the result of
 * joining to a collection over an association or element
 * collection that has been specified as a <code>java.util.Collection</code>.
 *
 * @param <Z> the source type of the join
 * @param <E> the element type of the target <code>Collection</code>
 *
 * @since 2.0
 */
public interface CollectionJoin<Z, E>
        extends PluralJoin<Z, Collection<E>, E> {

    /**
     *  Modify the join to restrict the result according to the
     *  specified ON condition and return the join object.
     *  Replaces the previous ON condition, if any.
     *  @param restriction  a simple or compound boolean expression
     *  @return the modified join object
     *  @since 2.1
     */
    CollectionJoin<Z, E> on(Expression<Boolean> restriction);

    /**
     *  Modify the join to restrict the result according to the
     *  specified ON condition and return the join object.
     *  Replaces the previous ON condition, if any.
     *  @param restrictions  zero or more restriction predicates
     *  @return the modified join object
     *  @since 2.1
     */
    CollectionJoin<Z, E> on(Predicate... restrictions);

    /**
     * Return the metamodel representation for the collection
     * attribute.
     * @return metamodel type representing the <code>Collection</code> that is
     *         the target of the join
     */
    CollectionAttribute<? super Z, E> getModel();
}
```

6.3.21. SetJoin Interface

```
package jakarta.persistence.criteria;

import java.util.Set;
import jakarta.persistence.metamodel.SetAttribute;

/**
 * The <code>SetJoin</code> interface is the type of the result of
 * joining to a collection over an association or element
 * collection that has been specified as a <code>java.util.Set</code>.
 *
 * @param <Z> the source type of the join
 * @param <E> the element type of the target <code>Set</code>
 *
 * @since 2.0
 */
public interface SetJoin<Z, E> extends PluralJoin<Z, Set<E>, E> {

    /**
     *  Modify the join to restrict the result according to the
     *  specified ON condition and return the join object.
     *  Replaces the previous ON condition, if any.
     *  @param restriction  a simple or compound boolean expression
     *  @return the modified join object
     *  @since 2.1
     */
    SetJoin<Z, E> on(Expression<Boolean> restriction);

    /**
     *  Modify the join to restrict the result according to the
     *  specified ON condition and return the join object.
     *  Replaces the previous ON condition, if any.
     *  @param restrictions  zero or more restriction predicates
     *  @return the modified join object
     *  @since 2.1
     */
    SetJoin<Z, E> on(Predicate... restrictions);

    /**
     * Return the metamodel representation for the set attribute.
     * @return metamodel type representing the <code>Set</code> that is
     *         the target of the join
     */
    SetAttribute<? super Z, E> getModel();
}
```

6.3.22. ListJoin Interface

```
package jakarta.persistence.criteria;

import java.util.List;
import jakarta.persistence.metamodel.ListAttribute;

/**
 * The <code>ListJoin</code> interface is the type of the result of
 * joining to a collection over an association or element
 * collection that has been specified as a <code>java.util.List</code>.
 *
 * @param <Z> the source type of the join
 * @param <E> the element type of the target List
 *
 * @since 2.0
 */
public interface ListJoin<Z, E>
        extends PluralJoin<Z, List<E>, E> {

    /**
     *  Modify the join to restrict the result according to the
     *  specified ON condition and return the join object.
     *  Replaces the previous ON condition, if any.
     *  @param restriction  a simple or compound boolean expression
     *  @return the modified join object
     *  @since 2.1
     */
    ListJoin<Z, E> on(Expression<Boolean> restriction);

    /**
     *  Modify the join to restrict the result according to the
     *  specified ON condition and return the join object.
     *  Replaces the previous ON condition, if any.
     *  @param restrictions  zero or more restriction predicates
     *  @return the modified join object
     *  @since 2.1
     */
    ListJoin<Z, E> on(Predicate... restrictions);

    /**
     * Return the metamodel representation for the list attribute.
     * @return metamodel type representing the <code>List</code> that is
     *         the target of the join
     */
    ListAttribute<? super Z, E> getModel();
```

```
    /**
     * Create an expression that corresponds to the index of
     * the object in the referenced association or element
     * collection.
     * This method must only be invoked upon an object that
     * represents an association or element collection for
     * which an order column has been defined.
     * @return expression denoting the index
     */
    Expression<Integer> index();
}
```

6.3.23. MapJoin Interface

```
package jakarta.persistence.criteria;

import java.util.Map;
import jakarta.persistence.metamodel.MapAttribute;

/**
 * The <code>MapJoin</code> interface is the type of the result of
 * joining to a collection over an association or element
 * collection that has been specified as a <code>java.util.Map</code>.
 *
 * @param <Z> the source type of the join
 * @param <K> the type of the target Map key
 * @param <V> the type of the target Map value
 *
 * @since 2.0
 */
public interface MapJoin<Z, K, V>
        extends PluralJoin<Z, Map<K, V>, V> {

    /**
     *  Modify the join to restrict the result according to the
     *  specified ON condition and return the join object.
     *  Replaces the previous ON condition, if any.
     *  @param restriction  a simple or compound boolean expression
     *  @return the modified join object
     *  @since 2.1
     */
    MapJoin<Z, K, V> on(Expression<Boolean> restriction);

    /**
     *  Modify the join to restrict the result according to the
     *  specified ON condition and return the join object.
```

```
     *  Replaces the previous ON condition, if any.
     *  @param restrictions  zero or more restriction predicates
     *  @return the modified join object
     *  @since 2.1
     */
    MapJoin<Z, K, V> on(Predicate... restrictions);

    /**
     * Return the metamodel representation for the map attribute.
     * @return metamodel type representing the <code>Map</code> that is
     *         the target of the join
     */
    MapAttribute<? super Z, K, V> getModel();

    /**
     * Create a path expression that corresponds to the map key.
     * @return path corresponding to map key
     */
    Path<K> key();

    /**
     * Create a path expression that corresponds to the map value.
     * This method is for stylistic use only: it just returns this.
     * @return path corresponding to the map value
     */
    Path<V> value();

    /**
     * Create an expression that corresponds to the map entry.
     * @return expression corresponding to the map entry
     */
    Expression<Map.Entry<K, V>> entry();
}
```

6.3.24. Order Interface

```
package jakarta.persistence.criteria;

/**
 * An object that defines an ordering over the query results.
 *
 * @since 2.0
 */
public interface Order {

   /**
    * Switch the ordering.
    * @return a new <code>Order</code> instance with the reversed ordering
    */
    Order reverse();

   /**
    * Whether ascending ordering is in effect.
    * @return boolean indicating whether ordering is ascending
    */
    boolean isAscending();

   /**
    * Return the expression that is used for ordering.
    * @return expression used for ordering
    */
   Expression<?> getExpression();
}
```

6.3.25. ParameterExpression Interface

```
package jakarta.persistence.criteria;

import jakarta.persistence.Parameter;

/**
 * Type of criteria query parameter expressions.
 *
 * @param <T> the type of the parameter expression
 *
 * @since 2.0
 */
public interface ParameterExpression<T> extends Parameter<T>, Expression<T> {}
```

6.4. Criteria Query API Usage

The *javax.persistence.criteria* API interfaces are designed both to allow criteria queries to be constructed in a strongly-typed manner, using metamodel objects to provide type safety, and to allow for string-based use as an alternative:

Metamodel objects are used to specify navigation through joins and through path expressions [76: The attributes of these metamodel objects play a role analogous to that which would be played by member literals.]. Typesafe navigation is achieved by specification of the source and target types of the navigation.

Strings may be used as an alternative to metamodel objects, whereby joins and navigation are specified by use of strings that correspond to attribute names.

Using either the approach based on metamodel objects or the string-based approach, queries can be constructed both statically and dynamically. Both approaches are equivalent in terms of the range of queries that can be expressed and operational semantics.

Section 6.5 provides a description of the use of the criteria API interfaces. This section is illustrated on the basis of the construction of strongly-typed queries using static metamodel classes. Section 6.6 describes how the *javax.persistence.metamodel* API can be used to construct strongly-typed queries in the absence of such classes. String-based use of the criteria API is described in Section 6.7.

6.5. Constructing Criteria Queries

A criteria query is constructed through the creation and modification of a *javax.persistence.criteria.CriteriaQuery* object.

The *CriteriaBuilder* interface is used to construct *CriteriaQuery*, *CriteriaUpdate*, and *CriteriaDelete* objects. The *CriteriaBuilder* implementation is accessed through the *getCriteriaBuilder* method of the *EntityManager* or *EntityManagerFactory* interface.

For example:

```
@PersistenceUnit
EntityManagerFactory emf;

CriteriaBuilder cb = emf.getCriteriaBuilder();
```

6.5.1. CriteriaQuery Creation

A *CriteriaQuery* object is created by means of one of the *createQuery* methods or the *createTupleQuery* method of the *CriteriaBuilder* interface. A *CriteriaQuery* object is typed according to its expected result type when the *CriteriaQuery* object is created. A *TypedQuery* instance created from the *CriteriaQuery* object by means of the *EntityManager createQuery* method will result in instances of this type when the

resulting query is executed.

The following methods are provided for the creation of *CriteriaQuery* objects:

```
<T> CriteriaQuery<T> createQuery(Class<T> resultClass);

CriteriaQuery<Tuple> createTupleQuery();

CriteriaQuery<Object> createQuery();
```

Methods for the creation of update and delete queries are described in Section 6.5.15.

The methods <T> CriteriaQuery<T> createQuery(Class<T> resultClass) and *createTupleQuery* provide for typing of criteria query results and for typesafe query execution using the *TypedQuery* API.

The effect of the *createTupleQuery* method is semantically equivalent to invoking the *createQuery* method with the *Tuple.class* argument. The *Tuple* interface supports the extraction of multiple selection items in a strongly typed manner. See Section 3.10.3 and Section 3.10.4.

The CriteriaQuery<Object> createQuery() method supports both the case where the *select* or *multiselect* method specifies only a single selection item and where the *multiselect* method specifies multiple selection items. If only a single item is specified, an instance of type *Object* will be returned for each result of the query execution. If multiple selection items are specified, an instance of type *Object[]* will be instantiated and returned for each result of the execution.

See Section 6.5.11 for further discussion of the specification of selection items.

6.5.2. Query Roots

A *CriteriaQuery* object defines a query over one or more entity, embeddable, or basic abstract schema types. The root objects of the query are entities, from which the other types are reached by navigation. A query root plays a role analogous to that of a range variable in the Java Persistence query language and forms the basis for defining the domain of the query.

A query root is created and added to the query by use of the *from* method of the *AbstractQuery* interface (from which both the *CriteriaQuery* and *Subquery* interfaces inherit). The argument to the *from* method is the entity class or *EntityType* instance for the entity. The result of the *from* method is a *Root* object. The *Root* interface extends the *From* interface, which represents objects that may occur in the from clause of a query.

A *CriteriaQuery* object may have more than one root. The addition of a query root has the semantic effect of creating a cartesian product between the entity type referenced by the added root and those of the other roots.

The following query illustrates the definition of a query root. When executed, this query causes all instances of the *Customer* entity to be returned.

```
CriteriaBuilder cb = ...
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> customer = q.from(Customer.class);
q.select(customer);
```

6.5.3. Joins

The *join* methods of the *From* interface extend the query domain by creating a join with a related class that can be navigated to or that is an element of the given class of the query domain.

The target of the join is specified by the corresponding *SingularAttribute* or collection-valued attribute (*CollectionAttribute*, *SetAttribute*, *ListAttribute*, or *MapAttribute*) of the corresponding metamodel class. [77: Metamodel objects are used to specify typesafe nagivation through joins and through path expressions. These metamodel objects capture both the source and target types of the attribute through which navigation occurs, and are thus the mechanism by which typesafe navigation is achieved.] [78: Attribute names serve this role for string-based queries. See Section 6.7.]

The *join* methods may be applied to instances of the *Root* and *Join* types.

The result of a *join* method is a *Join* object (instance of the *Join*, *CollectionJoin*, *SetJoin*, *ListJoin*, or *MapJoin* types) that captures the source and target types of the join.

For example, given the *Order* entity and corresponding *Order_* metamodel class shown in Section 6.2.1.2, a join to the lineItems of the order would be expressed as follows:

```
CriteriaQuery<Order> q = cb.createQuery(Order.class);
Root<Order> order = q.from(Order.class);
Join<Order, Item> item = order.join(Order_.lineItems);
q.select(order);
```

The argument to the *join* method, *Order*.lineItems_, is of type *javax.persistence.metamodel.SetAttribute<Order, Item>*.

The *join* methods have the same semantics as the corresponding Java Persistence query language operations, as described in Section 4.4.7.

Example:

```
CriteriaBuilder cb = ...
CriteriaQuery<String> q = cb.createQuery(String.class);
Root<Customer> customer = q.from(Customer.class);
Join<Customer, Order> order = customer.join(Customer_.orders);
Join<Order, Item> item = order.join(Order_.lineItems);
q.select(customer.get(Customer_.name))
    .where(cb.equal(item.get(Item_.product).get(Product_.productType), "printer"));
```

This query is equivalent to the following Java Persistence query language query:

```
SELECT c.name
FROM Customer c JOIN c.orders o JOIN o.lineItems i
WHERE i.product.productType = 'printer'
```

Joins can be chained, thus allowing this query to be written more concisely:

```
CriteriaQuery<String> q = cb.createQuery(String.class);
Root<Customer> customer = q.from(Customer.class);
Join<Order, Item> item = customer.join(Customer_.orders).join(Order_.lineItems);
q.select(customer.get(Customer_.name))
    .where(cb.equal(item.get(Item_.product).get(Product_.productType), "printer"));
```

By default, the *join* method defines an inner join. Outer joins are defined by specifying a *JoinType* argument. Only left outer joins and left outer fetch joins are required to be supported. Applications that make use of right outer joins or right outer fetch joins will not be portable.

The following query uses a left outer join:

```
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> customer = q.from(Customer.class);
Join<Customer,Order> order = customer.join(Customer_.orders, JoinType.LEFT);
q.where(cb.equal(customer.get(Customer_.status), 1))
    .select(customer);
```

This query is equivalent to the following Java Persistence query language query:

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

On-conditions can be specified for joins. The following query uses an on-condition with a left outer join:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Supplier> s = q.from(Supplier.class);
Join<Supplier, Product> p = s.join(Supplier_.products, JoinType.LEFT);
p.on(cb.equal(p.get(Product_.status), "inStock"));
q.groupBy(s.get(Supplier_.name));
q.multiselect(s.get(Supplier_.name), cb.count(p));
```

This query is equivalent to the following Java Persistence query language query:

```
SELECT s.name, COUNT(p)
FROM Suppliers s LEFT JOIN s.products p ON p.status = 'inStock'
GROUP BY s.name
```

6.5.4. Fetch Joins

Fetch joins are specified by means of the *fetch* method. The *fetch* method specifies that the referenced association or attribute is to be fetched as a side effect of the execution of the query. The *fetch* method can be applied to a *Root* or *Join* instance.

An association or attribute referenced by the *fetch* method must be referenced from an entity or embeddable that is returned as the result of the query. A fetch join has the same join semantics as the corresponding inner or outer join, except that the related objects are not top-level objects in the query result and cannot be referenced elsewhere by the query. See Section 4.4.5.3.

The *fetch* method must not be used in a subquery.

Multiple levels of fetch joins are not required to be supported by an implementation of this specification. Applications that use multi-level fetch joins will not be portable.

Example:

```
CriteriaQuery<Department> q = cb.createQuery(Department.class);
Root<Department> d = q.from(Department.class);
d.fetch(Department.employees, JoinType.LEFT);
q.where(cb.equal(d.get(Department_.deptno), 1)).select(d);
```

This query is equivalent to the following Java Persistence query language query:

```
SELECT d
FROM Department d LEFT JOIN FETCH d.employees
WHERE d.deptno = 1
```

6.5.5. Path Navigation

A *Path* instance can be a *Root* instance, a *Join* instance, a *Path* instance that has been derived from another *Path* instance by means of the *get* navigation method, or a *Path* instance derived from a map-valued association or element collection by use of the *key* or *value* method.

When a criteria query is executed, path navigation—like path navigation using the Java Persistence query language—is obtained using "inner join" semantics. That is, if the value of a non-terminal *Path* instance is null, the path is considered to have no value, and does not participate in the determination of the query result. See [a4792].

The *get* method is used for path navigation. The argument to the *get* method is specified by the corresponding *SingularAttribute* or collection-valued attribute *(CollectionAttribute*, *SetAttribute*, *ListAttribute*, or *MapAttribute*) of the corresponding metamodel class [79: Attribute names serve this role for string-based queries. See Section 6.7.].

Example 1:

In the following example, *ContactInfo* is an embeddable class consisting of an address and set of phones. *Phone* is an entity.

```
CriteriaQuery<Vendor> q = cb.createQuery(Vendor.class);
Root<Employee> emp = q.from(Employee.class);
Join<ContactInfo, Phone> phone =
    emp.join(Employee_.contactInfo).join(ContactInfo_.phones);
q.where(cb.equal(emp.get(Employee_.contactInfo)
        .get(ContactInfo_.address)
        .get(Address_.zipcode), "95054"))
    .select(phone.get(Phone_.vendor));
```

The following Java Persistence query language query is equivalent:

```
SELECT p.vendor
FROM Employee e JOIN e.contactInfo.phones p
WHERE e.contactInfo.address.zipcode = '95054'
```

Example 2:

In this example, the *photos* attribute corresponds to a map from photo label to filename. The map key is a string, the value an object. The result of this query will be returned as a *Tuple* object whose elements are of types *String* and *Object*. The *multiselect* method, described further in Section 6.5.11, is used to specify that the query returns multiple selection items.

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Item> item = q.from(Item.class);
MapJoin<Item, String, Object> photo = item.join(Item_.photos);
q.multiselect(item.get(Item_.name), photo)
    .where(cb.like(photo.key(), "%egret%"));
```

This query is equivalent to the following Java Persistence query language query:

```
SELECT i.name, p
FROM Item i JOIN i.photos p
WHERE KEY(p) LIKE '%egret%'
```

6.5.6. Restricting the Query Result

The result of a query can be restricted by specifying one or more predicate conditions. Restriction predicates are applied to the *CriteriaQuery* object by means of the *where* method. Invocation of the *where* method results in the modification of the *CriteriaQuery* object with the specified restriction(s).

The argument to the *where* method can be either an *Expression<Boolean>* instance or zero or more *Predicate* instances. A predicate can be either simple or compound.

A simple predicate is created by invoking one of the conditional methods of the *CriteriaBuilder* interface, or by the *isNull*, *isNotNull*, and *in* methods of the *Expression* interface. The semantics of the conditional methods—e.g., *equal*, *notEqual*, *gt*, *ge*, *lt*, *le*, *between*, and *like* — mirror those of the corresponding Java Persistence query language operators as described in Chapter 4.

Compound predicates are constructed by means of the *and*, *or*, and *not* methods of the *CriteriaBuilder* interface.

The restrictions upon the types to which conditional operations are permitted to be applied are the same as the respective operators of the Java Persistence query language as described in subsections Section 4.6.7 through Section 4.6.17. The same null value semantics as described in Section 4.11 and the subsections of Section 4.6 apply. The equality and comparison semantics described in Section 4.12 likewise apply.

Example 1:

```
CriteriaQuery<TransactionHistory> q = cb.createQuery(TransactionHistory.class);
Root<CreditCard> cc = q.from(CreditCard.class);
ListJoin<CreditCard,TransactionHistory> t = cc.join(CreditCard_.transactionHistory);
q.select(t)
    .where(cb.equal(cc.get(CreditCard_.customer)
             .get(Customer_.accountNum), 321987),
           cb.between (t.index(), 0, 9));
```

This query is equivalent to the following Java Persistence query language query:

```
SELECT t
FROM CreditCard c JOIN c.transactionHistory t
WHERE c.customer.accountNum = 321987 AND INDEX(t) BETWEEN 0 AND 9
```

Example 2:

```
CriteriaQuery<Order> q = cb.createQuery(Order.class);
Root<Order> order = q.from(Order.class);
q.where(cb.isEmpty(order.get(Order_.lineItems)))
    .select(order);
```

This query is equivalent to the following Java Persistence query language query:

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

6.5.7. Downcasting

Downcasting by means of the *treat* method is supported in joins and in the construction of *where* conditions.

Example 1:

```
CriteriaQuery<String> q = cb.createQuery(String.class);
Root<Order> order = q.from(Order.class);
Join<Order,Book> book = cb.treat(order.join(Order_.product), Book.class);
q.select(book.get(Book_.isbn));
```

This query is equivalent to the following Java Persistence query language query.

```
SELECT b.ISBN
FROM Order o JOIN TREAT(o.product AS Book) b
```

Example 2:

```
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> customer = q.from(Customer.class);
Join<Customer, Order> order = customer.join(Customer_.orders);
q.where(
    cb.equal(cb.treat(order.get(Order_.product), Book.class).get(Book_.name), "Iliad"));
q.select(customer);
```

This query is equivalent to the following Java Persistence query language query:

```
SELECT c
FROM Customer c JOIN c.orders o
WHERE TREAT(o.product AS Book).name = 'Iliad'
```

Example 3:

```
CriteriaQuery<Employee> q = cb.createQuery(Employee.class);
Root<Employee> e = q.from(Employee.class);
q.where(
    cb.or(cb.gt(cb.treat(e, Exempt.class).get(Exempt_.vacationDays), 10),
    cb.gt(cb.treat(e, Contractor.class).get(Contractor_.hours), 100)));
```

This query is equivalent to the following Java Persistence query language query:

```
SELECT e
FROM Employee e
WHERE TREAT(e AS Exempt).vacationDays > 10
    OR TREAT(e AS Contractor).hours > 100
```

6.5.8. Expressions

An *Expression* or one of its subtypes can be used in the construction of the query's select list or in the construction of *where* or *having* method conditions.

Paths and boolean predicates are expressions.

Other expressions are created by means of the methods of the *CriteriaBuilder* interface. The

CriteriaBuilder interface provides methods corresponding to the built-in arthmetic, string, datetime, and case operators and functions of the Java Persistence query language.

Example 1:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Customer> cust = q.from(Customer.class);
Join<Customer, Order> order = cust.join(Customer_.orders);
Join<Customer, Address> addr = cust.join(Customer_.address);
q.where(cb.equal(addr.get(Address_.state), "CA"),
        cb.equal(addr.get(Address_.county), "Santa Clara"));

q.multiselect(order.get(Order_.quantity),
        cb.prod(order.get(Order_.totalCost), 1.08),
        addr.get(Address_.zipcode));
```

The following Java Persistence query language query is equivalent:

```
SELECT o.quantity, o.totalCost*1.08, a.zipcode
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA' AND a.county = 'Santa Clara'
```

Example 2:

```
CriteriaQuery<Employee> q = cb.createQuery(Employee.class);
Root<Employee> emp = q.from(Employee.class);
q.select(emp)
    .where(cb.notEqual(emp.type(), Exempt.class));
```

The *type* method can only be applied to a path expression. Its result denotes the type navigated to by the path.

The following Java Persistence query language query is equivalent:

```
SELECT e
FROM Employee e
WHERE TYPE(e) <> Exempt
```

Example 3:

```
CriteriaQuery<String> q = cb.createQuery(String.class);
Root<Course> c = q.from(Course.class);
ListJoin<Course, Student> w = c.join(Course_.studentWaitlist);
q.where(cb.equal(c.get(Course_.name), "Calculus"),
        cb.equal(w.index(), 0))
 .select(w.get(Student_.name));
```

The *index* method can be applied to a *ListJoin* object that corresponds to a list for which an order column has been specified. Its result denotes the position of the item in the list.

The following Java Persistence query language query is equivalent:

```
SELECT w.name
FROM Course c JOIN c.studentWaitlist w
WHERE c.name = 'Calculus' AND INDEX(w) = 0
```

Example 4:

```
CriteriaQuery<BigDecimal> q = cb.createQuery(BigDecimal.class);
Root<Order> order = q.from(Order.class);
Join<Order, Item> item = order.join(Order_.lineItems);
Join<Order, Customer> cust = order.join(Order_.customer);

q.where(
    cb.equal(cust.get(Customer_.lastName), "Smith"),
    cb.equal(cust.get(Customer_.firstName), "John"));
q.select(cb.sum(item.get(Item_.price)));
```

The aggregation methods *avg*, *max*, *min* , *sum*, *count* can only be used in the construction of the select list or in *having* method conditions.

The following Java Persistence query language query is equivalent:

```
SELECT SUM(i.price)
FROM Order o JOIN o.lineItems i JOIN o.customer c
WHERE c.lastName = 'Smith' AND c.firstName = 'John'
```

Example 5:

```
CriteriaQuery<Integer> q = cb.createQuery(Integer.class);
Root<Department> d = q.from(Department.class);
q
    .where(cb.equal(d.get(Department_.name), "Sales"))
    .select(cb.size(d.get(Department_.employees)));
```

The *size* method can be applied to a path expression that corresponds to an association or element collection. Its result denotes the number of elements in the association or element collection.

The following Java Persistence query language query is equivalent:

```
SELECT SIZE(d.employees)
FROM Department d
WHERE d.name = 'Sales'
```

Example 6:

Both simple and general case expressions are supported. The query below illustrates use of a general case expression.

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Employee> e = q.from(Employee.class);
q.where(
    cb.equal(e.get(Employee_.department).get(Department_.name), "Engineering"));
q.multiselect(
    e.get(Employee_.name),
    cb.selectCase()
        .when(
            cb.equal(e.get(Employee_.rating), 1),
            cb.prod(e.get(Employee_.salary), 1.1))
        .when(
            cb.equal(e.get(Employee_.rating), 2),
            cb.prod(e.get(Employee_.salary), 1.2))
        .otherwise(cb.prod(e.get(Employee_.salary), 1.01)));
```

The following Java Persistence query language query is equivalent:

```
SELECT e.name,
    CASE
        WHEN e.rating = 1 THEN e.salary * 1.1
        WHEN e.rating = 2 THEN e.salary * 1.2
        ELSE e.salary * 1.01
    END
FROM EMPLOYEE e
WHERE e.department.name = 'Engineering'
```

6.5.8.1. Result Types of Expressions

The *getJavaType* method, as defined in the *TupleElement* interface, returns the runtime type of the object on which it is invoked.

In the case of the *In*, *Case*, *SimpleCase*, and *Coalesce* builder interfaces, the runtime results of the *getJavaType* method may differ from the *Expression* type and may vary as the expression is incrementally constructed. For non-numerical operands, the implementation must return the most specific common superclass of the types of the operands used to form the result.

In the case of the two-argument *sum*, *prod*, *diff*, *quot*, *coalesce*, and *nullif* methods, and the *In*, *Case*, *SimpleCase*, and *Coalesce* builder methods, the runtime result types will differ from the *Expression* type when the latter is *Number*. The following rules must be observed by the implementation when materializing the results of numeric expressions involving these methods. These rules correspond to those specified for the Java Persistence query language as defined in Section 4.8.6.

- If there is an operand of type Double, the result of the operation is of type Double;
- otherwise, if there is an operand of type Float, the result of the operation is of type Float;
- otherwise, if there is an operand of type BigDecimal, the result of the operation is of type BigDecimal;
- otherwise, if there is an operand of type BigInteger, the result of the operation is of type BigInteger, unless the method is *quot*, in which case the numeric result type is not further defined;
- otherwise, if there is an operand of type Long, the result of the operation is of type Long, unless the method is *quot*, in which case the numeric result type is not further defined;
- otherwise, if there is an operand of integral type, the result of the operation is of type Integer, unless the method is *quot*, in which case the numeric result type is not further defined.

Users should note that the semantics of the SQL division operation are not standard across databases. In particular, when both operands are of integral types, the result of the division operation will be an integral type in some databases, and an non-integral type in others. Portable applications should not assume a particular result type.

6.5.9. Literals

An *Expression* literal instance is obtained by passing a value to the *literal* method of the *CriteriaBuilder* interface. An *Expression* instance representing a null is created by the *nullLiteral* method of the *CriteriaBuilder* interface.

Example:

```
CriteriaQuery<String> q = cb.createQuery(String.class);
Root<Employee> emp = q.from(Employee.class);
Join<Employee, FrequentFlierPlan> fp = emp.join(Employee_.frequentFlierPlan);

q.select(
    cb.<String>selectCase()
        .when(
            cb.gt(fp.get(FrequentFlierPlan_.annualMiles), 50000),
            cb.literal("Platinum"))
        .when(
            cb.gt(fp.get(FrequentFlierPlan_.annualMiles), 25000),
            cb.literal("Silver"))
        .otherwise(cb.nullLiteral(String.class)));
```

The following Java Persistence query language query is equivalent:

```
SELECT
    CASE
        WHEN fp.annualMiles > 50000 THEN 'Platinum'
        WHEN fp.annualMiles > 25000 THEN 'Gold'
        ELSE NULL
    END
```

6.5.10. Parameter Expressions

A *ParameterExpression* instance is an expression that corresponds to a parameter whose value will be supplied before the query is executed. Parameter expressions can only be used in the construction of conditional predicates.

Example:

```
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> c = q.from(Customer.class);
ParameterExpression<Integer> param = cb.parameter(Integer.class);

q.select(c)
    .where(cb.equal(c.get(Customer_.status), param));
```

If a name is supplied when the *ParameterExpression* instance is created, the parameter may also be treated as a named parameter when the query is executed:

CriteriaQuery<Customer> q = cb.createQuery(Customer.class);

```
Root<Customer> c = q.from(Customer.class);
ParameterExpression<Integer> param = cb.parameter(Integer.class, "stat");
q.select(c).where(cb.equal(c.get(Customer_.status), param));
```

This is equivalent to the following query in the Java Persistence query language:

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

6.5.11. Specifying the Select List

The select list of a query is specified by use of the *select* or *multiselect* methods of the *CriteriaQuery* interface. The arguments to the *select* and *multiselect* methods are *Selection* instances.

Portable applications should use the *select* or *multiselect* method to specify the query's selection list. Applications that do not use one of these methods will not be portable.

The *select* method takes a single *Selection* argument, which can be either an *Expression* instance or a *CompoundSelection* instance. The type of the *Selection* item must be assignable to the defined *CriteriaQuery* result type, as described in Section 6.5.1.

The *construct*, *tuple* and *array* methods of the *CriteriaBuilder* interface are used to aggregate multiple selection items into a *CompoundSelection* instance.

The *multiselect* method also supports the specification and aggregation of multiple selection items. When the *multiselect* method is used, the aggregation of the selection items is determined by the result type of the *CriteriaQuery* object as described in Section 6.5.1 and Section 6.3.4.

A *Selection* instance passed to the *construct*, *tuple*, *array*, or *multiselect* methods can be one of the following:

- An *Expression* instance.

- A *Selection* instance obtained as the result of the invocation of the *CriteriaBuilder construct* method.

The *distinct* method of the *CriteriaQuery* interface is used to specify that duplicate values must be eliminated from the query result. If the *distinct* method is not used or *distinct(false)* is invoked on the criteria query object, duplicate values are not eliminated. When *distinct(true)* is used, and the select items include embeddable objects or map entry results, the elimination of duplicates is undefined.

The semantics of the *construct* method used in the selection list is as described in Section 4.8.2. The semantics of embeddables returned by the selection list areas described in Section 4.8.4.

Example 1:

In the following example, *videoInventory* is a Map from the entity *Movie* to the number of copies in stock.

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<VideoStore> v = q.from(VideoStore.class);
MapJoin<VideoStore, Movie, Integer> inv = v.join(VideoStore_.videoInventory);

q.multiselect(
    v.get(VideoStore_.location).get(Address_.street),
    inv.key().get(Movie_.title),
    inv);
q.where(cb.equal(v.get(VideoStore_.location).get(Address_.zipcode), "94301"),
        cb.gt(inv, 0));
```

This query is equivalent to the following, in which the *tuple* method is used:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<VideoStore> v = q.from(VideoStore.class);
MapJoin<VideoStore, Movie, Integer> inv = v.join(VideoStore_.videoInventory);

q.select(cb.tuple(
        v.get(VideoStore_.location).get(Address_.street),
        inv.key().get(Movie_.title),
        inv));
q.where(cb.equal(v.get(VideoStore_.location).get(Address_.zipcode), "94301"),
        cb.gt(inv, 0));
```

Both are equivalent to the following Java Persistence query language query:

```
SELECT v.location.street, KEY(i).title, VALUE(i)
FROM VideoStore v JOIN v.videoInventory i
WHERE v.location.zipcode = '94301' AND VALUE(i) > 0
```

Example 2:

The following two queries are equivalent to the Java Persistence query language query above. Because the result type is not specified by the_ *createQuery method, an Object[]* _is returned as a result of the query execution:

```
CriteriaQuery<Object> q = cb.createQuery();
Root<VideoStore> v = q.from(VideoStore.class);
MapJoin<VideoStore, Movie, Integer> inv = v.join(VideoStore_.videoInventory);

q.multiselect(
    v.get(VideoStore_.location).get(Address_.street),
    inv.key().get(Movie_.title),
    inv);

q.where(cb.equal(v.get(VideoStore_.location).get(Address_.zipcode), "94301"),
        cb.gt(inv, 0));
```

Equivalently:

```
CriteriaQuery<Object> q = cb.createQuery();
Root<VideoStore> v = q.from(VideoStore.class);
MapJoin<VideoStore, Movie, Integer> inv = v.join(VideoStore_.videoInventory);

q.select(cb.array(
        v.get(VideoStore_.location).get(Address_.street),
        inv.key().get(Movie_.title),
        inv));
q.where(cb.equal(v.get(VideoStore_.location).get(Address_.zipcode), "94301"),
        cb.gt(inv, 0));
```

Example 3:

The following example illustrates the specification of a constructor.

```
CriteriaQuery<CustomerDetails> q = cb.createQuery(CustomerDetails.class);
Root<Customer> c = q.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);

q.where(cb.gt(o.get(Order_.quantity), 100));
q.select(cb.construct(
        CustomerDetails.class,
        c.get(Customer_.id),
        c.get(Customer_.status),
        o.get(Order_.quantity)));
```

The following Java Persistence query language query is equivalent:

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

6.5.11.1. Assigning Aliases to Selection Items

The *alias* method of the *Selection* interface can be used to assign an alias to a selection item. The alias may then later be used to extract the corresponding item from the query result when the query is executed. The *alias* method assigns the given alias to the *Selection* item. Once assigned, the alias cannot be changed.

Example:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Customer> c = q.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
Join<Customer, Address> a = c.join(Customer_.address);

q.where(cb.equal(c.get(Customer_.id), 97510));
q.multiselect(
        o.get(Order_.quantity).alias("quantity"),
        cb.prod(o.get(Order_.totalCost), 1.08).alias("taxedCost"),
        a.get(Address_.zipcode).alias("zipcode"));

TypedQuery<Tuple> typedQuery = em.createQuery(q);
Tuple result = typedQuery.getSingleResult();
Double cost = (Double)result.get("taxedCost");
```

6.5.12. Subqueries

Both correlated and non-correlated subqueries can be used in restriction predicates. A subquery is constructed through the creation and modification of a *Subquery* object.

A *Subquery* instance can be passed as an argument to the *all*, *any*, or *some* methods of the *CriteriaBuilder* interface for use in conditional expressions.

A *Subquery* instance can be passed to the *CriteriaBuilder exists* method to create a conditional predicate.

Example 1: Non-correlated subquery

The query below contains a non-correlated subquery. A non-correlated subquery does not reference objects of the query of which it is a subquery. In particular, *Root*, *Join*, and *Path* instances are not shared between the subquery and the criteria query instance of which it is a subquery.

```
// create criteria query instance, with root Customer
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> goodCustomer = q.from(Customer.class);

// create subquery instance, with root Customer
// the Subquery object is typed according to its return type
Subquery<Double> sq = q.subquery(Double.class);
Root<Customer> customer = sq.from(Customer.class);

// the result of the first query depends on the subquery
q.where(cb.lt(
    goodCustomer.get(Customer_.balanceOwed),
    sq.select(cb.avg(customer.get(Customer_.balanceOwed)))));
q.select(goodCustomer);
```

This query corresponds to the following Java Persistence query language query.

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

Example 2: Correlated subquery

```
// create CriteriaQuery instance, with root Employee
CriteriaQuery<Employee> q = cb.createQuery(Employee.class);
Root<Employee> emp = q.from(Employee.class);

// create Subquery instance, with root Employee
Subquery<Employee> sq = q.subquery(Employee.class);
Root<Employee> spouseEmp = sq.from(Employee.class);

// the subquery references the root of the containing query
sq.where(cb.equal(spouseEmp, emp.get(Employee_.spouse)))
    .select(spouseEmp);

// an exists condition is applied to the subquery result:
q.where(cb.exists(sq));
q.select(emp).distinct(true);
```

The above query corresponds to the following Java Persistence query language query.

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

Example 3: Subquery qualified by all()

```
// create CriteriaQuery instance, with root Employee
CriteriaQuery<Employee> q = cb.createQuery(Employee.class);
Root<Employee> emp = q.from(Employee.class);

// create Subquery instance, with root Manager
Subquery<BigDecimal> sq = q.subquery(BigDecimal.class);
Root<Manager> manager = sq.from(Manager.class);

sq.select(manager.get(Manager_.salary));
sq.where(cb.equal(
    manager.get(Manager_.department),
    emp.get(Employee_.department)));

// an all expression is applied to the subquery result
q.select(emp)
    .where(cb.gt(emp.get(Employee_.salary), cb.all(sq)));
```

This query corresponds to the following Java Persistence query language query:

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

Example 4: A Special case

In order to express some correlated subqueries involving unidirectional relationships, it may be useful to correlate the domain of the subquery with the domain of the containing query. This is performed by using the *correlate* method of the *Subquery* interface.

For example:

```
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> customer = q.from(Customer.class);
Subquery<Long> sq = q.subquery(Long.class);
Root<Customer> customerSub = sq.correlate(customer);
Join<Customer,Order> order = customerSub.join(Customer_.orders);

q.where(cb.gt(sq.select(cb.count(order)), 10))
    .select(customer);
```

This query corresponds to the following Java Persistence query language query:

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

Note that joins involving the derived subquery root do not affect the join conditions of the containing query. The following two query definitions thus differ in semantics:

```
CriteriaQuery<Order> q = cb.createQuery(Order.class);
Root<Order> order = q.from(Order.class);
Subquery<Integer> sq = q.subquery(Integer.class);
Root<Order> orderSub = sq.correlate(order);
Join<Order,Customer> customer = orderSub.join(Order_.customer);
Join<Customer,Account> account = customer.join(Customer_.accounts);

sq.select(account.get(Account_.balance));
q.where(cb.lt(cb.literal(10000), cb.all(sq)));
```

and

```
CriteriaQuery<Order> q = cb.createQuery(Order.class);
Root<Order> order = q.from(Order.class);
Join<Order,Customer> customer = order.join(Order_.customer);
Subquery<Integer> sq = q.subquery(Integer.class);
Join<Order,Customer> customerSub = sq.correlate(customer);
Join<Customer,Account> account = customerSub.join(Customer_.accounts);

sq.select(account.get(Account_.balance));
q.where(cb.lt(cb.literal(10000), cb.all(sq)));
```

The first of these queries will return orders that are not associated with customers, whereas the second will not. The corresponding Java Persistence query language queries are the following:

```
SELECT o
FROM Order o
WHERE 10000 < ALL (
    SELECT a.balance
    FROM o.customer c JOIN c.accounts a)
```

and

```
SELECT o
FROM Order o JOIN o.customer c
WHERE 10000 < ALL (
    SELECT a.balance
    FROM c.accounts a)
```

6.5.13. GroupBy and Having

The *groupBy* method of the *CriteriaQuery* interface is used to define a partitioning of the query results

into groups. The *having* method of the *CriteriaQuery* interface can be used to filter over the groups.

The arguments to the *groupBy* method are *Expression* instances.

When the *groupBy* method is used, each selection item that is not the result of applying an aggregate method must correspond to a path expression that is used for defining the grouping. Requirements on the types that correspond to the elements of the grouping and having constructs and their relationship to the select items are as specified in Section 4.7.

Example:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Customer> customer = q.from(Customer.class);

q.groupBy(customer.get(Customer_.status));
q.having(cb.in(customer.get(Customer_.status)).value(1).value(2));
q.select(cb.tuple(
        customer.get(Customer_.status),
        cb.avg(customer.get(Customer_.filledOrderCount)),
        cb.count(customer)));
```

This query is equivalent to the following Java Persistence query language query:

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

6.5.14. Ordering the Query Results

The ordering of the results of a query is defined by use of the *orderBy* method of the *CriteriaQuery* instance. The arguments to the *orderBy* method are *Order* instances.

An *Order* instance is created by means of the *asc* and *desc* methods of the *CriteriaBuilder* interface. An argument to either of these methods must be one of the following:

- Any *Expression* instance that corresponds to an orderable state field of an entity or embeddable class abstract schema type that is specified as an argument to the *select* or *multiselect* method or that is an argument to a tuple or array constructor that is passed as an argument to the *select* method.
- Any *Expression* instance that corresponds to the same state field of the same entity or embeddable abstract schema type as an *Expression* instance that is specified as an argument to the *select* or *multiselect* method or that is an argument to a tuple or array constructor that is passed as an argument to the *select* method.

- An *Expression* instance that is specified as an argument to the *select* or *multiselect* method or that is an argument to a tuple or array constructor that is passed as an argument to the *select* method or that is semantically equivalent to such an *Expression* instance.

If more than one *Order* instance is specified, the order in which they appear in the argument list of the *orderBy* method determines the precedence, whereby the first item has highest precedence.

SQL rules for the ordering of null values apply, as described in Section 4.9.

Example 1:

```
CriteriaQuery<Order> q = cb.createQuery(Order.class);
Root<Customer> c = q.from(Customer.class);
Join<Customer,Order> o = c.join(Customer_.orders);
Join<Customer,Address> a = c.join(Customer_.address);

q.where(cb.equal(a.get(Address_.state), "CA"));
q.select(o);
q.orderBy(cb.desc(o.get(Order_.quantity)),
    cb.asc(o.get(Order_.totalCost)));
```

This query corresponds to the following Java Persistence query language query:

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

Example 2:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Customer> c = q.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
Join<Customer, Address> a = c.join(Customer_.address);

q.where(cb.equal(a.get(Address_.state), "CA"));
q.orderBy(cb.asc(o.get(Order_.quantity)),
    cb.asc(a.get(Address_.zipcode)));
q.multiselect(o.get(Order_.quantity),
    a.get(Address_.zipcode));
```

This query corresponds to the following Java Persistence query language query:

```
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
```

It can be equivalently expressed as follows:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Customer> c = q.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
Join<Customer, Address> a = c.join(Customer_.address);

q.where(cb.equal(a.get(Address_.state), "CA"));
q.orderBy(cb.asc(o.get(Order_.quantity)),
    cb.asc(a.get(Address_.zipcode)));
q.select(cb.tuple(o.get(Order_.quantity),
    a.get(Address_.zipcode)));
```

Example 3:

```
CriteriaQuery<Object[]> q = cb.createQuery(Object[].class);
Root<Customer> c = q.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
Join<Customer, Address> a = c.join(Customer_.address);

q.where(cb.equal(a.get(Address_.state), "CA"),
    cb.equal(a.get(Address_.county), "Santa Clara"));
q.select(cb.array(o.get(Order_.quantity),
    cb.prod(o.get(Order_.totalCost), 1.08),
    a.get(Address_.zipcode)));
q.orderBy(cb.asc(o.get(Order_.quantity)),
    cb.asc(cb.prod(o.get(Order_.totalCost), 1.08)),
    cb.asc(a.get(Address_.zipcode)));
```

This query corresponds to the following Java Persistence query language query:

```
SELECT o.quantity, o.totalCost * 1.08 AS taxedCost, a.zipcode
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA' AND a.county = 'Santa Clara'
ORDER BY o.quantity, taxedCost, a.zipcode
```

6.5.15. Bulk Update and Delete Operations

A bulk update query is constructed through the creation and modification of a *javax.persistence.criteria.CriteriaUpdate* object.

A *CriteriaUpdate* object is created by means of one of the *createCriteriaUpdate* methods of the *CriteriaBuilder* interface. A *CriteriaUpdate* object is typed according to the entity type that is the target of the update. A *CriteriaUpdate* object has a single root, the entity that is being updated.

A bulk delete query is constructed through the creation and modification of a *javax.persistence.criteria.CriteriaDelete* object.

A *CriteriaDelete* object is created by means of one of the *createCriteriaDelete* methods of the *CriteriaBuilder* interface. A *CriteriaDelete* object is typed according to the entity type that is the target of the delete. A *CriteriaDelete* object has a single root, the entity that is being deleted.

Example 1:

```
CriteriaUpdate<Customer> q = cb.createCriteriaUpdate(Customer.class);
Root<Customer> c = q.from(Customer.class);

q.set(c.get(Customer_.status), "outstanding")
    .where(cb.lt(c.get(Customer_.balance), 10000));
```

The following Java Persistence query language update statement is equivalent.

```
UPDATE Customer c
SET c.status = 'outstanding'
WHERE c.balance < 10000
```

Example 2:

```
CriteriaUpdate<Employee> q = cb.createCriteriaUpdate(Employee.class);
Root<Employee> e = q.from(Employee.class);

q.set(e.get(Employee_.address).get(Address_.building), 22)
    .where(
        cb.equal(e.get(Employee_.address).get(Address_.building), 14),
        cb.equal(e.get(Employee_.address).get(Address_.city), "Santa Clara"),
        cb.equal(e.get(Employee_.project).get(Project_.name), "Java EE"));
```

Address is an embeddable class. Note that updating across implicit joins is not supported.

The following Java Persistence query language update statement is equivalent.

```
UPDATE Employee e
SET e.address.building = 22
WHERE e.address.building = 14
    AND e.address.city = 'Santa Clara'
    AND e.project.name = 'Java EE'
```

Example 3:

The following update query causes multiple attributes to be updated.

```
CriteriaUpdate<Employee> q = cb.createCriteriaUpdate(Employee.class);
Root<Employee> e = q.from(Employee.class);

q.set(e.get(Employee_.salary), cb.prod(e.get(Employee_.salary), 1.1f))
    .set(e.get(Employee_.commission), cb.prod(e.get(Employee_.commission), 1.1f))
    .set(e.get(Employee_.bonus), cb.sum(e.get(Employee_.bonus), 5000))
    .where(cb.equal(e.get(Employee_.dept).get(Department_.name), "Sales"));
```

The following Java Persistence query language update statement is equivalent.

```
UPDATE Employee e
SET e.salary = e.salary * 1.1,
    e.commission = e.commission * 1.1,
    e.bonus = e.bonus + 5000
WHERE e.dept.name = 'Sales'
```

Example 4:

```
CriteriaDelete<Customer> q = cb.createCriteriaDelete(Customer.class);
Root<Customer> c = q.from(Customer.class);

q.where(
    cb.equal(c.get(Customer_.status), "inactive"),
    cb.isEmpty(c.get(Customer_.orders)));
```

The following Java Persistence query language delete statement is equivalent.

```
DELETE
FROM Customer c
WHERE c.status = 'inactive'
    AND c.orders IS EMPTY
```

Like bulk update and delete operations made through the Java Persistence query language, criteria API bulk update and delete operations map directly to database operations, bypassing any optimistic locking checks. Portable applications using bulk update operations must manually update the value of the version column, if desired, and/or manually validate the value of the version column.

The persistence context is not synchronized with the result of the bulk update or delete. See Section 4.10.

6.6. Constructing Strongly-typed Queries using the javax.persistence.metamodel Interfaces

Strongly-typed queries can also be constructed, either statically or dynamically, in the absence of generated metamodel classes. The *javax.persistence.metamodel* interfaces are used to access the metamodel objects that correspond to the managed classes.

The following examples illustrate this approach. These are equivalent to the example queries shown in Section 6.5.5.

The *Metamodel* interface is obtained from the EntityManager or EntityManagerFactory for the persistence unit, and then used to obtain the corresponding metamodel objects for the managed types referenced by the queries.

Example 1:

```
EntityManager em = ...;

Metamodel mm = em.getMetamodel();
EntityType<Employee> emp_ =mm.entity(Employee.class);
EmbeddableType<ContactInfo> cinfo_ = mm.embeddable(ContactInfo.class);
EntityType<Phone> phone_ = mm.entity(Phone.class);
EmbeddableType<Address> addr_ = mm.embeddable(Address.class);

CriteriaQuery<Vendor> q = cb.createQuery(Vendor.class);
Root<Employee> emp = q.from(Employee.class);
Join<Employee, ContactInfo> cinfo =
    emp.join(emp_.getSingularAttribute("contactInfo", ContactInfo.class));
Join<ContactInfo, Phone> p =
    cinfo.join(cinfo_.getSingularAttribute("phones", Phone.class));
q.where(
    cb.equal(emp.get(emp_.getSingularAttribute("contactInfo", ContactInfo.class))
        .get(cinfo_.getSingularAttribute("address", Address.class))
        .get(addr_.getSingularAttribute("zipcode", String.class)), "95054"))
    .select(p.get(phone_.getSingularAttribute("vendor",Vendor.class)));
```

Example 2:

```
EntityManager em = ...;
Metamodel mm = em.getMetamodel();

EntityType<Item> item_ = mm.entity(Item.class);
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Item> item = q.from(Item.class);
MapJoin<Item, String, Object> photo =
    item.join(item_.getMap("photos", String.class, Object.class));
q.multiselect(
    item.get(item_.getSingularAttribute("name", String.class)), photo)
        .where(cb.like(photo.key(), "%egret%"));
```

6.7. Use of the Criteria API with Strings to Reference Attributes

The Criteria API provides the option of specifying the attribute references used in joins and navigation by attribute names used as arguments to the various *join*, *fetch*, and *get* methods.

The resulting queries have the same semantics as described in Section 6.5, but do not provide the same level of type safety.

The examples in this section illustrate this approach. These examples are derived from among those of sections Section 6.5.3 and Section 6.5.5.

Example 1:

```
CriteriaBuilder cb = ...
CriteriaQuery<String> q = cb.createQuery(String.class);
Root<Customer> cust = q.from(Customer.class);
Join<Order, Item> item = cust.join("orders").join("lineItems");
q.select(cust.<String>get("name"))
    .where(cb.equal(item.get("product").get("productType"), "printer"));
```

This query is equivalent to the following Java Persistence query language query:

```
SELECT c.name
FROM Customer c JOIN c.orders o JOIN o.lineItems i
WHERE i.product.productType = 'printer'
```

It is not required that type parameters be used. However, their omission may result in compiler warnings, as with the below version of the same query:

```
CriteriaBuilder cb = ...
CriteriaQuery q = cb.createQuery();
Root cust = q.from(Customer.class);
Join item = cust.join("orders").join("lineItems");
q.select(cust.get("name")).where(
    cb.equal(item.get("product").get("productType"),"printer"));
```

Example 2:

The following query uses an outer join:

```
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> cust = q.from(Customer.class);
Join<Customer,Order> order = cust.join("orders", JoinType.LEFT);
q.where(cb.equal(cust.get("status"), 1))
    .select(cust);
```

This query is equivalent to the following Java Persistence query language query:

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

Example 3:

In the following example, *ContactInfo* is an embeddable class consisting of an address and set of phones. *Phone* is an entity.

```
CriteriaQuery<Vendor> q = cb.createQuery(Vendor.class);
Root<Employee> emp = q.from(Employee.class);
Join<ContactInfo, Phone> phone = emp.join("contactInfo").join("phones");
q.where(cb.equal(emp.get("contactInfo")
    .get("address")
    .get("zipcode"), "95054"));
q.select(phone.<Vendor>get("vendor"));
```

The following Java Persistence query language query is equivalent:

```
SELECT p.vendor
FROM Employee e JOIN e.contactInfo.phones p
WHERE e.contactInfo.address.zipcode = '95054'
```

Example 4:

In this example, the *photos* attribute corresponds to a map from photo label to filename. The map key is a string, the value an object.

```
CriteriaQuery<Object> q = cb.createQuery();
Root<Item> item = q.from(Item.class);
MapJoin<Item, String, Object> photo = item.joinMap("photos");
q.multiselect(item.get("name"), photo)
    .where(cb.like(photo.key(), "%egret%"));
```

This query is equivalent to the following Java Persistence query language query:

```
SELECT i.name, p
FROM Item i JOIN i.photos p
WHERE KEY(p) LIKE '%egret%'
```

6.8. Query Modification

A *CriteriaQuery*, *CriteriaUpdate*, or *CriteriaDelete* object may be modified, either before or after *Query* or *TypedQuery* objects have been created and executed from it. For example, such modification may entail replacement of the *where* predicate or the *select* list. Modifications may thus result in the same query object "base" being reused for several query instances.

For example, the user might create and execute a query from the following *CriteriaQuery* object:

```
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> c = q.from(Customer.class);

Predicate pred = cb.equal(c.get(Customer_.address).get(Address_.city),"Chicago");

q.select(c);
q.where(pred);
```

The *CriteriaQuery* object might then be modified to reflect a different predicate condition, for example:

```
Predicate pred2 = cb.gt(c.get(Customer_.balanceOwed), 1000);
q.where(pred2);
```

Note, however, that query elements—-in this example, predicate conditions—are dependent on the *CriteriaQuery*, *CriteriaUpdate*, or *CriteriaDelete* instance, and are thus not portably reusable with different instances.

6.9. Query Execution

A criteria query is executed by passing the *CriteriaQuery*, *CriteriaUpdate*, or *CriteriaDelete* object to the *createQuery* method of the *EntityManager* interface to create an executable *TypedQuery* object (or, in the case of *CriteriaUpdate* and *CriteriaDelete*, a *Query* object), which can then be passed to one of the query execution methods of the *TypedQuery* or *Query* interface.

A *CriteriaQuery*, *CriteriaUpdate*, or *CriteriaDelete* object may be further modified after an executable query object has been created from it. The modification of the *CriteriaQuery*, *CriteriaUpdate*, or *CriteriaDelete* object does not have any impact on the already created executable query object. If the modified *CriteriaQuery*, *CriteriaUpdate*, or *CriteriaDelete* object is passed to the *createQuery* method, the persistence provider must insure that a new executable query object is created and returned that reflects the semantics of the changed query definition.

CriteriaQuery, *CriteriaUpdate*, and *CriteriaDelete* objects must be serializable. A persistence vendor is required to support the subsequent deserialization of such an object into a separate JVM instance of that vendor's runtime, where both runtime instances have access to any required vendor implementation classes. *CriteriaQuery*, *CriteriaUpdate*, and *CriteriaDelete* objects are not required to be interoperable across vendors.

Chapter 7. Entity Managers and Persistence Contexts

7.1. Persistence Contexts

A persistence context is a set of managed 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.

In Java EE environments, a JTA transaction typically involves calls across multiple components. Such components may often need to access the same persistence context within a single transaction. To facilitate such use of entity managers in Java EE environments, when an entity manager is injected into a component or looked up directly in the JNDI naming context, its persistence context will automatically be propagated with the current JTA transaction, and the EntityManager references that are mapped to the same persistence unit will provide access to this same persistence context within the JTA transaction. This propagation of persistence contexts by the Java EE container avoids the need for the application to pass references to EntityManager instances from one component to another. An entity manager for which the container manages the persistence context in this manner is termed a *container-managed entity manager*. A container-managed entity manager's lifecycle is managed by the Java EE container.

In less common use cases within Java EE environments, applications may need to access a persistence context that is "stand-alone"—i.e. not propagated along with the JTA transaction across the EntityManager references for the given persistence unit. Instead, each instance of creating an entity manager causes a new isolated persistence context to be created that is not accessible through other EntityManager references within the same transaction. These use cases are supported through the *createEntityManager* methods of the *EntityManagerFactory* interface. An entity manager that is used by the application to create and destroy a persistence context in this manner is termed an *application-managed entity manager*. An application-managed entity manager's lifecycle is managed by the application.

Both container-managed entity managers and application-managed entity managers and their persistence contexts are required to be supported in Java EE web containers and EJB containers. Within an EJB environment, container-managed entity managers are typically used.

In Java SE environments and in Java EE application client containers, only application-managed entity managers are required to be [80: Note that the use of JTA is not required to be supported in application client containers.].

7.2. Obtaining an EntityManager

The entity manager for a persistence context is obtained from an entity manager factory.

When container-managed entity managers are used (in Java EE environments), the application does not interact with the entity manager factory. The entity managers are obtained directly through dependency injection or from JNDI, and the container manages interaction with the entity manager factory transparently to the application.

When application-managed entity managers are used, the application must use the entity manager factory to manage the entity manager and persistence context lifecycle.

An entity manager must not be shared among multiple concurrently executing threads, as the entity manager and persistence context are not required to be threadsafe. Entity managers must only be accessed in a single-threaded manner.

7.2.1. Obtaining an Entity Manager in the Java EE Environment

A container-managed entity manager is obtained by the application through dependency injection or through direct lookup of the entity manager in the JNDI namespace. The container manages the persistence context lifecycle and the creation and the closing of the entity manager instance transparently to the application.

The *PersistenceContext* annotation is used for entity manager injection. The *type* element specifies whether a transaction-scoped or extended persistence context is to be used, as described in Section 7.6. The *synchronization* element specifies whether the persistence context is always automatically joined to the current transaction (the default) or is not joined to the current transaction unless the *joinTransaction* method is invoked by the application. The *unitName* element may optionally be specified to designate the persistence unit whose entity manager factory is used by the container. The semantics of the persistence context synchronization type are further described in [a11797>. Section <<a13887] provides further information about the *unitName* element.

For example,

```
@PersistenceContext
EntityManager em;

@PersistenceContext(type=PersistenceContextType.EXTENDED)
EntityManager orderEM;
```

The JNDI lookup of an entity manager is illustrated below:

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

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

        // ...
    }
}
```

7.2.2. Obtaining an Application-managed Entity Manager

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

The *EntityManagerFactory* API used to obtain an application-managed entity manager is the same independent of whether this API is used in Java EE or Java SE environments.

7.3. Obtaining an Entity Manager Factory

The *EntityManagerFactory* interface is used by the application to create an application-managed entity manager [81: It may also be used internally by the Java EE container. See Section 7.9.].

Each 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. [82: This may be the case when using multiple databases, since in a typical configuration a single entity manager only communicates with a single database. There is only one entity manager factory per persistence unit, however.]

Methods of the *EntityManagerFactory* interface are threadsafe.

7.3.1. Obtaining an Entity Manager Factory in a Java EE Container

Within a Java EE environment, an entity manager factory can be injected using the *PersistenceUnit* annotation or obtained through JNDI lookup. The *unitName* element may optionally be specified to designate the persistence unit whose entity manager factory is used. (See Section 10.5.2).

For example,

```
@PersistenceUnit
EntityManagerFactory emf;
```

7.3.2. Obtaining an Entity Manager Factory in a Java SE Environment

Outside a Java EE 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, described in Section 9.7.

For example,

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

7.4. EntityManagerFactory Interface

The *EntityManagerFactory* interface is used by the application to obtain an application-managed entity manager. When the application has finished using the entity manager factory, and/or at application shutdown, the application should close the entity manager factory. Once an entity manager factory has been closed, all its entity managers are considered to be in the closed state.

The *EntityManagerFactory* interface provides access to information and services that are global to the persistence unit. This includes access to the second level cache that is maintained by the persistence provider and to the *PersistenceUnitUtil* interface. The *Cache* interface is described in Section 7.10; the *PersistenceUnitUtil* interface in Section 7.11.

```
package jakarta.persistence;

import java.util.Map;
import jakarta.persistence.metamodel.Metamodel;
import jakarta.persistence.criteria.CriteriaBuilder;

/**
 * Interface used to interact with the entity manager factory
 * for the persistence unit.
 *
 * <p>When the application has finished using the entity manager
 * factory, and/or at application shutdown, the application should
 * close the entity manager factory.  Once an
 * <code>EntityManagerFactory</code> has been closed, all its entity managers
 * are considered to be in the closed state.
```

```
 *
 * @since 1.0
 */
public interface EntityManagerFactory {

    /**
     * Create a new application-managed <code>EntityManager</code>.
     * This method returns a new <code>EntityManager</code> instance each time
     * it is invoked.
     * The <code>isOpen</code> method will return true on the returned instance.
     * @return entity manager instance
     * @throws IllegalStateException if the entity manager factory
     * has been closed
     */
    public EntityManager createEntityManager();

    /**
     * Create a new application-managed <code>EntityManager</code> with the
     * specified Map of properties.
     * This method returns a new <code>EntityManager</code> instance each time
     * it is invoked.
     * The <code>isOpen</code> method will return true on the returned instance.
     * @param map properties for entity manager
     * @return entity manager instance
     * @throws IllegalStateException if the entity manager factory
     * has been closed
     */
    public EntityManager createEntityManager(Map map);

    /**
     * Create a new JTA application-managed <code>EntityManager</code> with the
     * specified synchronization type.
     * This method returns a new <code>EntityManager</code> instance each time
     * it is invoked.
     * The <code>isOpen</code> method will return true on the returned instance.
     * @param synchronizationType  how and when the entity manager should be
     * synchronized with the current JTA transaction
     * @return entity manager instance
     * @throws IllegalStateException if the entity manager factory
     * has been configured for resource-local entity managers or is closed
     *
     * @since 2.1
     */
    public EntityManager createEntityManager(SynchronizationType synchronizationType);

    /**
     * Create a new JTA application-managed <code>EntityManager</code> with the
     * specified synchronization type and map of properties.
```

```
     * This method returns a new <code>EntityManager</code> instance each time
     * it is invoked.
     * The <code>isOpen</code> method will return true on the returned instance.
     * @param synchronizationType  how and when the entity manager should be
     * synchronized with the current JTA transaction
     * @param map properties for entity manager
     * @return entity manager instance
     * @throws IllegalStateException if the entity manager factory
     * has been configured for resource-local entity managers or is closed
     *
     * @since 2.1
     */
    public EntityManager createEntityManager(SynchronizationType synchronizationType, Map
map);

    /**
     * Return an instance of <code>CriteriaBuilder</code> for the creation of
     * <code>CriteriaQuery</code> objects.
     * @return CriteriaBuilder instance
     * @throws IllegalStateException if the entity manager factory
     * has been closed
     *
     * @since 2.0
     */
    public CriteriaBuilder getCriteriaBuilder();

    /**
     * Return an instance of <code>Metamodel</code> interface for access to the
     * metamodel of the persistence unit.
     * @return Metamodel instance
     * @throws IllegalStateException if the entity manager factory
     * has been closed
     *
     * @since 2.0
     */
    public Metamodel getMetamodel();

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

    /**
     * Close the factory, releasing any resources that it holds.
     * After a factory instance has been closed, all methods invoked
     * on it will throw the <code>IllegalStateException</code>, except
```

```
     * for <code>isOpen</code>, which will return false. Once an
     * <code>EntityManagerFactory</code> has been closed, all its
     * entity managers are considered to be in the closed state.
     * @throws IllegalStateException if the entity manager factory
     * has been closed
     */
    public void close();

    /**
     * Get the properties and associated values that are in effect
     * for the entity manager factory. Changing the contents of the
     * map does not change the configuration in effect.
     * @return properties
     * @throws IllegalStateException if the entity manager factory
     * has been closed
     *
     * @since 2.0
     */
    public Map<String, Object> getProperties();

    /**
     * Access the cache that is associated with the entity manager
     * factory (the "second level cache").
     * @return instance of the <code>Cache</code> interface or null if
     * no cache is in use
     * @throws IllegalStateException if the entity manager factory
     * has been closed
     *
     * @since 2.0
     */
    public Cache getCache();

    /**
     * Return interface providing access to utility methods
     * for the persistence unit.
     * @return <code>PersistenceUnitUtil</code> interface
     * @throws IllegalStateException if the entity manager factory
     * has been closed
     *
     * @since 2.0
     */
    public PersistenceUnitUtil getPersistenceUnitUtil();

    /**
     * Define the query, typed query, or stored procedure query as
     * a named query such that future query objects can be created
     * from it using the <code>createNamedQuery</code> or
     * <code>createNamedStoredProcedureQuery</code> method.
```

```
     * <p>Any configuration of the query object (except for actual
     * parameter binding) in effect when the named query is added
     * is retained as part of the named query definition.
     * This includes configuration information such as max results,
     * hints, flush mode, lock mode, result set mapping information,
     * and information about stored procedure parameters.
     * <p>When the query is executed, information that can be set
     * by means of the query APIs can be overridden. Information
     * that is overridden does not affect the named query as
     * registered with the entity manager factory, and thus does
     * not affect subsequent query objects created from it by
     * means of the <code>createNamedQuery</code> or
     * <code>createNamedStoredProcedureQuery</code> method.
     * <p>If a named query of the same name has been previously
     * defined, either statically via metadata or via this method,
     * that query definition is replaced.
     *
     * @param name name for the query
     * @param query Query, TypedQuery, or StoredProcedureQuery object
     *
     * @since 2.1
     */
    public void addNamedQuery(String name, Query query);

    /**
     * Return an object of the specified type to allow access to the
     * provider-specific API. If the provider's EntityManagerFactory
     * implementation does not support the specified class, the
     * PersistenceException is thrown.
     * @param cls the class of the object to be returned. This is
     * normally either the underlying EntityManagerFactory
     * implementation class or an interface that it implements.
     * @return an instance of the specified class
     * @throws PersistenceException if the provider does not
     * support the call
     * @since 2.1
     */
    public <T> T unwrap(Class<T> cls);

    /**
     * Add a named copy of the EntityGraph to the
     * EntityManagerFactory.  If an entity graph with the same name
     * already exists, it is replaced.
     * @param graphName  name for the entity graph
     * @param entityGraph  entity graph
     * @since 2.1
     */
    public <T> void addNamedEntityGraph(String graphName, EntityGraph<T> entityGraph);
```

```
}
```

Any number of vendor-specific properties may be included in the map passed to the *createEntityManager* methods. Properties that are not recognized by a vendor must be ignored.

Note that the policies of the installation environment may restrict some information from being made available through the *EntityManagerFactory getProperties* method (for example, JDBC user, password, URL).

Vendors should use vendor namespaces for properties (e.g., *com.acme.persistence.logging*). Entries that make use of the namespace *javax.persistence* and its subnamespaces must not be used for vendor-specific information. The namespace *javax.persistence* is reserved for use by this specification.

7.5. Controlling Transactions

Depending on the transactional type of the entity manager, transactions involving EntityManager operations may be controlled either through JTA or through use of the 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 whose underlying transactions are controlled through JTA is termed a *JTA entity manager*.

An entity manager whose underlying transactions are controlled by the application through the *EntityTransaction* API is termed a *resource-local entity manager*.

A container-managed entity manager must be a JTA entity manager. JTA entity managers are only specified for use in Java EE containers.

An application-managed entity manager may be either a JTA entity manager or a resource-local 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. See sections Section 8.2.1.2 and Section 9.1.

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

7.5.1. JTA EntityManagers

An entity manager whose transactions are controlled through JTA is a JTA entity manager. In general, 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.

7.5.2. Resource-local EntityManagers

An entity manager whose transactions are controlled by the application through the *EntityTransaction* API is 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.

7.5.3. The EntityTransaction Interface

The *EntityTransaction* interface is used to control resource transactions on resource-local entity managers. The *EntityManager.getTransaction()* method returns an instance of the *EntityTransaction* interface.

When a resource-local entity manager is used, and the persistence provider runtime throws an exception defined to cause transaction rollback, the persistence provider must mark the transaction for rollback.

If the *EntityTransaction.commit* operation fails, the persistence provider must roll back the transaction.

```
package jakarta.persistence;

/**
 * Interface used to control transactions on resource-local entity
 * managers.  The {@link EntityManager#getTransaction
 * EntityManager.getTransaction()} method returns the
 * <code>EntityTransaction</code> interface.
 *
 * @since 1.0
 */
public interface EntityTransaction {

    /**
     * Start a resource transaction.
     * @throws IllegalStateException if <code>isActive()</code> is true
     */
    public void begin();

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

    /**
```

```
     * Roll back the current resource transaction.
     * @throws IllegalStateException if <code>isActive()</code> is false
     * @throws PersistenceException if an unexpected error
     *         condition is encountered
     */
    public void rollback();

    /**
     * Mark the current resource transaction so that the only
     * possible outcome of the transaction is for the transaction
     * to be rolled back.
     * @throws IllegalStateException if <code>isActive()</code> is false
     */
    public void setRollbackOnly();

    /**
     * Determine whether the current resource transaction has been
     * marked for rollback.
     * @return boolean indicating whether the transaction has been
     *         marked for rollback
     * @throws IllegalStateException if <code>isActive()</code> is false
     */
    public boolean getRollbackOnly();

    /**
     * Indicate whether a resource transaction is in progress.
     * @return boolean indicating whether transaction is
     *         in progress
     * @throws PersistenceException if an unexpected error
     *         condition is encountered
     */
    public boolean isActive();
}
```

7.5.4. Example

The following example illustrates the creation of an entity manager factory in a Java SE environment, and its use in creating and using a resource-local entity manager.

```
import javax.persistence.*;

public class PasswordChanger {
    public static void main (String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("Order");
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();

        User user = (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();
    }
}
```

7.6. Container-managed Persistence Contexts

When a container-managed entity manager is used, the lifecycle of the persistence context is always managed automatically, transparently to the application, and the persistence context is propagated with the JTA transaction.

A container-managed persistence context may be defined to have either a lifetime that is scoped to a single transaction or an extended lifetime that spans multiple transactions, depending on the *PersistenceContextType* that is specified when its entity manager is created. This specification refers to such persistence contexts as *transaction-scoped persistence contexts* and *extended persistence contexts* respectively.

The lifetime of the persistence context is declared using the *PersistenceContext* annotation or the *persistence-context-ref* deployment descriptor element. By default, a transaction-scoped persistence context is used.

Sections Section 7.6.2 and Section 7.6.3 describe transaction-scoped and extended persistence contexts in the absence of persistence context propagation. Persistence context propagation is described in Section 7.6.4.

Persistence contexts are always associated with an entity manager factory. In the following sections,

"the persistence context" should be understood to mean "the persistence context associated with a particular entity manager factory".

7.6.1. Persistence Context Synchronization Type

By default, a container-managed persistence context is of type *SynchronizationType.SYNCHRONIZED*. Such a persistence context is automatically joined to the current JTA transaction, and updates made to the persistence context are propagated to the underlying resource manager.

A container-managed persistence context may be specified to be of type *SynchronizationType.UNSYNCHRONIZED*. A persistence context of type *SynchronizationType.UNSYNCHRONIZED* is not enlisted in any JTA transaction unless explicitly joined to that transaction by the application. A persistence context of type *SynchronizationType.UNSYNCHRONIZED* is enlisted in a JTA transaction and registered for subsequent transaction notifications against that transaction by the invocation of the *EntityManager joinTransaction* method. The persistence context remains joined to the transaction until the transaction commits or rolls back. After the transaction commits or rolls back, the persistence context will not be joined to any subsequent transaction unless the *joinTransaction* method is invoked in the scope of that subsequent transaction.

A persistence context of type *SynchronizationType.UNSYNCHRONIZED* must not be flushed to the database unless it is joined to a transaction. The application's use of queries with pessimistic locks, bulk update or delete queries, etc. result in the provider throwing the *TransactionRequiredException*. After the persistence context has been joined to the JTA transaction, these operations are again allowed.

The application is permitted to invoke the persist, merge, remove, and refresh entity lifecycle operations on an entity manager of type *SynchronizationType.UNSYNCHRONIZED* independent of whether the persistence context is joined to the current transaction. After the persistence context has been joined to a transaction, changes in a persistence context can be flushed to the database either explicitly by the application or by the provider. If the *flush* method is not explicitly invoked, the persistence provider may defer flushing until commit time depending on the operations invoked and the flush mode setting in effect.

If an extended persistence context of type *SynchronizationType.UNSYNCHRONIZED* has not been joined to the current JTA transaction, rollback of the JTA transaction will have no effect upon the persistence context. In general, it is recommended that a non-JTA datasource be specified for use by the persistence provider for a persistence context of type *SynchronizationType.UNSYNCHRONIZED* that has not been joined to a JTA transaction in order to alleviate the risk of integrating uncommitted changes into the persistence context in the event that the transaction is later rolled back.

If a persistence context of type *SynchronizationType.UNSYNCHRONIZED* has been joined to the JTA transaction, transaction rollback will cause the persistence context to be cleared and all pre-existing managed and removed instances to become detached. (See Section 3.3.3.)

When a JTA transaction exists, a persistence context of type *SynchronizationType.UNSYNCHRONIZED* is

propagated with that transaction according to the rules in Section 7.6.4.1 regardless of whether the persistence context has been joined to that transaction.

7.6.2. Container-managed Transaction-scoped Persistence Context

The application can obtain a container-managed entity manager with transaction-scoped persistence context by injection or direct lookup in the JNDI namespace. The persistence context type for the entity manager is defaulted or defined as *PersistenceContextType.TRANSACTION*.

A new persistence context begins when the container-managed entity manager is invoked [83: Specifically, when one of the methods of the EntityManager interface 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 JTA transaction. This association of the persistence context with the JTA transaction is independent of the synchronization type of the persistence context and whether the persistence context has been joined to the transaction.

The persistence context ends when the associated JTA transaction commits or rolls back, and all entities that were managed by the EntityManager become detached. [84: Note that this applies to a transaction-scoped persistence context of type SynchronizationType.UNSYNCHRONIZED that has not been joined to the transaction as well.]

If the entity manager is invoked outside the scope of a transaction, any entities loaded from the database will immediately become detached at the end of the method call.

7.6.3. Container-managed Extended Persistence Context

A container-managed extended persistence context can only be initiated within the scope of a stateful session bean. It exists from the point at which the stateful session bean that declares a dependency on an entity manager of type *PersistenceContextType.EXTENDED* is created, and is said to be *bound* to the stateful session bean. The dependency on the extended persistence context is declared by means of the *PersistenceContext* annotation or *persistence-context-ref* deployment descriptor element. The association of the extended persistence context with the JTA transaction is independent of the synchronization type of the persistence context and whether the persistence context has been joined to the transaction.

The persistence context is closed by the container when the *@Remove* method of the stateful session bean completes (or the stateful session bean instance is otherwise destroyed).

7.6.3.1. Inheritance of Extended Persistence Context

If a stateful session bean instantiates a stateful session bean (executing in the same EJB container instance) which also has such an extended persistence context with the same synchronization type, the extended persistence context of the first stateful session bean is inherited by the second stateful session bean and bound to it, and this rule recursively applies—independently of whether transactions are active or not at the point of the creation of the stateful session beans. If the stateful session beans differ in declared synchronization type, the EJBException is thrown by the container.

If the persistence context has been inherited by any stateful session beans, the container does not close the persistence context until all such stateful session beans have been removed or otherwise destroyed.

7.6.4. Persistence Context Propagation

As described in Section 7.1, a single persistence context may correspond to one or more JTA entity manager instances (all associated with the same entity manager factory [85: Entity manager instances obtained from different entity manager factories never share the same persistence context.]).

The persistence context is propagated across the entity manager instances as the JTA transaction is propagated. A persistence context of type *SynchronizationType.UNSYNCHRONIZED* is propagated with the JTA transaction regardless of whether it has been joined to the transaction.

Propagation of persistence contexts only applies within a local environment. Persistence contexts are not propagated to remote tiers.

7.6.4.1. Requirements for Persistence Context Propagation

Persistence contexts are propagated by the container across component invocations as follows.

If a component is called and there is no JTA transaction or the JTA transaction is not propagated, the persistence context is not propagated.

- If an entity manager is then invoked from within the component:
  - Invocation of an entity manager defined with *PersistenceContextType.TRANSACTION* will result in use of a new persistence context (as described in Section 7.6.2).
  - Invocation of an entity manager defined with *PersistenceContextType.EXTENDED* will result in the use of the existing extended persistence context bound to that component.
  - If the entity manager is invoked within a JTA transaction, the persistence context will be associated with the JTA transaction.

If a component is called and the JTA transaction is propagated into that component:

- If the component is a stateful session bean to which an extended persistence context has been bound and there is a different persistence context associated with the JTA transaction, an *EJBException* is thrown by the container.
- If there is a persistence context of type *SynchronizationType.UNSYNCHRONIZED* associated with the JTA transaction and the target component specifies a persistence context of type *SynchronizationType.SYNCHRONIZED*, the *IllegalStateException* is thrown by the container.
- Otherwise, if there is a persistence context associated with the JTA transaction, that persistence context is propagated and used.

Note that a component with a persistence context of type *SynchronizationType.UNSYNCHRONIZED* may be called by a component propagating either a persistence context of type *SynchronizationType.UNSYNCHRONIZED* or a persistence context of type *SynchronizationType.SYNCHRONIZED* into it.

7.6.5. Examples

7.6.5.1. Container-managed Transaction-scoped Persistence Context

```
@Stateless
public class ShoppingCartImpl implements ShoppingCart {
    @PersistenceContext
    EntityManager em;

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

    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;
    }
}
```

7.6.5.2. Container-managed Extended Persistence Context

```
/*
 * An extended transaction context is used. The entities remain
 * managed in the persistence context across multiple transactions.
 */
@Stateful
@Transaction(REQUIRES_NEW)
public class ShoppingCartImpl implements ShoppingCart {
    @PersistenceContext(type = EXTENDED)
    EntityManager em;

    private Order order;
    private Product product;

    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);
        em.persist(li);
        return li;
    }
}
```

7.7. Application-managed Persistence Contexts

When an application-managed entity manager is used, the application interacts directly with the persistence provider's entity manager factory to manage the entity manager lifecycle and to obtain and destroy persistence contexts.

All such application-managed persistence contexts are extended in scope, and can span multiple transactions.

The *EntityManagerFactory . createEntityManager* method and the *EntityManager close* and *isOpen* methods are used to manage the lifecycle of an application-managed entity manager and its associated persistence context.

The extended persistence context exists from the point at which the entity manager has been created using *EntityManagerFactory.createEntityManager* until the entity manager is closed by means of *EntityManager.close*.

An extended persistence context obtained from the application-managed entity manager is a stand-alone persistence context—it is not propagated with the transaction.

When a JTA application-managed entity manager is used, an application-managed persistence context may be specified to be of type *SynchronizationType.UNSYNCHRONIZED*. A persistence context of type *SynchronizationType.UNSYNCHRONIZED* is not enlisted in any JTA transaction unless explicitly joined to that transaction by the application. A persistence context of type *SynchronizationType.UNSYNCHRONIZED* is enlisted in a JTA transaction and registered for subsequent transaction notifications against that transaction by the invocation of the *EntityManager joinTransaction* method. The persistence context remains joined to the transaction until the transaction commits or rolls back. After the transaction commits or rolls back, the persistence context will not be joined to any subsequent transaction unless the *joinTransaction* method is invoked in the scope of that subsequent transaction.

When a JTA application-managed entity manager is used, if the entity manager is created outside the scope of the current JTA transaction, it is the responsibility of the application to join the entity manager to the transaction (if desired) by calling *EntityManager.joinTransaction*. If the entity manager is created outside the scope of a JTA transaction, it is not joined to the transaction unless *EntityManager.joinTransaction* is called.

The *EntityManager.close* method closes an entity manager to release its persistence context and other resources. After calling *close*, the application must not invoke any further methods on the *EntityManager* instance except for *getTransaction* and *isOpen*, or the *IllegalStateException* will be thrown. If the *close* method is invoked when a transaction is active, the persistence context remains managed until the transaction completes.

The *EntityManager.isOpen* method indicates whether the entity manager is open. The *isOpen* method returns true until the entity manager has been closed.

7.7.1. Examples

7.7.1.1. Application-managed Persistence Context used in Stateless Session Bean

```
/*
 * Container-managed transaction demarcation is used.
 * The session bean creates and closes an entity manager
 * in each business method.
 */
@Stateless
public class ShoppingCartImpl implements ShoppingCart {
    @PersistenceUnit
    private EntityManagerFactory emf;

    public Order getOrder(Long id) {
        EntityManager em = emf.createEntityManager();
        Order order = em.find(Order.class, id);
        order.getLineItems();
        em.close();
        return order;
    }

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

    public LineItem createLineItem(Order order, Product product, int quantity) {
        EntityManager em = emf.createEntityManager();
        LineItem li = new LineItem(order, product, quantity);
        order.getLineItems().add(li);
        em.persist(li);
        em.close();
        return li; // remains managed until JTA transaction ends
    }
}
```

7.7.1.2. Application-managed Persistence Context used in Stateless Session Bean

```
/*
 * Container-managed transaction demarcation is used.
 * The session bean creates entity manager in PostConstruct
 * method and clears persistence context at the end of each
 * business method.
 */
```

```
@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) {
        Order order = em.find(Order.class, id);
        order.getLineItems();
        em.clear(); // entities are detached
        return order;
    }

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

    public LineItem createLineItem(Order order, Product product, int quantity) {
        em.joinTransaction();
        LineItem li = new LineItem(order, product, quantity);
        order.getLineItems().add(li);
        em.persist(li);
        // persistence context is flushed to database;
        // all updates will be committed to database on tx commit
        em.flush();
        // entities in persistence context are detached
        em.clear();
        return li;
    }

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

7.7.1.3. Application-managed Persistence Context used in Stateful Session Bean

```
/*
 * Container-managed transaction demarcation is used.
 * Entities remain managed until the entity manager is closed.
 */
@Stateful
public class ShoppingCartImpl implements ShoppingCart {
    @PersistenceUnit
    private EntityManagerFactory emf;

    private EntityManager em;

    private Order order;

    private Product product;

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

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

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

7.7.1.4. Application-managed Persistence Context with Resource Transaction

```
// Usage in an ordinary Java class
public class ShoppingImpl {
    private EntityManager em;
    private EntityManagerFactory emf;

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

    private Order order;
    private Product product;

    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.persist(li);
        em.getTransaction().commit();
        return li;
    }

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

7.8. Requirements on the Container

7.8.1. Application-managed Persistence Contexts

When application-managed persistence contexts are used, the container must instantiate the entity manager factory and expose it to the application via JNDI. The container might use internal APIs to create the entity manager factory, or it might use the *PersistenceProvider.createContainerEntityManagerFactory* method. However, the container is required to support third-party persistence providers, and in this case the container must use the *PersistenceProvider.createContainerEntityManagerFactory* method to create the entity manager factory and the *EntityManagerFactory.close* method to destroy the entity manager factory prior to shutdown (if it has not been previously closed by the application).

7.8.2. Container Managed Persistence Contexts

The container is responsible for managing the lifecycle of container-managed persistence contexts, for injecting *EntityManager* references into web components and session bean and message-driven bean components, and for making *EntityManager* references available to direct lookups in JNDI.

When operating with a third-party persistence provider, the container uses the contracts defined in Section 7.9 to create and destroy container-managed 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.

If a persistence context is already associated with a JTA transaction, the container uses that persistence context for subsequent invocations within the scope of that transaction, according to the semantics for persistence context propagation defined in Section 7.6.4.

7.9. Runtime Contracts between the Container and Persistence Provider

This section describes contracts between the container and the persistence provider for the pluggability of third-party persistence providers. Containers are required to support these pluggability contracts. [86: It is not required that these contracts be used when a third-party persistence provider is not used: the container might use these same APIs or its might use its own internal APIs.]

7.9.1. Container Responsibilities

For the management of a transaction-scoped persistence context, if there is no EntityManager already associated with the JTA transaction:

- The container creates a new entity manager by calling *EntityManagerFactory.createEntityManager* when the first invocation of an entity manager with *PersistenceContextType.TRANSACTION* occurs within the scope of a business method executing in the JTA transaction.
- After the JTA transaction has completed (either by transaction commit or rollback), the container

closes the entity manager by calling *EntityManager.close*. [87: The container may choose to pool EntityManagers: it instead of creating and closing in each case, it may acquire one from its pool and call *clear()* on it.] Note that the JTA transaction may rollback in a background thread (e.g., as a result of transaction timeout), in which case the container should arrange for the entity manager to be closed but the *EntityManager.close* method should not be concurrently invoked while the application is in an EntityManager invocation.

The container must throw the *TransactionRequiredException* if a transaction-scoped persistence context is used and the *EntityManager persist*, *remove*, *merge*, or *refresh* method is invoked when no transaction is active.

For stateful session beans with extended persistence contexts:

- The container creates an entity manager by calling *EntityManagerFactory.createEntityManager* when a stateful session bean is created that declares a dependency on an entity manager with *PersistenceContextType.EXTENDED*. (See Section 7.6.3).
- The container closes the entity manager by calling *EntityManager.close* after the stateful session bean and all other stateful session beans that have inherited the same persistence context as the entity manager have been removed.
- When a business method of the stateful session bean is invoked, if the stateful session bean uses container managed transaction demarcation, and the entity manager is not already associated with the current JTA transaction, the container associates the entity manager with the current JTA transaction and, if the persistence context is of type *SynchronizationType.SYNCHRONIZED*, the container calls *EntityManager.joinTransaction*. If there is a different persistence context already associated with the JTA transaction, the container throws the *EJBException*.
- When a business method of the stateful session bean is invoked, if the stateful session bean uses bean managed transaction demarcation and a UserTransaction is begun within the method, the container associates the persistence context with the JTA transaction and, if the persistence context is of type *SynchronizationType.SYNCHRONIZED*, the container calls *EntityManager.joinTransaction*.

The container must throw the *IllegalStateException* if the application calls *EntityManager.close* on a container-managed entity manager.

When the container creates an entity manager, it may pass a map of properties to the persistence provider by using the *EntityManagerFactory.createEntityManager(Map map)* method. If properties have been specified in the *PersistenceContext* annotation or the *persistence-context-ref* deployment descriptor element, this method must be used and the map must include the specified properties.

If the application invokes *EntityManager.unwrap(Class<T> cls)*, and the container cannot satisfy the request, the container must delegate the *unwrap* invocation to the provider's entity manager instance.

7.9.2. Provider Responsibilities

The Provider has no knowledge of the distinction between transaction-scoped and extended persistence contexts. It provides entity managers to the container when requested and registers for

transaction synchronization notifications.

- When *EntityManagerFactory.createEntityManager* is invoked, the provider must create and return a new entity manager. If a JTA transaction is active and the persistence context is of type *SynchronizationType.SYNCHRONIZED*, the provider must register for synchronization notifications against the JTA transaction.
- When *EntityManager.joinTransaction* is invoked, the provider must register for synchronization notifications against the current JTA transaction if a previous *joinTransaction* invocation for the transaction has not already been processed.
- When the JTA transaction commits, if the persistence context is of type *SynchronizationType.SYNCHRONIZED* or has otherwise been joined to the transaction, the provider must flush all modified entity state to the database.
- When the JTA transaction rolls back, the provider must detach all managed entities if the persistence context is of type *SynchronizationType.SYNCHRONIZED* or has otherwise been joined to the transaction. Note that the JTA transaction may rollback in a background thread (e.g., as a result of transaction timeout), in which case the provider should arrange for the managed entities to be detached from the persistence context but not concurrently while the application is in an EntityManager invocation.
- When the provider throws an exception defined to cause transaction rollback, the provider must mark the transaction for rollback if the persistence context is of type *SynchronizationType.SYNCHRONIZED* or has otherwise been joined to the transaction.
- When *EntityManager.close* is invoked, the provider should release all resources that it may have allocated after any outstanding transactions involving the entity manager have completed. If the entity manager was already in a closed state, the provider must throw the *IllegalStateException*.
- When *EntityManager.clear* is invoked, the provider must detach all managed entities.

7.10. Cache Interface

The *Cache* interface provides basic functionality over the persistence provider's second level cache, if used.

```
package jakarta.persistence;

/**
 * Interface used to interact with the second-level cache.
 * If a cache is not in use, the methods of this interface have
 * no effect, except for <code>contains</code>, which returns false.
 *
 * @since 2.0
 */
public interface Cache {
```

```
    /**
     * Whether the cache contains data for the given entity.
     * @param cls  entity class
     * @param primaryKey  primary key
     * @return boolean indicating whether the entity is in the cache
     */
    public boolean contains(Class cls, Object primaryKey);

    /**
     * Remove the data for the given entity from the cache.
     * @param cls  entity class
     * @param primaryKey  primary key
     */
    public void evict(Class cls, Object primaryKey);

    /**
     * Remove the data for entities of the specified class (and its
     * subclasses) from the cache.
     * @param cls  entity class
     */
    public void evict(Class cls);

    /**
     * Clear the cache.
     */
    public void evictAll();

    /**
     * Return an object of the specified type to allow access to the
     * provider-specific API.  If the provider's Cache
     * implementation does not support the specified class, the
     * PersistenceException is thrown.
     * @param cls  the class of the object to be returned.  This is
     * normally either the underlying Cache implementation
     * class or an interface that it implements.
     * @return an instance of the specified class
     * @throws PersistenceException if the provider does not
     * support the call
     * @since 2.1
     */
    public <T> T unwrap(Class<T> cls);
}
```

7.11. PersistenceUnitUtil Interface

The *PersistenceUnitUtil* interface provides access to utility methods that can be invoked on entities

associated with the persistence unit. The behavior is undefined if these methods are invoked on an entity instance that is not associated with the persistence unit from whose entity manager factory this interface has been obtained.

```
package jakarta.persistence;

/**
 * Utility interface between the application and the persistence
 * provider managing the persistence unit.
 *
 * <p>The methods of this interface should only be invoked on entity
 * instances obtained from or managed by entity managers for this
 * persistence unit or on new entity instances.
 *
 * @since 2.0
 */
public interface PersistenceUnitUtil extends PersistenceUtil {

    /**
     * Determine the load state of a given persistent attribute
     * of an entity belonging to the persistence unit.
     * @param entity  entity instance containing the attribute
     * @param attributeName name of attribute whose load state is
     *        to be determined
     * @return false if entity's state has not been loaded or if
     *         the attribute state has not been loaded, else true
     */
    public boolean isLoaded(Object entity, String attributeName);

    /**
     * Determine the load state of an entity belonging to the
     * persistence unit.  This method can be used to determine the
     * load state of an entity passed as a reference.  An entity is
     * considered loaded if all attributes for which
     * <code>FetchType.EAGER</code> has been specified have been
     * loaded.
     * <p> The <code>isLoaded(Object, String)</code> method
     * should be used to determine the load state of an attribute.
     * Not doing so might lead to unintended loading of state.
     * @param entity   entity instance whose load state is to be determined
     * @return false if the entity has not been loaded, else true
     */
    public boolean isLoaded(Object entity);

    /**
     *  Return the id of the entity.
     *  A generated id is not guaranteed to be available until after
```

```
     *  the database insert has occurred.
     *  Returns null if the entity does not yet have an id.
     *  @param entity  entity instance
     *  @return id of the entity
     *  @throws IllegalArgumentException if the object is found not
     *          to be an entity
     */
    public Object getIdentifier(Object entity);
}
```

Chapter 8. Entity Packaging

This chapter describes the packaging of persistence units.

8.1. Persistence Unit

A persistence unit is a logical grouping that includes:

- An entity manager factory and its entity managers, together with their configuration information.
- The set of managed classes included in the persistence unit and managed by the entity managers of the entity manager factory.
- Mapping metadata (in the form of metadata annotations and/or XML metadata) that specifies the mapping of the classes to the database.

8.2. Persistence Unit Packaging

Within Java EE environments, an EJB-JAR, WAR, EAR, or application client JAR can define a persistence unit. Any number of persistence units may be defined within these scopes.

A persistence unit may be packaged within one or more jar files contained within a WAR or EAR, as a set of classes within an EJB-JAR file or in the WAR *classes* directory, or as a combination of these as defined below.

A persistence unit is defined by a *persistence.xml* file. The jar file or directory whose *META-INF* directory contains the *persistence.xml* file is termed the *root* of the persistence unit. In Java EE environments, the root of a persistence unit must be one of the following:

- an EJB-JAR file
- the *WEB-INF/classes* directory of a WAR file [88: The root of the persistence unit is the *WEB-INF/classes* directory; the *persistence.xml* file is therefore contained in the *WEB-INF/classes/META-INF* directory.]
- a jar file in the *WEB-INF/lib* directory of
- a WAR file
- a jar file in the EAR library directory
- an application client jar file

It is not required that an EJB-JAR or WAR file containing a persistence unit be packaged in an EAR unless the persistence unit contains persistence classes in addition to those contained within the EJB-JAR or WAR. See Section 8.2.1.6.

Java Persistence 1.0 supported use of a jar file in the root of the EAR as the root of a persistence unit. This use is no longer supported. Portable applications should use the EAR library directory for this case instead. See [6].

A persistence unit must have a name. Only one persistence unit of any given name must be defined within a single EJB-JAR file, within a single WAR file, within a single application client jar, or within an EAR. See Section 8.2.2.

The *persistence.xml* file may be used to designate more than one persistence unit within the same scope.

All persistence classes defined at the level of the Java EE EAR must be accessible to other Java EE components in the application—i.e. loaded by the application classloader—such that if the same entity class is referenced by two different Java EE components (which may be using different persistence units), the referenced class is the same identical class.

In Java SE environments, the metadata mapping files, jar files, and classes described in the following sections can be used. To insure the portability of a Java SE application, it is necessary to explicitly list the managed persistence classes that are included in the persistence unit using the *class* element of the *persistence.xml* file. See Section 8.2.1.6.

8.2.1. persistence.xml file

A *persistence.xml* file defines a persistence unit. The *persistence.xml* file is located in the *META-INF* directory of the root of the persistence unit. It may be used to specify managed persistence classes included in the persistence unit, object/relational mapping information for those classes, scripts for use in schema generation and the bulk loading of data, and other configuration information for the persistence unit and for the entity manager(s) and entity manager factory for the persistence unit. This information may be defined by containment or by reference, as described below.

The object/relational mapping information can take the form of annotations on the managed persistence classes included in the persistence unit, an *orm.xml* file contained in the *META-INF* directory of the root of the persistence unit, one or more XML files on the classpath and referenced from the *persistence.xml* file, or a combination of these.

The managed persistence classes may either be contained within the root of the persistence unit; or they may be specified by reference—i.e., by naming the classes, class archives, or XML mapping files (which in turn reference classes) that are accessible on the application classpath; or they may be specified by some combination of these means. See Section 8.2.1.6.

The root element of the *persistence.xml* file is the *persistence* element. The *persistence* element consists of one or more *persistence-unit* elements.

The *persistence-unit* element consists of the *name* and *transaction-type* attributes and the following sub-elements: *description*, *provider*, *jta-data-source*, *non-jta-data-source*, *mapping-file*, *jar-file*, *class*, *exclude-unlisted-classes*, *shared-cache-mode*, *validation-mode*, and *properties*.

The *name* attribute is required; the other attributes and elements are optional. Their semantics are described in the following subsections.

Examples:

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

```
<persistence>
    <persistence-unit name="OrderManagement2">
        <description>
            This unit manages inventory for auto parts.
            It depends on features provided by the
            com.acme.persistence implementation.
        </description>
        <provider>com.acme.AcmePersistence</provider>
        <jta-data-source>jdbc/MyPartDB</jta-data-source>
        <mapping-file>ormap2.xml</mapping-file>
        <jar-file>MyPartsApp.jar</jar-file>
        <properties>
            <property name="com.acme.persistence.sql-logging" value="on"/>
        </properties>
    </persistence-unit>
</persistence>
```

8.2.1.1. name

The *name* attribute defines the name for the persistence unit. This name may be used to identify a persistence unit referred to by the *PersistenceContext* and *PersistenceUnit* annotations and in the programmatic API for creating an entity manager factory.

8.2.1.2. transaction-type

The *transaction-type* attribute is used to specify whether the entity managers provided by the entity manager factory for the persistence unit must be JTA entity managers or resource-local entity managers. The value of this element is *JTA* or *RESOURCE_LOCAL*. A *transaction-type* of *JTA* assumes that a JTA data source will be provided—either as specified by the *jta-data-source* element or provided by the container. In general, in Java EE environments, a *transaction-type* of *RESOURCE_LOCAL* assumes that a non-JTA datasource will be provided. In a Java EE environment, if this element is not specified, the default is *JTA*. In a Java SE environment, if this element is not specified, the default is *RESOURCE_LOCAL*.

8.2.1.3. description

The *description* element provides optional descriptive information about the persistence unit.

8.2.1.4. provider

The *provider* element specifies the name of the persistence provider's *javax.persistence.spi.PersistenceProvider* class. The *provider* element is optional, but should be specified if the application is dependent upon a particular persistence provider being used.

8.2.1.5. jta-data-source, non-jta-data-source

In Java EE environments, the *jta-data-source* and *non-jta-data-source* elements are used to specify the JNDI name of the JTA and/or non-JTA data source to be used by the persistence provider. If neither is specified, the deployer must specify a JTA data source at deployment or the default JTA data source must be provided by the container, and a JTA EntityManagerFactory will be created to correspond to it.

In Java SE environments, these elements may be used or the data source information may be specified by other means—depending upon the requirements of the provider.

8.2.1.6. mapping-file, jar-file, class, exclude-unlisted-classes

The following classes must be implicitly or explicitly denoted as managed persistence classes to be included within a persistence unit: entity classes; embeddable classes; mapped superclasses; converter classes.

The set of managed persistence classes that are managed by a persistence unit is defined by using one or more of the following: [89: Note that an individual class may be used in more than one persistence unit.]

- Annotated managed persistence classes contained in the root of the persistence unit (unless the *exclude-unlisted-classes* element is specified)
- One or more object/relational mapping XML files
- One or more jar files that will be searched for classes
- An explicit list of classes

The set of entities managed by the persistence unit is the union of these 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 minimum portable level of overriding is at the level of the persistent field or property.

The classes and/or jars that are named as part of a persistence unit must be on the classpath; referencing them from the *persistence.xml* file does not cause them to be placed on the classpath.

All classes must be on the classpath to ensure that entity managers from different persistence units that map the same class will be accessing the same identical class.

Annotated Classes in the Root of the Persistence Unit

All classes contained in the root of the persistence unit are searched for annotated managed persistence classes—classes with the *Entity*, *Embeddable*, *MappedSuperclass*, or *Converter* annotation—and any mapping metadata annotations found on these classes will be processed, or they will be mapped using the mapping annotation defaults. If it is not intended that the annotated persistence classes contained in the root of the persistence unit be included in the persistence unit, the *exclude-unlisted-classes* element must be specified as *true*. The *exclude-unlisted-classes* element is not intended for use in Java SE environments.

Object/relational Mapping Files

An object/relational mapping XML file contains mapping information for the classes listed in it.

An object/relational mapping XML file named *orm.xml* may be specified in the *META-INF* directory in the root of the persistence unit or in the *META-INF* directory of any jar file referenced by the *persistence.xml*. Alternatively, or in addition, one or more mapping files may be referenced by the *mapping-file* elements of the *persistence-unit* element. These mapping files may be present anywhere on the class path.

An *orm.xml* mapping file or other mapping file is loaded as a resource by the persistence provider. If a mapping file is specified, the classes and mapping information specified in the mapping file will be used as described in Chapter 12. If multiple mapping files are specified (possibly including one or more *orm.xml* files), the resulting mappings are obtained by combining the mappings from all of the files. The result is undefined if multiple mapping files (including any *orm.xml* file) 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.

Jar Files

One or more JAR files may be specified using the *jar-file* elements instead of, or in addition to the mapping files specified in the *mapping-file* elements. If specified, these JAR files will be searched for managed persistence 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.

Such JAR files are specified relative to the directory or jar file that _contains_ [90: This semantics applies to persistence.xml files written to the persistence_2_0.xsd or later schema. Due to ambiguity in the Java Persistence 1.0 specification, provider-specific interpretation of the relative references used by this element may apply to earlier versions.] the root of the persistence unit. [91: Persistence providers are encouraged to support this syntax for use in Java SE environments.]

The following examples illustrate the use of the *jar-file* element to reference additional persistence classes. These examples use the convention that a jar file with a name terminating in " *PUnit* " contains the *persistence.xml* file and that a jar file with a name terminating in " *Entities* " contains additional persistence classes.

Example 1:

```
app.ear
    lib/earEntities.jar
    earRootPUnit.jar (with META-INF/persistence.xml)
```

persistence.xml contains:

```
<jar-file>lib/earEntities.jar</jar-file>
```

Example 2:

```
app.ear
    lib/earEntities.jar
    lib/earLibPUnit.jar (with META-INF/persistence.xml)
```

persistence.xml contains:

```
<jar-file>earEntities.jar</jar-file>
```

Example 3:

```
app.ear
    lib/earEntities.jar
    ejbjar.jar (with META-INF/persistence.xml)
```

persistence.xml contains:

```
<jar-file>lib/earEntities.jar</jar-file>
```

Example 4:

```
app.ear
    war1.war
        WEB-INF/lib/warEntities.jar
        WEB-INF/lib/warPUnit.jar (with META-INF/persistence.xml)
```

persistence.xml contains:

```
<jar-file>warEntities.jar</jar-file>
```

Example 5:

```
app.ear
    war2.war
        WEB-INF/lib/warEntities.jar
        WEB-INF/classes/META-INF/persistence.xml
```

persistence.xml contains:

```
<jar-file>lib/warEntities.jar</jar-file>
```

Example 6:

```
app.ear
    lib/earEntities.jar
    war2.war
        WEB-INF/classes/META-INF/persistence.xml
```

persistence.xml contains:

```
<jar-file>../../lib/earEntities.jar</jar-file>
```

Example 7:

```
app.ear
    lib/earEntities.jar
    war1.war
        WEB-INF/lib/warPUnit.jar (with META-INF/persistence.xml)
```

persistence.xml contains:

```
<jar-file>../../../lib/earEntities.jar</jar-file>
```

List of Managed Classes

A list of named managed persistence classes—entity classes, embeddable classes, mapped superclasses, and converter classes—may 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. The *class* element is used to list a managed persistence class.

A list of all named managed persistence classes must be specified in Java SE environments to insure portability. Portable Java SE applications should not rely on the other mechanisms described here to specify the managed persistence classes of a persistence unit. Persistence providers may 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 in Java SE environments.

8.2.1.7. shared-cache-mode

The *shared-cache-mode* element determines whether second-level caching is in effect for the persistence unit. See Section 3.9.1.

8.2.1.8. validation-mode

The *validation-mode* element determines whether automatic lifecycle event time validation is in effect. See Section 3.6.1.1.

8.2.1.9. properties

The *properties* element is used to specify both standard and vendor-specific properties and hints that apply to the persistence unit and its entity manager factory configuration.

The following properties and hints defined by this specification are intended for use in both Java EE and Java SE environments:

- *javax.persistence.lock.timeout* — value in milliseconds for pessimistic lock timeout. This is a hint only.
- *javax.persistence.query.timeout* — value in milliseconds for query timeout. This is a hint only.
- *javax.persistence.validation.group.pre-persist* — groups that are targeted for validation upon the pre-persist event (overrides the default behavior).
- *javax.persistence.validation.group.pre-update* — groups that are targeted for validation upon the pre-update event (overrides the default behavior).
- *javax.persistence.validation.group.pre-remove* — groups that are targeted for validation upon the pre-remove event (overrides the default behavior).

The following properties defined by this specification are intended for use in Java SE environments.

- *javax.persistence.jdbc.driver* — fully qualified name of the driver class
- *javax.persistence.jdbc.url* — driver-specific URL
- *javax.persistence.jdbc.user* — username used by database connection
- *javax.persistence.jdbc.password* — password for database connection validation

Scripts for use in schema generation may be specified using the *javax.persistence.schema-generation.create-script-source* and *javax.persistence.schema-generation.drop-script-source* properties. A script to specify SQL for the bulk loading of data may be specified by the *javax.persistence.sql-load-script-source* property. These properties are intended for use in both Java EE and Java SE environments:

- *javax.persistence.schema-generation.create-script-source — name of a script packaged as part of the persistence application or a string corresponding to a file URL string that designates a script.*
- *javax.persistence.schema-generation.drop-script-source — name of a script packaged as part of the persistence application or a string corresponding to a file URL string that designates a script.*
- *javax.persistence.sql-load-script-source — name of a script packaged as part of the persistence unit or a string corresponding to a file URL string that designates a script.*

When scripts are packaged as part of the persistence application, these properties must specify locations relative to the root of the persistence unit. When scripts are provided externally (or when schema generation is to occur into script files, as described below), strings corresponding to file URLs must be specified. In Java EE environments, such file URL specifications must be absolute paths (not relative). In Java EE environments, all source and target file locations must be accessible to the application server deploying the persistence unit.

In general, it is expected that schema generation will be initiated by means of the APIs described in Section 9.4. However, schema generation actions may also be specified by means of the following properties used in the *persistence.xml* file.

- *javax.persistence.schema-generation.database.action*
  The *javax.persistence.schema-generation.database.action* property specifies the action to be taken by the persistence provider with regard to the database artifacts. The values for this property are *none*, *create*, *drop-and-create*, *drop*. If this property is not specified, it is assumed that schema generation is not needed or will be initiated by other means, and, by default, no schema generation actions will be taken on the database. (See Section 9.4.)
- *javax.persistence.schema-generation.scripts.action*
  The *javax.persistence.schema-generation.scripts.action* property specifies which scripts are to be generated by the persistence provider. The values for this property are *none*, *create*, *drop-and-create*, *drop*. A script will only be generated if the script target is specified. If this property is not specified, it is assumed that script generation is not needed or will be initiated by other means, and, by default, no scripts will be generated. (See Section 9.4.)

- *javax.persistence.schema-generation.create-source*
  The *javax.persistence.schema-generation.create-source* property specifies whether the creation of database artifacts is to occur on the basis of the object/relational mapping metadata, DDL script, or a combination of the two. The values for this property are *metadata*, *script*, *metadata-then-script*, *script-then-metadata*. If this property is not specified, and a script is specified by the *javax.persistence.schema-generation.create-script-source* property, the script (only) will be used for schema generation; otherwise if this property is not specified, schema generation will occur on the basis of the object/relational mapping metadata (only). The *metadata-then-script* and *script-then-metadata* values specify that a combination of metadata and script is to be used and the order in which this use is to occur. If either of these values is specified and the resulting database actions are not disjoint, the results are undefined and schema generation may fail.
- *javax.persistence.schema-generation.drop-source*
  The *javax.persistence.schema-generation.drop-source* property specifies whether the dropping of database artifacts is to occur on the basis of the object/relational mapping metadata, DDL script, or a combination of the two. The values for this property are *metadata*, *script*, *metadata-then-script*, *script-then-metadata*. If this property is not specified, and a script is specified by the *javax.persistence.schema-generation.drop-script-source* property, the script (only) will be used for the dropping of database artifacts; otherwise if this property is not specified, the dropping of database artifacts will occur on the basis of the object/relational mapping metadata (only). The *metadata-then-script* and *script-then-metadata* values specify that a combination of metadata and script is to be used and the order in which this use is to occur. If either of these values is specified and the resulting database actions are not disjoint, the results are undefined and the dropping of database artifacts may fail.
- *javax.persistence.schema-generation.scripts.create-target*, *javax.persistence.schema-generation.scripts.drop-target*
  If scripts are to be generated, the target locations for the writing of these scripts must be specified. These targets are specified as strings corresponding to file URLs.

If a persistence provider does not recognize a property (other than a property defined by this specification), the provider must ignore it.

Vendors should use vendor namespaces for properties (e.g., *com.acme.persistence.logging*). Entries that make use of the namespace *javax.persistence* and its subnamespaces must not be used for vendor-specific information. The namespace *javax.persistence* is reserved for use by this specification.

8.2.1.10. Examples

The following are sample contents of a *persistence.xml* file.

Example 1:

```
<persistence-unit name="OrderManagement"/>
```

A persistence unit named *OrderManagement* is created.

Any annotated managed persistence classes found in the root of the persistence unit are added to the list of managed persistence classes. If a *META-INF/orm.xml* file exists, any classes referenced by it and mapping information contained in it are used as specified above. Because no provider is specified, the persistence unit is assumed to be portable across providers. Because the transaction type is not specified, JTA is assumed for Java EE environments. The container must provide the data source (it may be specified at application deployment, for example). In Java SE environments, the data source may be specified by other means and a transaction type of *RESOURCE_LOCAL* is assumed.

Example 2:

```
<persistence-unit name="OrderManagement2">
    <mapping-file>mappings.xml</mapping-file>
</persistence-unit>
```

A persistence unit named *OrderManagement2* is created. Any annotated managed persistence classes found in the root of the persistence unit are added to the list of managed persistence classes. The *mappings.xml* resource exists on the classpath and any classes and mapping information contained in it are used as specified above. If a *META-INF/orm.xml* file exists, any classes and mapping information contained in it are used as well. The transaction type, data source, and provider are as described above.

Example 3:

```
<persistence-unit name="OrderManagement3">
    <jar-file>order.jar</jar-file>
    <jar-file>order-supplemental.jar</jar-file>
</persistence-unit>
```

A persistence unit named *OrderManagement3* is created. Any annotated managed persistence classes found in the root of the persistence unit are added to the list of managed persistence classes. If a *META-INF/orm.xml* file exists, any classes and mapping information contained in it are used as specified above. The *order.jar* and *order-supplemental.jar* files are searched for managed persistence classes and any annotated managed persistence classes found in them and/or any classes specified in the *orm.xml* files of these jar files are added. The transaction-type, data source and provider are as described above.

Example 4:

```
<persistence-unit name="OrderManagement4" transaction-type=RESOURCE_LOCAL>
    <non-jta-data-source>java:app/jdbc/MyDB</non-jta-data-source>
    <mapping-file>order-mappings.xml</mapping-file>
    <class>com.acme.Order</class>
    <class>com.acme.Customer</class>
    <class>com.acme.Item</class>
    <exclude-unlisted-classes/>
</persistence-unit>
```

A persistence unit named *OrderManagement4* is created. The file *order-mappings.xml* is read as a resource and any classes referenced by it and mapping information contained in it are used [92: Note that in this example a META-INF/orm.xml file is assumed not to exist.]. The annotated *Order*, *Customer* and *Item* classes are loaded and are added. No (other) classes contained in the root of the persistence unit are added to the list of managed persistence classes. The persistence unit assumed to be portable across providers. A entity manager factory supplying resource-local entity managers will be created. The data source *java:app/jdbc/MyDB* must be used.

Example 5:

```
<persistence-unit name="OrderManagement5">
    <provider>com.acme.AcmePersistence</provider>
    <mapping-file>order1.xml</mapping-file>
    <mapping-file>order2.xml</mapping-file>
    <jar-file>order.jar</jar-file>
    <jar-file>order-supplemental.jar</jar-file>
</persistence-unit>
```

A persistence unit named *OrderManagement5* is created. Any annotated managed persistence classes found in the root of the persistence unit are added to the list of managed classes. The *order1.xml* and *order2.xml* files are read as resources and any classes referenced by them and mapping information contained in them are also used as specified above. The *order.jar* is a jar file on the classpath containing another persistence unit, while *order-supplemental.jar* is just a library of classes. Both of these jar files are searched for annotated managed persistence classes and any annotated managed persistence classes found in them and any classes specified in the *orm.xml* files (if any) of these jar files are added. The provider *com.acme.AcmePersistence* must be used.

Note that the *persistence.xml* file contained in *order.jar* is not used to augment the persistence unit *OrderManagement5* with the classes of the persistence unit whose root is *order.jar*.

8.2.2. Persistence Unit Scope

An EJB-JAR, WAR, application client jar, or EAR can define a persistence unit.

When referencing a persistence unit using the *unitName* annotation element or *persistence-unit-name* deployment descriptor element, the visibility scope of the persistence unit is determined by its point of definition:

- A persistence unit that is defined at the level of an EJB-JAR, WAR, or application client jar is scoped to that EJB-JAR, WAR, or application jar respectively and is visible to the components defined in that jar or war.
- A persistence unit that is defined at the level of the EAR is generally visible to all components in the application. However, if a persistence unit of the same name is defined by an EJB-JAR, WAR, or application jar file within the EAR, the persistence unit of that name defined at EAR level will not be visible to the components defined by that EJB-JAR, WAR, or application jar file unless the persistence unit reference uses the persistence unit name # syntax to specify a path name to disambiguate the reference. When the # syntax is used, the path name is relative to the referencing application component jar file. For example, the syntax *../lib/persistenceUnitRoot.jar#myPersistenceUnit* refers to a persistence unit whose name, as specified in the name element of the *persistence.xml* file, is *myPersistenceUnit* and for which the relative path name of the root of the persistence unit is *../lib/persistenceUnitRoot.jar*. The # syntax may be used with both the *unitName* annotation element or *persistence-unit-name* deployment descriptor element to reference a persistence unit defined at EAR level.

8.3. persistence.xml Schema

This section provides the XML schema for the *persistence.xml* file.

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- persistence.xml schema -->
<xsd:schema targetNamespace="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:persistence="http://xmlns.jcp.org/xml/ns/persistence"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  version="2.2">

  <xsd:annotation>
    <xsd:documentation>
      @(#)persistence_2_2.xsd 2.2  July 17, 2017
    </xsd:documentation>
  </xsd:annotation>

  <xsd:annotation>
    <xsd:documentation>

  Copyright (c) 2008, 2019 Oracle and/or its affiliates. All rights reserved.
```

```
This program and the accompanying materials are made available under the
terms of the Eclipse Public License v. 2.0 which is available at
http://www.eclipse.org/legal/epl-2.0,
or the Eclipse Distribution License v. 1.0 which is available at
http://www.eclipse.org/org/documents/edl-v10.php.

SPDX-License-Identifier: EPL-2.0 OR BSD-3-Clause

Contributors:
    Linda DeMichiel - Version 2.2 (July 7, 2017)

  </xsd:documentation>
</xsd:annotation>

 <xsd:annotation>
   <xsd:documentation><![CDATA[

   This is the XML Schema for the persistence configuration file.
   The file must be named "META-INF/persistence.xml" in the
   persistence archive.

   Persistence configuration files must indicate
   the persistence schema by using the persistence namespace:

   http://xmlns.jcp.org/xml/ns/persistence

   and indicate the version of the schema by
   using the version element as shown below:

    <persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd"
      version="2.2">
        ...
    </persistence>

  ]]></xsd:documentation>
</xsd:annotation>

<xsd:simpleType name="versionType">
  <xsd:restriction base="xsd:token">
    <xsd:pattern value="[0-9]+(\.[0-9]+)*"/>
  </xsd:restriction>
</xsd:simpleType>

<!-- **************************************************** -->
```

```
<xsd:element name="persistence">
  <xsd:complexType>
    <xsd:sequence>

      <!-- ************************************************** -->

      <xsd:element name="persistence-unit"
                   minOccurs="1" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:annotation>
            <xsd:documentation>

              Configuration of a persistence unit.

            </xsd:documentation>
          </xsd:annotation>
          <xsd:sequence>

          <!-- ************************************************** -->

            <xsd:element name="description" type="xsd:string"
                         minOccurs="0">
              <xsd:annotation>
                <xsd:documentation>

                  Description of this persistence unit.

                </xsd:documentation>
              </xsd:annotation>
            </xsd:element>

            <!-- ************************************************** -->

            <xsd:element name="provider" type="xsd:string"
                         minOccurs="0">
              <xsd:annotation>
                <xsd:documentation>

                  Provider class that supplies EntityManagers for this
                  persistence unit.

                </xsd:documentation>
              </xsd:annotation>
            </xsd:element>

            <!-- ************************************************** -->

            <xsd:element name="jta-data-source" type="xsd:string"
```

```
                minOccurs="0">
    <xsd:annotation>
      <xsd:documentation>

        The container-specific name of the JTA datasource to use.

      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>

  <!-- **************************************************** -->

  <xsd:element name="non-jta-data-source" type="xsd:string"
                minOccurs="0">
    <xsd:annotation>
      <xsd:documentation>

        The container-specific name of a non-JTA datasource to use.

      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>

  <!-- **************************************************** -->

  <xsd:element name="mapping-file" type="xsd:string"
                minOccurs="0" maxOccurs="unbounded">
    <xsd:annotation>
      <xsd:documentation>

        File containing mapping information. Loaded as a resource
        by the persistence provider.

      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>

  <!-- **************************************************** -->

  <xsd:element name="jar-file" type="xsd:string"
                minOccurs="0" maxOccurs="unbounded">
    <xsd:annotation>
      <xsd:documentation>

        Jar file that is to be scanned for managed classes.

      </xsd:documentation>
    </xsd:annotation>
```

```
    </xsd:element>

    <!-- **************************************************** -->

    <xsd:element name="class" type="xsd:string"
                 minOccurs="0" maxOccurs="unbounded">
      <xsd:annotation>
        <xsd:documentation>

          Managed class to be included in the persistence unit and
          to scan for annotations.  It should be annotated
          with either @Entity, @Embeddable or @MappedSuperclass.

        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>

    <!-- **************************************************** -->

    <xsd:element name="exclude-unlisted-classes" type="xsd:boolean"
                 default="true" minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>

          When set to true then only listed classes and jars will
          be scanned for persistent classes, otherwise the
          enclosing jar or directory will also be scanned.
          Not applicable to Java SE persistence units.

        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>

    <!-- **************************************************** -->

    <xsd:element name="shared-cache-mode"
                 type="persistence:persistence-unit-caching-type"
                 minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>

          Defines whether caching is enabled for the
          persistence unit if caching is supported by the
          persistence provider. When set to ALL, all entities
          will be cached. When set to NONE, no entities will
          be cached. When set to ENABLE_SELECTIVE, only entities
          specified as cacheable will be cached. When set to
          DISABLE_SELECTIVE, entities specified as not cacheable
```

```
      will not be cached. When not specified or when set to
      UNSPECIFIED, provider defaults may apply.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<!-- **************************************************** -->

<xsd:element name="validation-mode"
             type="persistence:persistence-unit-validation-mode-type"
             minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      The validation mode to be used for the persistence unit.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>


<!-- **************************************************** -->

<xsd:element name="properties" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      A list of standard and vendor-specific properties
      and hints.

    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="property"
                   minOccurs="0" maxOccurs="unbounded">
        <xsd:annotation>
          <xsd:documentation>
            A name-value pair.
          </xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
          <xsd:attribute name="name" type="xsd:string"
                         use="required"/>
          <xsd:attribute name="value" type="xsd:string"
                         use="required"/>
        </xsd:complexType>
```

```
                </xsd:element>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>

        </xsd:sequence>

        <!-- **************************************************** -->

        <xsd:attribute name="name" type="xsd:string" use="required">
          <xsd:annotation>
            <xsd:documentation>

              Name used in code to reference this persistence unit.

            </xsd:documentation>
          </xsd:annotation>
        </xsd:attribute>

        <!-- **************************************************** -->

        <xsd:attribute name="transaction-type"
                       type="persistence:persistence-unit-transaction-type">
          <xsd:annotation>
            <xsd:documentation>

              Type of transactions used by EntityManagers from this
              persistence unit.

            </xsd:documentation>
          </xsd:annotation>
        </xsd:attribute>

      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="version" type="persistence:versionType"
                 fixed="2.2" use="required"/>
</xsd:complexType>
</xsd:element>

<!-- **************************************************** -->

<xsd:simpleType name="persistence-unit-transaction-type">
  <xsd:annotation>
    <xsd:documentation>

      public enum PersistenceUnitTransactionType {JTA, RESOURCE_LOCAL};
```

```
      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="JTA"/>
      <xsd:enumeration value="RESOURCE_LOCAL"/>
    </xsd:restriction>
  </xsd:simpleType>

<!-- **************************************************** -->

  <xsd:simpleType name="persistence-unit-caching-type">
    <xsd:annotation>
      <xsd:documentation>

        public enum SharedCacheMode { ALL, NONE, ENABLE_SELECTIVE, DISABLE_SELECTIVE,
UNSPECIFIED};

      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="ALL"/>
      <xsd:enumeration value="NONE"/>
      <xsd:enumeration value="ENABLE_SELECTIVE"/>
      <xsd:enumeration value="DISABLE_SELECTIVE"/>
      <xsd:enumeration value="UNSPECIFIED"/>
    </xsd:restriction>
  </xsd:simpleType>

<!-- **************************************************** -->

  <xsd:simpleType name="persistence-unit-validation-mode-type">
    <xsd:annotation>
      <xsd:documentation>

        public enum ValidationMode { AUTO, CALLBACK, NONE};

      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="AUTO"/>
      <xsd:enumeration value="CALLBACK"/>
      <xsd:enumeration value="NONE"/>
    </xsd:restriction>
  </xsd:simpleType>

</xsd:schema>
```

Chapter 9. Container and Provider Contracts for Deployment and Bootstrapping

This chapter defines requirements on the Java EE container and on the persistence provider for deployment and bootstrapping.

9.1. Java EE Deployment

Each persistence unit deployed into a Java EE container consists of a single *persistence.xml* file, any number of mapping files, and any number of class files.

At deployment time the container is responsible for scanning the locations specified in Section 8.2 and discovering the *persistence.xml* files and processing them.

When the container finds a *persistence.xml* file, it must process the persistence unit definitions that it contains. The container must validate the *persistence.xml* file against the *persistence_2_2.xsd*, *persistence_2_1.xsd*, *persistence_2_0.xsd*, or *persistence_1_0.xsd* schema in accordance with the version specified by the *persistence.xml* file and report any validation errors. Provider or data source information not specified in the *persistence.xml* file must be provided at deployment time or defaulted by the container. The container may optionally add any container-specific properties to be passed to the provider when creating the entity manager factory for the persistence unit.

Once the container has read the persistence metadata, it determines the *javax.persistence.spi.PersistenceProvider* implementation class for each deployed named persistence unit. The container then creates an instance of the *PersistenceProvider* implementation class for each deployed named persistence unit and invokes the *createContainerEntityManagerFactory* method on that instance.

- The container must implement the *PersistenceUnitInfo* interface described in Section 9.6 and pass the metadata—in the form of a *PersistenceUnitInfo* instance—to the persistence provider as part of this call.
- If a Bean Validation provider exists in the container environment and the *validation-mode NONE* is not specified, a *ValidatorFactory* instance must be made available by the container. The container is responsible for passing this *ValidatorFactory* instance via the map that is passed as an argument to the *createContainerEntityManagerFactory* call. The map key used must be the standard property name *javax.persistence.validation.factory*.
- If CDI is enabled, a *BeanManager* instance must be made available by the container. The container is responsible for passing this *BeanManager* instance via the map that is passed as an argument to the *createContainerEntityManagerFactory* call. The map key used must be the standard property name *javax.persistence.bean.manager*.

The *EntityManagerFactory* instance obtained as a result will be used by the container to create container-managed entity managers. Only one EntityManagerFactory is permitted to be created for

each deployed persistence unit configuration. Any number of EntityManager instances may be created from a given factory.

In a Java EE environment, the classes of the persistence unit should not be loaded by the application class loader or any of its parent class loaders until after the entity manager factory for the persistence unit has been created.

When a persistence unit is redeployed, the container should call the *close* method on the previous *EntityManagerFactory* instance and call the *createContainerEntityManagerFactory* method again, with the required *PersistenceUnitInfo* metadata, to achieve the redeployment.

9.2. Bootstrapping in Java SE Environments

In Java SE environments, the *Persistence.createEntityManagerFactory* method is used by the application to create an entity manager factory [93: Use of these Java SE bootstrapping APIs may be supported in Java EE containers; however, support for such use is not required.].

A persistence provider implementation running in a Java SE environment should also act as a service provider by supplying a service provider configuration file as defined by the Java SE platform.

The provider configuration file serves to export the provider implementation class to the *Persistence* bootstrap class, positioning the provider as a candidate for backing named persistence units. The provider supplies the provider configuration file by creating a text file named *javax.persistence.spi.PersistenceProvider* and placing it in the *META-INF/services* directory of one of its JAR files. The contents of the file should be the name of the provider implementation class of the *javax.persistence.spi.PersistenceProvider* interface.

Example:

A persistence vendor called ACME persistence products ships a JAR called *acme.jar* that contains its persistence provider implementation. The JAR includes the provider configuration file.

```
acme.jar
    META-INF/services/javax.persistence.spi.PersistenceProvider
    com.acme.PersistenceProvider
    ...
```

The contents of the *META-INF/services/javax.persistence.spi.PersistenceProvider* file is nothing more than the name of the implementation class: *com.acme.PersistenceProvider*.

Persistence provider jars may be installed or made available in the same ways as other service providers, e.g. as extensions or added to the application classpath.

The *Persistence* bootstrap class must locate all of the persistence providers using the *PersistenceProviderResolver* mechanism described in Section 9.3 and call *createEntityManagerFactory*

on them in turn until an appropriate backing provider returns an *EntityManagerFactory* instance. A provider may deem itself as appropriate for the persistence unit if any of the following are true:

- Its implementation class has been specified in the *provider* element for that persistence unit in the *persistence.xml* file and has not been overridden by a different *javax.persistence.provider* property value included in the Map passed to the *createEntityManagerFactory* method.
- The *javax.persistence.provider* property was included in the Map passed to *createEntityManagerFactory* and the value of the property is the provider's implementation class.
- No provider was specified for the persistence unit in either the *persistence.xml* or the property map.

If a provider does not qualify as the provider for the named persistence unit, it must return *null* when *createEntityManagerFactory* is invoked on it.

9.2.1. Schema Generation

In Java SE environments, the *Persistence.generateSchema* method may be used by the application to cause schema generation to occur as a separate phase from entity manager factory creation.

In this case, the *Persistence* bootstrap class must locate all of the persistence providers using the *PersistenceProviderResolver* mechanism described in Section 9.3 and call *generateSchema* on them in turn until an appropriate backing provider returns *true*. A provider may deem itself as appropriate for the persistence unit if any of the following are true:

- Its implementation class has been specified in the *provider* element for that persistence unit in the *persistence.xml* file and has not been overridden by a different *javax.persistence.provider* property value included in the Map passed to the *generateSchema* method.
- The *javax.persistence.provider* property was included in the Map passed to *generateSchema* and the value of the property is the provider's implementation class.
- No provider was specified for the persistence unit in either the *persistence.xml* or the property map.

If a provider does not qualify as the provider for the named persistence unit, it must return *false* when *generateSchema* is invoked on it.

9.3. Determining the Available Persistence Providers

The *PersistenceProviderResolver* and *PersistenceProviderResolverHolder* mechanism supports the dynamic discovery of persistence providers. [94: In dynamic environments (e.g., OSGi-based environments, containers based on dynamic kernels, etc.), the list of persistence providers may change.]

The *PersistenceProviderResolver* instance is responsible for returning the list of providers available in the environment.

The *PersistenceProviderResolverHolder* class holds the *PersistenceProviderResolver* instance that is in use. The implementation of *PersistenceProviderResolverHolder* must be threadsafe, but no guarantee is made against multiple threads setting the resolver.

The container is allowed to implement and set a specific *PersistenceProviderResolver* provided that it respects the *PersistenceProviderResolver* contract. The *PersistenceProviderResolver* instance to be used is set by the container using the *PersistenceProviderResolverHolder.setPersistenceProviderResolver* method. [95: If a custom PersistenceProviderResolver is needed in a JavaSE environment, it must be set before Persistence.createEntityManagerFactory is called. Note, however, that the setPersistenceProviderResolver method is not intended for general use, but rather is aimed at containers maintaining a dynamic environment.]

If no *PersistenceProviderResolver* is set, the *PersistenceProviderResolverHolder* must return a *PersistenceProviderResolver* that returns the providers whose persistence provider jars have been installed or made available as service providers or extensions. This default *PersistenceProviderResolver* instance does not guarantee the order in which persistence providers are returned.

A *PersistenceProviderResolver* must be threadsafe.

The *PersistenceProviderResolver.getPersistenceProviders()* method must be used to determine the list of available persistence providers.

The results of calling the *PersistenceProviderResolverHolder.getPersistenceProviderResolver* and the *PersistenceProviderResolver.getPersistenceProviders* methods must not be cached. In particular, the following methods must use the *PersistenceProviderResolver* instance returned by the *PersistenceProviderResolverHolder.getPersistenceProviderResolver* method to determine the list of available providers:

- *Persistence.createEntityManagerFactory(String)*
- *Persistence.createEntityManagerFactory(String, Map)*
- *PersistenceUtil.isLoaded(Object)*
- *PersistenceUtil.isLoaded(Object, String)*

These methods must not cache the list of providers and must not cache the *PersistenceProviderResolver* instance.

Note that the *PersistenceProviderResolver.getPersistenceProviders()* method can potentially be called many times. It is therefore recommended that the implementation of this method make use of caching.

Note that only a single *PersistenceProviderResolver* instance can be defined in a given classloader hierarchy at a given time.

9.3.1. PersistenceProviderResolver interface

```
package jakarta.persistence.spi;

import java.util.List;

/**
 * Determine the list of persistence providers available in the
 * runtime environment.
 *
 * <p> Implementations must be thread-safe.
 *
 * <p> Note that the <code>getPersistenceProviders</code> method can potentially
 * be called many times: it is recommended that the implementation
 * of this method make use of caching.
 *
 * @see PersistenceProvider
 * @since 2.0
 */
public interface PersistenceProviderResolver {

    /**
     * Returns a list of the <code>PersistenceProvider</code> implementations
     * available in the runtime environment.
     *
     * @return list of the persistence providers available
     *         in the environment
     */
    List<PersistenceProvider> getPersistenceProviders();

    /**
     * Clear cache of providers.
     *
     */
    void clearCachedProviders();
}
```

9.3.2. PersistenceProviderResolverHolder class

```
package jakarta.persistence.spi;

import java.util.List;

/**
 * Holds the global {@link PersistenceProviderResolver}
 * instance. If no <code>PersistenceProviderResolver</code> is set by the
 * environment, the default <code>PersistenceProviderResolver</code> is used.
 * Enable "jakarta.persistence.spi" logger to show diagnostic information.
 *
 * Implementations must be thread-safe.
 *
 * @since 2.0
 */
public class PersistenceProviderResolverHolder {

    private static PersistenceProviderResolver singleton = new
DefaultPersistenceProviderResolver();

    /**
     * Returns the current persistence provider resolver.
     *
     * @return the current persistence provider resolver
     */
    public static PersistenceProviderResolver getPersistenceProviderResolver() {
        return singleton;
    }

    /**
     * Defines the persistence provider resolver used.
     *
     * @param resolver persistence provider resolver to be used.
     */
    public static void setPersistenceProviderResolver(PersistenceProviderResolver
resolver) {
        if (resolver == null) {
            singleton = new DefaultPersistenceProviderResolver();
        } else {
            singleton = resolver;
        }
    }

}
```

9.4. Schema Generation

In cases where a preconfigured database (or a “legacy” database) is not used or is not available, the Java Persistence schema generation facility may be used to generate the tables and other database artifacts required by the persistence application. Whether schema generation entails the creation of schemas proper in the database is determined by the environment and the configuration of the schema generation process, as described below.

Schema generation may happen either prior to application deployment or when the entity manager factory is created as part of the application deployment and initialization process.

- In Java EE environments, the container may call the *PersistenceProvider generateSchema* method separately from and/or prior to the creation of the entity manager factory for the persistence unit, or the container may pass additional information to the *createContainerEntityManagerFactory* call to cause schema generation to happen as part of the entity manager factory creation and application initialization process. The information passed to these methods controls whether the generation occurs directly in the target database, whether DDL scripts for schema generation are created, or both.
- In Java SE environments, the application may call the *Persistence generateSchema* method separately from and/or prior to the creation of the entity manager factory or may pass information to the *createEntityManagerFactory* method to cause schema generation to occur as part of the entity manager factory creation.

The application may provide DDL scripts to be used for schema generation as described in Section 8.2.1.9. The application developer may package these scripts as part of the persistence unit or may specify strings corresponding to file URLs for the location of such scripts. In Java EE environments, such scripts may be executed by the container, or the container may direct the persistence provider to execute the scripts. In Java SE environments, the execution of the scripts is the responsibility of the persistence provider. In the absence of the specification of scripts, schema generation, if requested, will be determined by the object/relational metadata of the persistence unit.

The following standard properties are defined for configuring the schema generation process. In Java EE environments these properties are passed by the container in the *Map* argument to either the *PersistenceProvider generateSchema* method or the *createContainerEntityManagerFactory* method. In Java SE environments, they are passed in the *Map* argument to either the *Persistence generateSchema* method or *createEntityManagerFactory* method.

In Java EE environments, any strings corresponding to file URLs for script sources or targets must specify absolute paths (not relative). In Java EE environments, all source and target file locations must be accessible to the application server deploying the persistence unit

- *javax.persistence.schema-generation.database.action*
  The *javax.persistence.schema-generation.database.action* property specifies the action to be taken by the persistence provider with regard to the database artifacts. The values for this property are *"none"*, *"create"*, *"drop-and-create"*, *"drop"*. If the *javax.persistence.schema-*

generation.database.action property is not specified, no schema generation actions must be taken on the database.

- *javax.persistence.schema-generation.scripts.action*
  The *javax.persistence.schema-generation.scripts.action* property specifies which scripts are to be generated by the persistence provider. The values for this property are *"none"*, *"create"*, *"drop-and-create"* , *"drop"*. A script will only be generated if the script target is specified. If this property is not specified, no scripts will be generated.
- *javax.persistence.schema-generation.create-source*
  The *javax.persistence.schema-generation.create-source* property specifies whether the creation of database artifacts is to occur on the basis of the object/relational mapping metadata, DDL script, or a combination of the two. The values for this property are *"metadata"*, *"script"*, *"metadata-then-script"*, *"script-then-metadata"*. If this property is not specified, and a script is specified by the *javax.persistence.schema-generation.create-script-source* property, the script (only) will be used for schema generation; otherwise if this property is not specified, schema generation will occur on the basis of the object/relational mapping metadata (only). The *"metadata-then-script"* and *"script-then-metadata"* values specify that a combination of metadata and script is to be used and the order in which this use is to occur. If either of these values is specified and the resulting database actions are not disjoint, the results are undefined and schema generation may fail.
- *javax.persistence.schema-generation.drop-source*
  The *javax.persistence.schema-generation.drop-source* property specifies whether the dropping of database artifacts is to occur on the basis of the object/relational mapping metadata, DDL script, or a combination of the two. The values for this property are *"metadata"*, *"script"*, *"metadata-then-script"*, *"script-then-metadata"*. If this property is not specified, and a script is specified by the *javax.persistence.schema-generation.drop-script-source* property, the script (only) will be used for the dropping of database artifacts; otherwise if this property is not specified, the dropping of database artifacts will occur on the basis of the object/relational mapping metadata (only). The *"metadata-then-script"* and *"script-then-metadata"* values specify that a combination of metadata and script is to be used and the order in which this use is to occur. If either of these values is specified and the resulting database actions are not disjoint, the results are undefined and the dropping of database artifacts may fail.
- *javax.persistence.schema-generation.create-database-schemas*
  In Java EE environments, it is anticipated that the Java EE platform provider may wish to control the creation of database schemas rather than delegate this task to the persistence provider. The *javax.persistence.schema-generation.create-database-schemas* property specifies whether the persistence provider is to create the database schema(s) in addition to creating database objects such as tables, sequences, constraints, etc. The value of this boolean property should be set to true if the persistence provider is to create schemas in the database or to generate DDL that contains “CREATE SCHEMA” commands. If this property is not supplied, the provider should not attempt to create database schemas. This property may also be specified in Java SE environments.
- *javax.persistence.schema-generation.scripts.create-target*, *javax.persistence.schema-generation.scripts.drop-target*
  If scripts are to be generated, the target locations for the writing of these scripts must be specified. The *javax.persistence.schema-generation.scripts.create-target* property specifies a *java.IO.Writer*

configured for use by the persistence provider for output of the DDL script or a string specifying the file URL for the DDL script. This property should only be specified if scripts are to be generated.
The *javax.persistence.schema-generation.scripts.drop-target* property specifies a *java.IO.Writer* configured for use by the persistence provider for output of the DDL script or a string specifying the file URL for the DDL script. This property should only be specified if scripts are to be generated.

- *javax.persistence.database-product-name*,
javax.persistence.database-major-version,
javax.persistence.database-minor-version
If scripts are to be generated by the persistence provider and a connection to the target database is not supplied, the *javax.persistence.database-product-name* property must be specified. The value of this property should be the value returned for the target database by the JDBC *DatabaseMetaData* method *getDatabaseProductName*. If sufficient database version information is not included in the result of this method, the *javax.persistence.database-major-version* and *javax.persistence.database-minor-version* properties should be specified as needed. These should contain the values returned by the JDBC *getDatabaseMajorVersion* and *getDatabaseMinorVersion* methods respectively.

- *javax.persistence.schema-generation.create-script-source*,
javax.persistence.schema-generation.drop-script-source
The *javax.persistence.schema-generation.create-script-source* and *javax.persistence.schema-generation.drop-script-source* properties are used for script execution. In Java EE container environments, it is generally expected that the container will be responsible for executing DDL scripts, although the container is permitted to delegate this task to the persistence provider. If DDL scripts are to be used in Java SE environments or if the Java EE container delegates the execution of scripts to the persistence provider, these properties must be specified.
The *javax.persistence.schema-generation.create-script-source* property specifies a *java.IO.Reader* configured for reading of the DDL script or a string designating a file URL for the DDL script.
The *javax.persistence.schema-generation.drop-script-source* property specifies a *java.IO.Reader* configured for reading of the DDL script or a string designating a file URL for the DDL script.

- *javax.persistence.schema-generation.connection*
The *javax.persistence.schema-generation.connection* property specifies the JDBC connection to be used for schema generation. This is intended for use in Java EE environments, where the platform provider may want to control the database privileges that are available to the persistence provider. This connection is provided by the container, and should be closed by the container when the schema generation request or entity manager factory creation completes. The connection provided must have credentials sufficient for the persistence provider to carry out the requested actions. If this property is not specified, the persistence provider should use the DataSource that has otherwise been provided.

9.4.1. Data Loading

Data loading, by means of the use of SQL scripts, may occur as part of the schema generation process after the creation of the database artifacts or independently of schema generation. The specification of the *javax.persistence.sql-load-script-source* controls whether data loading will occur.

- *javax.persistence.sql-load-script-source*

In Java EE container environments, it is generally expected that the container will be responsible for executing data load scripts, although the container is permitted to delegate this task to the persistence provider. If a load script is to be used in Java SE environments or if the Java EE container delegates the execution of the load script to the persistence provider, this property must be specified. + The *javax.persistence.sql-load-script-source* property specifies a *java.IO.Reader* configured for reading of the SQL load script for database initialization or a string designating a file URL for the script.

9.5. Responsibilities of the Persistence Provider

The persistence provider must implement the *PersistenceProvider* SPI.

In Java EE environments, the persistence provider must process the metadata that is passed to it at the time *createContainerEntityManagerFactory* method is called and create an instance of *EntityManagerFactory* using the *PersistenceUnitInfo* metadata for the factory. The factory is then returned to the container.

In Java SE environments, the persistence provider must validate the *persistence.xml* file against the *persistence* schema that corresponds to the version specified by the *persistence.xml* file and report any validation errors.

The persistence provider processes the metadata annotations on the managed classes of the persistence unit.

When the entity manager factory for a persistence unit is created, it is the responsibility of the persistence provider to initialize the state of the metamodel classes of the persistence unit.

When the persistence provider obtains an object/relational mapping file, it processes the definitions that it contains. The persistence provider must validate any object/relational mapping files against the object/relational mapping schema version specified by the object/relational mapping file and report any validation errors. The object relational mapping file must specify the object/relational mapping schema that it is written against by indicating the *version* element.

In Java SE environments, the application can pass the *ValidatorFactory* instance via the map that is passed as an argument to the *Persistence.createEntityManagerFactory* call. The map key used must be the standard property name *javax.persistence.validation.factory*. If no *ValidatorFactory* instance is provided by the application, and if a Bean Validation provider is present in the classpath, the persistence provider must instantiate the *ValidatorFactory* using the default bootstrapping approach as defined by the Bean Validation specification [5], namely *Validation.buildDefaultValidatorFactory()*.

9.5.1. javax.persistence.spi.PersistenceProvider

The interface *javax.persistence.spi.PersistenceProvider* must be implemented by the persistence provider.

It is invoked by the container in Java EE environments and by the *javax.persistence.Persistence* class in

Java SE environments. The *javax.persistence.spi.PersistenceProvider* implementation is not intended to be used by the application.

The *PersistenceProvider* implementation class must have a public no-arg constructor.

```
package jakarta.persistence.spi;

import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;
import jakarta.persistence.PersistenceException;
import java.util.Map;

/**
 * Interface implemented by the persistence provider.
 *
 * <p> It is invoked by the container in Jakarta EE environments and
 * by the {@link Persistence} class in Java SE environments to
 * create an {@link EntityManagerFactory} and/or to cause
 * schema generation to occur.
 *
 * @since 1.0
 */
public interface PersistenceProvider {

    /**
     * Called by <code>Persistence</code> class when an
     * <code>EntityManagerFactory</code> is to be created.
     *
     * @param emName  the name of the persistence unit
     * @param map  a Map of properties for use by the
     * persistence provider. These properties may be used to
     * override the values of the corresponding elements in
     * the <code>persistence.xml</code> file or specify values for
     * properties not specified in the <code>persistence.xml</code>
     * (and may be null if no properties are specified).
     * @return EntityManagerFactory for the persistence unit,
     * or null if the provider is not the right provider
     */
    public EntityManagerFactory createEntityManagerFactory(String emName, Map map);

    /**
     * Called by the container when an <code>EntityManagerFactory</code>
     * is to be created.
     *
     * @param info  metadata for use by the persistence provider
     * @param map  a Map of integration-level properties for use
     * by the persistence provider (may be null if no properties
```

```
     * are specified).  These properties may include properties to
     * control schema generation.
     * If a Bean Validation provider is present in the classpath,
     * the container must pass the <code>ValidatorFactory</code> instance in
     * the map with the key <code>"jakarta.persistence.validation.factory"</code>.
     * If the containing archive is a bean archive, the container
     * must pass the BeanManager instance in the map with the key
     * <code>"jakarta.persistence.bean.manager"</code>.
     * @return EntityManagerFactory for the persistence unit
     * specified by the metadata
     */
    public EntityManagerFactory createContainerEntityManagerFactory(PersistenceUnitInfo
info, Map map);


    /**
     * Create database schemas and/or tables and/or create DDL
     * scripts as determined by the supplied properties.
     * <p>
     * Called by the container when schema generation is to
     * occur as a separate phase from creation of the entity
     * manager factory.
     * <p>
     * @param info metadata for use by the persistence provider
     * @param map properties for schema generation;  these may
     *             also include provider-specific properties
     * @throws PersistenceException if insufficient or inconsistent
     *         configuration information is provided of if schema
     *         generation otherwise fails
     *
     * @since 2.1
     */
    public void generateSchema(PersistenceUnitInfo info, Map map);

    /**
     * Create database schemas and/or tables and/or create DDL
     * scripts as determined by the supplied properties.
     * <p>
     * Called by the Persistence class when schema generation is to
     * occur as a separate phase from creation of the entity
     * manager factory.
     * <p>
     * @param persistenceUnitName the name of the persistence unit
     * @param map properties for schema generation;  these may
     *             also contain provider-specific properties.  The
     *             value of these properties override any values that
     *             may have been configured elsewhere.
     * @return true  if schema was generated, otherwise false
```

```
     * @throws PersistenceException if insufficient or inconsistent
     *         configuration information is provided or if schema
     *         generation otherwise fails
     *
     * @since 2.1
     */
    public boolean generateSchema(String persistenceUnitName, Map map);

    /**
     * Return the utility interface implemented by the persistence
     * provider.
     * @return ProviderUtil interface
     *
     * @since 2.0
     */
    public ProviderUtil getProviderUtil();
}
```

The properties used in the *createEntityManagerFactory* method in Java SE environments are described further in Section 9.7 below.

9.5.2. javax.persistence.spi.ProviderUtil

The *ProviderUtil* interface is invoked by the *PersistenceUtil* implementation to determine the load status of an entity or entity attribute. It is not intended to be invoked by the application.

```
package jakarta.persistence.spi;

import jakarta.persistence.PersistenceUtil;

/**
 * Utility interface implemented by the persistence provider.  This
 * interface is invoked by the {@link
 * PersistenceUtil} implementation to determine
 * the load status of an entity or entity attribute.
 *
 * @since 2.0
 */
public interface ProviderUtil {

    /**
     * If the provider determines that the entity has been provided
     * by itself and that the state of the specified attribute has
     * been loaded, this method returns <code>LoadState.LOADED</code>.
     * <p> If the provider determines that the entity has been provided
     * by itself and that either entity attributes with <code>FetchType.EAGER</code>
```

```
     * have not been loaded or that the state of the specified
     * attribute has not been loaded, this methods returns
     * <code>LoadState.NOT_LOADED</code>.
     * <p> If a provider cannot determine the load state, this method
     * returns <code>LoadState.UNKNOWN</code>.
     * <p> The provider's implementation of this method must not obtain
     * a reference to an attribute value, as this could trigger the
     * loading of entity state if the entity has been provided by a
     * different provider.
     * @param entity  entity instance
     * @param attributeName  name of attribute whose load status is
     *        to be determined
     * @return load status of the attribute
     */
    public LoadState isLoadedWithoutReference(Object entity, String attributeName);

    /**
     * If the provider determines that the entity has been provided
     * by itself and that the state of the specified attribute has
     * been loaded, this method returns <code>LoadState.LOADED</code>.
     * <p> If a provider determines that the entity has been provided
     * by itself and that either the entity attributes with <code>FetchType.EAGER</code>
     * have not been loaded or that the state of the specified
     * attribute has not been loaded, this method returns
     * return <code>LoadState.NOT_LOADED</code>.
     * <p> If the provider cannot determine the load state, this method
     * returns <code>LoadState.UNKNOWN</code>.
     * <p> The provider's implementation of this method is permitted to
     * obtain a reference to the attribute value.  (This access is
     * safe because providers which might trigger the loading of the
     * attribute state will have already been determined by
     * <code>isLoadedWithoutReference</code>. )
     *
     * @param entity  entity instance
     * @param attributeName  name of attribute whose load status is
     *        to be determined
     * @return load status of the attribute
     */
    public LoadState isLoadedWithReference(Object entity, String attributeName);

    /**
     * If the provider determines that the entity has been provided
     * by itself and that the state of all attributes for which
     * <code>FetchType.EAGER</code> has been specified have been loaded, this
     * method returns <code>LoadState.LOADED</code>.
     * <p> If the provider determines that the entity has been provided
     * by itself and that not all attributes with <code>FetchType.EAGER</code>
     * have been loaded, this method returns <code>LoadState.NOT_LOADED</code>.
```

```
     * <p> If the provider cannot determine if the entity has been
     * provided by itself, this method returns <code>LoadState.UNKNOWN</code>.
     * <p> The provider's implementation of this method must not obtain
     * a reference to any attribute value, as this could trigger the
     * loading of entity state if the entity has been provided by a
     * different provider.
     * @param entity whose loaded status is to be determined
     * @return load status of the entity
     */
    public LoadState isLoaded(Object entity);
}
```

```
package jakarta.persistence.spi;

/**
 * Load states returned by the {@link ProviderUtil} SPI methods.
 * @since 2.0
 *
 */
public enum LoadState {
    /** The state of the element is known to have been loaded. */
    LOADED,
    /** The state of the element is known not to have been loaded. */
    NOT_LOADED,
    /** The load state of the element cannot be determined. */
    UNKNOWN
}
```

9.6. javax.persistence.spi.PersistenceUnitInfo Interface

```
package jakarta.persistence.spi;

import javax.sql.DataSource;
import java.util.List;
import java.util.Properties;
import java.net.URL;
import jakarta.persistence.SharedCacheMode;
import jakarta.persistence.ValidationMode;
import jakarta.persistence.EntityManagerFactory;

/**
 * Interface implemented by the container and used by the
 * persistence provider when creating an {@link EntityManagerFactory}.
 *
```

```
 * @since 1.0
 */
public interface PersistenceUnitInfo {

    /**
     * Returns the name of the persistence unit. Corresponds to the
     * <code>name</code> attribute in the <code>persistence.xml</code> file.
     * @return  the name of the persistence unit
     */
    public String getPersistenceUnitName();

    /**
     * Returns the fully qualified name of the persistence provider
     * implementation class. Corresponds to the <code>provider</code> element in
     * the <code>persistence.xml</code> file.
     * @return  the fully qualified name of the persistence provider
     * implementation class
     */
    public String getPersistenceProviderClassName();

    /**
     * Returns the transaction type of the entity managers created by
     * the <code>EntityManagerFactory</code>. The transaction type corresponds to
     * the <code>transaction-type</code> attribute in the <code>persistence.xml</code>
file.
     * @return  transaction type of the entity managers created
     * by the EntityManagerFactory
     */
    public PersistenceUnitTransactionType getTransactionType();

    /**
     * Returns the JTA-enabled data source to be used by the
     * persistence provider. The data source corresponds to the
     * <code>jta-data-source</code> element in the <code>persistence.xml</code> file or
is
     * provided at deployment or by the container.
     * @return the JTA-enabled data source to be used by the
     * persistence provider
     */
    public DataSource getJtaDataSource();

    /**
     * Returns the non-JTA-enabled data source to be used by the
     * persistence provider for accessing data outside a JTA
     * transaction. The data source corresponds to the named
     * <code>non-jta-data-source</code> element in the <code>persistence.xml</code> file
or
     * provided at deployment or by the container.
```

```
     * @return the non-JTA-enabled data source to be used by the
     * persistence provider for accessing data outside a JTA
     * transaction
     */
    public DataSource getNonJtaDataSource();

    /**
     * Returns the list of the names of the mapping files 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.  Each mapping file name
     * corresponds to a <code>mapping-file</code> element in the
     * <code>persistence.xml</code> file.
     * @return the list of mapping file names that the persistence
     * provider must load to determine the mappings for the entity
     * classes
     */
    public List<String> getMappingFileNames();

    /**
     * Returns a list of URLs for the jar files or exploded jar
     * file directories that the persistence provider must examine
     * for managed classes of the persistence unit. Each URL
     * corresponds to a <code>jar-file</code> element in the
     * <code>persistence.xml</code> file. A URL will either be a
     * file: URL referring to a jar file or referring to a directory
     * that contains an exploded jar file, or some other URL from
     * which an InputStream in jar format can be obtained.
     * @return a list of URL objects referring to jar files or
     * directories
     */
    public List<URL> getJarFileUrls();

    /**
     * Returns the URL for the jar file or directory that is the
     * root of the persistence unit. (If the persistence unit is
     * rooted in the WEB-INF/classes directory, this will be the
     * URL of that directory.)
     * The URL will either be a file: URL referring to a jar file
     * or referring to a directory that contains an exploded jar
     * file, or some other URL from which an InputStream in jar
     * format can be obtained.
     * @return a URL referring to a jar file or directory
     */
    public URL getPersistenceUnitRootUrl();

    /**
```

```
     * Returns the list of the names of the classes that the
     * persistence provider must add to its set of managed
     * classes. Each name corresponds to a named <code>class</code> element in the
     * <code>persistence.xml</code> file.
     * @return the list of the names of the classes that the
     * persistence provider must add to its set of managed
     * classes
     */
    public List<String> getManagedClassNames();

    /**
     * Returns whether classes in the root of the persistence unit
     * that have not been explicitly listed are to be included in the
     * set of managed classes. This value corresponds to the
     * <code>exclude-unlisted-classes</code> element in the <code>persistence.xml</code>
file.
     * @return whether classes in the root of the persistence
     * unit that have not been explicitly listed are to be
     * included in the set of managed classes
     */
    public boolean excludeUnlistedClasses();

    /**
     * Returns the specification of how the provider must use
     * a second-level cache for the persistence unit.
     * The result of this method corresponds to the <code>shared-cache-mode</code>
     * element in the <code>persistence.xml</code> file.
     * @return the second-level cache mode that must be used by the
     * provider for the persistence unit
     *
     * @since 2.0
     */
    public SharedCacheMode getSharedCacheMode();

    /**
     * Returns the validation mode to be used by the persistence
     * provider for the persistence unit.  The validation mode
     * corresponds to the <code>validation-mode</code> element in the
     * <code>persistence.xml</code> file.
     * @return the validation mode to be used by the
     * persistence provider for the persistence unit
     *
     * @since 2.0
     */
    public ValidationMode getValidationMode();

    /**
     * Returns a properties object. Each property corresponds to a
```

```
     * <code>property</code> element in the <code>persistence.xml</code> file
     * or to a property set by the container.
     * @return Properties object
     */
    public Properties getProperties();

    /**
     * Returns the schema version of the <code>persistence.xml</code> file.
     * @return persistence.xml schema version
     *
     * @since 2.0
     */
    public String getPersistenceXMLSchemaVersion();

    /**
     * Returns ClassLoader that the provider may use to load any
     * classes, resources, or open URLs.
     * @return ClassLoader that the provider may use to load any
     * classes, resources, or open URLs
     */
    public ClassLoader getClassLoader();

    /**
     * Add a transformer supplied by the provider that will be
     * called for every new class definition or class redefinition
     * that gets loaded by the loader returned by the
     * {@link PersistenceUnitInfo#getClassLoader} method. The transformer
     * has no effect on the result returned by the
     * {@link PersistenceUnitInfo#getNewTempClassLoader} method.
     * Classes are only transformed once within the same classloading
     * scope, regardless of how many persistence units they may be
     * a part of.
     * @param transformer   provider-supplied transformer that the
     * container invokes at class-(re)definition time
     */
    public void addTransformer(ClassTransformer transformer);

    /**
     * Return a new instance of a ClassLoader that the provider may
     * use to temporarily load any classes, resources, or open
     * URLs. The scope and classpath of this loader is exactly the
     * same as that of the loader returned by {@link
     * PersistenceUnitInfo#getClassLoader}. None of the classes loaded
     * by this class loader will be visible to application
     * components. The provider may only use this ClassLoader within
     * the scope of the {@link
     * PersistenceProvider#createContainerEntityManagerFactory} call.
     * @return temporary ClassLoader with same visibility as current
```

```
     * loader
     */
    public ClassLoader getNewTempClassLoader();
}
```

The enum *javax.persistence.spi.PersistenceUnitTransactionType* defines whether the entity managers created by the factory will be JTA or resource-local entity managers.

```
package jakarta.persistence.spi;

import jakarta.persistence.EntityManagerFactory;

/**
 * Specifies whether entity managers created by the {@link
 * EntityManagerFactory} will be JTA or
 * resource-local entity managers.
 *
 * @since 1.0
 */
public enum PersistenceUnitTransactionType {

    /** JTA entity managers will be created. */
    JTA,

    /** Resource-local entity managers will be created. */
    RESOURCE_LOCAL
}
```

The enum *javax.persistence.SharedCacheMode* defines the use of caching. The *persistence.xml shared-cache-mode* element has no default value. The *getSharedCacheMode* method must return *UNSPECIFIED* if the *shared-cache-mode* element has not been specified for the persistence unit.

```
package jakarta.persistence;

import jakarta.persistence.spi.PersistenceUnitInfo;

/**
 * Specifies how the provider must use a second-level cache for the
 * persistence unit.  Corresponds to the value of the <code>persistence.xml</code>
 * <code>shared-cache-mode</code> element, and returned as the result of
 * {@link PersistenceUnitInfo#getSharedCacheMode()}.
 *
 * @since 2.0
 */
public enum SharedCacheMode {

    /**
     * All entities and entity-related state and data are cached.
     */
    ALL,

    /**
     * Caching is disabled for the persistence unit.
     */
    NONE,

    /**
     * Caching is enabled for all entities for <code>Cacheable(true)</code>
     * is specified.  All other entities are not cached.
     */
    ENABLE_SELECTIVE,

    /**
     * Caching is enabled for all entities except those for which
     * <code>Cacheable(false)</code> is specified.  Entities for which
     * <code>Cacheable(false)</code> is specified are not cached.
     */
    DISABLE_SELECTIVE,

    /**
     *
     * Caching behavior is undefined: provider-specific defaults may apply.
     */
    UNSPECIFIED
}
```

The enum *javax.persistence.ValidationMode* defines the validation mode.

```
package jakarta.persistence;

/**
 * The validation mode to be used by the provider for the persistence
 * unit.
 *
 * @since 2.0
 */
public enum ValidationMode {

    /**
     * If a Bean Validation provider is present in the environment,
     * the persistence provider must perform the automatic validation
     * of entities.  If no Bean Validation provider is present in the
     * environment, no lifecycle event validation takes place.
     * This is the default behavior.
     */
    AUTO,

    /**
     * The persistence provider must perform the lifecycle event
     * validation.  It is an error if there is no Bean Validation
     * provider present in the environment.
     */
    CALLBACK,

    /**
     * The persistence provider must not perform lifecycle event validation.
     */
    NONE
}
```

9.6.1. javax.persistence.spi.ClassTransformer Interface

The *javax.persistence.spi.ClassTransformer* interface is implemented by a persistence provider that wants to transform entities and managed classes at class load time or at class redefinition time.

```
package jakarta.persistence.spi;

import java.security.ProtectionDomain;
import java.lang.instrument.IllegalClassFormatException;

/**
 * A persistence provider supplies an instance of this
 * interface to the {@link PersistenceUnitInfo#addTransformer
 * PersistenceUnitInfo.addTransformer}
 * method. The supplied transformer instance will get
 * called to transform entity class files when they are
 * loaded or redefined. The transformation occurs before
 * the class is defined by the JVM.
 *
 * @since 1.0
 */
public interface ClassTransformer {

    /**
     * Invoked when a class is being loaded or redefined.
     * The implementation of this method may transform the
     * supplied class file and return a new replacement class
     * file.
     *
     * @param loader  the defining loader of the class to be
     *        transformed, may be null if the bootstrap loader
     * @param className  the name of the class in the internal form
     *        of fully qualified class and interface names
     * @param classBeingRedefined  if this is a redefine, the
     *        class being redefined, otherwise null
     * @param protectionDomain  the protection domain of the
     *        class being defined or redefined
     * @param classfileBuffer  the input byte buffer in class
     *        file format - must not be modified
     * @return a well-formed class file buffer (the result of
     *         the transform), or null if no transform is performed
     * @throws IllegalClassFormatException  if the input does
     *         not represent a well-formed class file
     */
    byte[] transform(ClassLoader loader,
                     String className,
                     Class<?> classBeingRedefined,
                     ProtectionDomain protectionDomain,
                     byte[] classfileBuffer)
        throws IllegalClassFormatException;
}
```

9.7. javax.persistence.Persistence Class

The *Persistence* class is used to obtain an *EntityManagerFactory* instance in Java SE environments. It may also be used for schema generation— i.e., to create database schemas and/or tables and/or to create DDL scripts.

The *Persistence* class is available in a Java EE container environment as well; however, support for the Java SE bootstrapping APIs is not required in container environments.

The *Persistence* class is used to obtain a *PersistenceUtil* instance in both Java EE and Java SE environments.

```
package jakarta.persistence;

import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.HashSet;
import jakarta.persistence.spi.PersistenceProvider;
import jakarta.persistence.spi.PersistenceProviderResolver;
import jakarta.persistence.spi.PersistenceProviderResolverHolder;
import jakarta.persistence.spi.LoadState;

/**
 * Bootstrap class that is used to obtain an {@link EntityManagerFactory}
 * in Java SE environments.  It may also be used to cause schema
 * generation to occur.
 *
 * <p> The <code>Persistence</code> class is available in a Jakarta EE
 * container environment as well; however, support for the Java SE
 * bootstrapping APIs is not required in container environments.
 *
 * <p> The <code>Persistence</code> class is used to obtain a {@link
 * PersistenceUtil PersistenceUtil} instance in both
 * Jakarta EE and Java SE environments.
 *
 * @since 1.0
 */
public class Persistence {

    /**
     * Create and return an EntityManagerFactory for the named
     * persistence unit.
     *
     * @param persistenceUnitName
     *            the name of the persistence unit
     * @return the factory that creates EntityManagers configured according to
```

```
     *         the specified persistence unit
     */
    public static EntityManagerFactory createEntityManagerFactory(String
persistenceUnitName) {
        return createEntityManagerFactory(persistenceUnitName, null);
    }

    /**
     * Create and return an EntityManagerFactory for the named persistence unit
     * using the given properties.
     *
     * @param persistenceUnitName
     *            the name of the persistence unit
     * @param properties
     *            Additional properties to use when creating the factory.
     *            These properties may include properties to control
     *            schema generation.  The values of these properties override
     *            any values that may have been configured elsewhere.
     * @return the factory that creates EntityManagers configured according to
     *         the specified persistence unit.
     */
    public static EntityManagerFactory createEntityManagerFactory(String
persistenceUnitName, Map properties) {

        EntityManagerFactory emf = null;
        PersistenceProviderResolver resolver = PersistenceProviderResolverHolder
.getPersistenceProviderResolver();

        List<PersistenceProvider> providers = resolver.getPersistenceProviders();

        for (PersistenceProvider provider : providers) {
            emf = provider.createEntityManagerFactory(persistenceUnitName, properties);
            if (emf != null) {
                break;
            }
        }
        if (emf == null) {
            throw new PersistenceException("No Persistence provider for EntityManager
named " + persistenceUnitName);
        }
        return emf;
    }


    /**
     * Create database schemas and/or tables and/or create DDL
     * scripts as determined by the supplied properties.
     * <p>
```

```
     * Called when schema generation is to occur as a separate phase
     * from creation of the entity manager factory.
     * <p>
     * @param persistenceUnitName the name of the persistence unit
     * @param map properties for schema generation;  these may
     *             also contain provider-specific properties.  The
     *             value of these properties override any values that
     *             may have been configured elsewhere..
     * @throws PersistenceException if insufficient or inconsistent
     *         configuration information is provided or if schema
     *         generation otherwise fails.
     *
     * @since 2.1
     */
    public static void generateSchema(String persistenceUnitName, Map map) {
        PersistenceProviderResolver resolver = PersistenceProviderResolverHolder
.getPersistenceProviderResolver();
        List<PersistenceProvider> providers = resolver.getPersistenceProviders();

        for (PersistenceProvider provider : providers) {
            if (provider.generateSchema(persistenceUnitName, map)) {
                return;
            }
        }

        throw new PersistenceException("No Persistence provider to generate schema named
" + persistenceUnitName);
    }


    /**
     * Return the PersistenceUtil instance
     * @return PersistenceUtil instance
     * @since 2.0
     */
    public static PersistenceUtil getPersistenceUtil() {
       // ...
    }

    // ...
}
```

The *properties* argument passed to the *createEntityManagerFactory* method is used to specify both standard and vendor-specific properties and hints intended for use in creating the entity manager factory.

The following properties correspond to the elements and properties in the *persistence.xml* file. When

any of these properties are specified in the Map parameter passed to the *createEntityManagerFactory* method, their values override the values of the corresponding elements and properties in the *persistence.xml* file for the named persistence unit. They also override any defaults that the persistence provider might have applied.

- *javax.persistence.lock.timeout* — integer value in milliseconds for pessimistic lock timeout or string corresponding to integer value. This corresponds to the property of the same name in the *persistence.xml*, and is a hint only. See Section 3.4.4.3.
- *javax.persistence.query.timeout* — integer value in milliseconds for query timeout or string corresponding to integer value. This corresponds to the property of the same name in the *persistence.xml*, and is a hint only. See Section 3.10.10.
- *javax.persistence.provider* — string corresponding to the *provider* element in the pe *rsistence.xml*. See Section 8.2.1.4.
- *javax.persistence.transactionType* — string corresponding to the *transaction-type* attribute in the *persistence.xml*. See Section 8.2.1.2.
- *javax.persistence.jtaDataSource* — string corresponding to the *jta-data-source* element in the *persistence.xml*. See Section 8.2.1.5.
- *javax.persistence.nonJtaDataSource* — string corresponding to the *non-jta-data-source* element in the *persistence.xml*. See Section 8.2.1.5.
- *javax.persistence.sharedCache.mode* — string corresponding to the *shared-cache-mode* element in the *persistence.xml*. See Section 8.2.1.7.
- *javax.persistence.validation.mode* — string corresponding to the *validation-mode* element in the *persistence.xml*

  1. The value is " *auto* ", " *callback* ", or " *none* ". See Section 8.2.1.8 and Section 3.6.1.1.

- *javax.persistence.validation.group.pre-persist* — string corresponding to the *javax.persistence.validation.group.pre-persist* property in the *persistence.xml*. See Section 8.2.1.9 and Section 3.6.1.2.
- *javax.persistence.validation.group.pre-update* — string corresponding to the *javax.persistence.validation.group.pre-update* property in the *persistence.xml*. See Section 8.2.1.9 and Section 3.6.1.2.
- *javax.persistence.validation.group.pre-remove* — string corresponding to the *javax.persistence.validation.group.pre-remove* property in the *persistence.xml*. See Section 8.2.1.9 and Section 3.6.1.2.
- *javax.persistence.schema-generation.create-script-source* — string corresponding to the *javax.persistence.schema-generation.create-script-source* property in the *persistence.xml*. See Section 8.2.1.9.
- *javax.persistence.schema-generation.drop-script-source* — string corresponding to the *javax.persistence.schema-generation.drop-script-source* property in the *persistence.xml*. See Section 8.2.1.9.

- *javax.persistence.sql-load-script-source* — string corresponding to the *javax.persistence.sql-load-script-source* property in the *persistence.xml*. See Section 8.2.1.9.
- *javax.persistence.schema-generation.database.action* — string corresponding to the *javax.persistence.schema-generation.database.action* property in the *persistence.xml*. See Section 8.2.1.9.
- *javax.persistence.schema-generation.scripts.action* — string corresponding to the *javax.persistence.schema-generation.scripts.action* property in the *persistence.xml*. See Section 8.2.1.9.
- *javax.persistence.schema-generation.create-source* — string corresponding to the *javax.persistence.schema-generation.create-source* property in the *persistence.xml*. See Section 8.2.1.9.
- *javax.persistence.schema-generation.drop-source* — string corresponding to the *javax.persistence.schema-generation.drop-source* property in the *persistence.xml*. See Section 8.2.1.9.
- *javax.persistence.schema-generation.scripts.create-target* —string corresponding to the *javax.persistence.schema-generation.scripts.create-target* property in the *persistence.xml*. See Section 8.2.1.9.
- *javax.persistence.schema-generation.scripts.drop-target* — string corresponding to the *javax.persistence.schema-generation.scripts.drop-target* property in the *persistence.xml*. See Section 8.2.1.9.

The following additional standard properties are defined by this specification for the configuration of the entity manager factory:

- *javax.persistence.jdbc.driver* — value is the fully qualified name of the driver class.
- *javax.persistence.jdbc.url* — string corresponding to the driver-specific URL.
- *javax.persistence.jdbc.user* — value is the username used by database connection.
- *javax.persistence.jdbc.password* — value is the password for database connection validation.
- *javax.persistence.dataSource* — value is instance of *javax.sql.DataSource* to be used for the specified persistence unit.
- *javax.persistence.validation.factory* — value is instance of *javax.validation.ValidatorFactory*.

Any number of vendor-specific properties may also be included in the map. If a persistence provider does not recognize a property (other than a property defined by this specification), the provider must ignore it.

Vendors should use vendor namespaces for properties (e.g., *com.acme.persistence.logging*). Entries that make use of the namespace *javax.persistence* and its subnamespaces must not be used for vendor-specific information. The namespace *javax.persistence* is reserved for use by this specification.

9.8. PersistenceUtil Interface

This interface is used to determine load state. The semantics of the methods of this interface are defined in Section 9.8.1 below.

```
package jakarta.persistence;

/**
 * Utility interface between the application and the persistence
 * provider(s).
 *
 * <p> The <code>PersistenceUtil</code> interface instance obtained from the
 * {@link Persistence} class is used to determine the load state of an
 * entity or entity attribute regardless of which persistence
 * provider in the environment created the entity.
 *
 * @since 2.0
 */
public interface PersistenceUtil {

    /**
     * Determine the load state of a given persistent attribute.
     * @param entity  entity containing the attribute
     * @param attributeName name of attribute whose load state is
     *        to be determined
     * @return false if entity's state has not been loaded or
     *  if the attribute state has not been loaded, else true
     */
    public boolean isLoaded(Object entity, String attributeName);

    /**
     * Determine the load state of an entity.
     * This method can be used to determine the load state
     * of an entity passed as a reference.  An entity is
     * considered loaded if all attributes for which
     * <code>FetchType.EAGER</code> has been specified have been loaded.
     * <p> The <code>isLoaded(Object, String)</code> method should be used to
     * determine the load state of an attribute.
     * Not doing so might lead to unintended loading of state.
     * @param entity whose load state is to be determined
     * @return false if the entity has not been loaded, else true
     */
    public boolean isLoaded(Object entity);
}
```

9.8.1. Contracts for Determining the Load State of an Entity or Entity Attribute

The implementation of the *PersistenceUtil.isLoaded(Object)* method must determine the list of persistence providers available in the runtime environment [96: The determining of the persistence providers that are available is discussed in Section 9.3.] and call the *ProviderUtil.isLoaded(Object)* method on each of them until either:

- one provider returns *LoadState.LOADED*. In this case *PersistenceUtil.isLoaded* returns *true*.
- one provider returns *LoadState.NOT_LOADED*. In this case *PersistenceUtil.isLoaded* returns *false*.
- all providers return *LoadState.UNKNOWN*. In this case *PersistenceUtil.isLoaded* returns *true*.

If the *PersistenceUtil* implementation determines that only a single provider is available in the environment, it is permitted to use provider-specific methods to determine the result of *isLoaded(Object)* as long as the semantics defined in Section 3.2.9 are observed.

The implementation of the *PersistenceUtil.isLoaded(Object,String)* method must determine the list of persistence providers available in the environment and call the *ProviderUtil.isLoadedWithoutReference* method on each of them until either:

- one provider returns *LoadState.LOADED*. In this case *PersistenceUtil.isLoaded* returns *true*.
- one provider returns *LoadState.NOT_LOADED*. In this case *PersistenceUtil.isLoaded* returns *false*.
- all providers return *LoadState.UNKNOWN*. In this case, the *PersistenceUtil.isLoaded* method then calls *ProviderUtil.isLoadedWithReference* on each of the providers until:
  - one provider returns *LoadState.LOADED*. In this case *PersistenceUtil.isLoaded* return *true*.
  - one provider returns *LoadState.NOT_LOADED*. In this case, *PersistenceUtil.isLoaded* returns *false*.
  - all providers return *LoadState.UNKNOWN*. In this case, *PersistenceUtil.isLoaded* returns *true*.

If the *PersistenceUtil* implementation determines that only a single provider is available in the environment, it is permitted to use provider specific methods to determine the result of *isLoaded(Object, String)* as long as the semantics defined in Section 3.2.9 are observed.

The rationale for splitting the determination of load state between the methods isLoadedWithoutReference and isLoadedWithReference is the following.

- *It is assumed that the provider that loaded the entity is present in the environment.*
- *Providers that use bytecode enhancement don't need to access an attribute reference to determine its load state, and can determine if the entity has been provided by them.*
- *By first querying all providers using bytecode enhancement, it is insured that no attribute will be loaded by side effect.*
- *Proxy-based providers do need to access an attribute reference to determine load state, but will not trigger attribute loading as a side effect.*

- *If no provider recognizes an entity as provided by it, it is assumed to be an object that is not instrumented and is considered loaded.*

Chapter 10. Metadata Annotations

This chapter and chapter Chapter 11 define the metadata annotations introduced by this specification.

The XML schema defined in chapter Chapter 12 provides an alternative to the use of metadata annotations.

These annotations and types are in the package *javax.persistence*.

10.1. Entity

The *Entity* annotation specifies that the class is an entity. This annotation is applied to the entity class.

The *name* annotation element specifies the entity name. If the *name* element is not specified, the entity name defaults to the unqualified name of the entity class. This name is used to refer to the entity in queries.

```
@Documented
@Target(TYPE)
@Retention(RUNTIME)
public @interface Entity {
    String name() default "";
}
```

10.2. Callback Annotations

The *EntityListeners* annotation specifies the callback listener classes to be used for an entity or mapped superclass. The *EntityListeners* annotation may be applied to an entity class or mapped superclass.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface EntityListeners {
    Class[] value();
}
```

The ExcludeSuperclassListeners annotation specifies that the invocation of superclass listeners is to be excluded for the entity class (or mapped superclass) and its subclasses.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface ExcludeSuperclassListeners {
}
```

The ExcludeDefaultListeners annotation specifies that the invocation of default listeners is to be excluded for the entity class (or mapped superclass) and its subclasses.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface ExcludeDefaultListeners {
}
```

The following annotations are used to specify callback methods for the corresponding lifecycle events. These annotations may be applied to methods of an entity class, of a mapped superclass, or of an entity listener 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 {}
```

10.3. EntityGraph Annotations

10.3.1. NamedEntityGraph and NamedEntityGraphs Annotations

The *NamedEntityGraph* annotation is used to define a named entity graph. The entity graph may be retrieved by name using the *EntityManagerFactory* interface. The entity graph may be used to specify the path and boundaries for *find* operations or queries.

The *NamedEntityGraph* annotation must be applied to the entity class that forms the root of the corresponding graph of entities.

The *name* element is used to refer to the entity graph. It defaults to the entity name of the root entity to which the annotation is applied. Entity graph names must be unique within the persistence unit.

The *attributeNodes* element lists attributes of the annotated entity class that are to be included in the entity graph.

The *includeAllAttributes* element specifies that all attributes of the annotated entity class are to be included in the entity graph. An *attributeNode* element may still be used in conjunction with this element to specify a subgraph for the attribute.

The *subgraphs* element specifies a list of subgraphs, further specifying attributes that are managed types. These subgraphs are referenced by name from *NamedAttributeNode* definitions.

The *subclassSubgraphs* element specifies a list of subgraphs that add additional attributes for subclasses of the root entity to which the annotation is applied.

The *NamedEntityGraphs* annotation can be used to specify multiple named entity graphs for the entity to which it is applied.

```
@Target({TYPE})
@Retention(RUNTIME)
@Repeatable(NamedEntityGraphs.class)
public @interface NamedEntityGraph {
    String name() default "";
    NamedAttributeNode[] attributeNodes() default {};
    boolean includeAllAttributes() default false;
    NamedSubgraph[] subgraphs() default {};
    NamedSubgraph[] subclassSubgraphs() default {};
}

@Target({TYPE})
@Retention(RUNTIME)
public @interface NamedEntityGraphs {
    NamedEntityGraph[] value();
}
```

10.3.2. NamedAttributeNode Annotation

The *NamedAttributeNode* annotation is used to specify an attribute node of within an entity graph or subgraph.

The *value* element specifies the name of the corresponding attribute.

The *subgraph* element is used to refer to a *NamedSubgraph* specification that further characterizes an attribute node corresponding to a managed type (entity or embeddable). The value of the *subgraph* element must correspond to the *name* used for the subgraph in the *NamedSubgraph* element. If the referenced attribute is an entity which has entity subclasses, there may be more than one *NamedSubgraph* element with this name, and the *subgraph* element is considered to refer to all of these.

The *keySubgraph* element is used to refer to a *NamedSubgraph* specification that further characterizes an attribute node corresponding to the key of a Map-valued attribute. The value of the the *keySubgraph* element must correspond to the *name* used for the subgraph in the *NamedSubgraph* element. If the referenced attribute is an entity which has entity subclasses, there may be more than one *NamedSubgraph* element with this name, and the *keySubgraph* element is considered to refer to all of these.

```
@Target({})
@Retention(RUNTIME)
public @interface NamedAttributeNode {
    String value();
    String subgraph() default "";
    String keySubgraph() default "";
}
```

10.3.3. NamedSubgraph Annotation

The *NamedSubgraph* annotation is used to further define an attribute node. It is referenced by its name from the *subgraph* or *keySubgraph* element of a *NamedAttributeNode* element.

The *name* element is the name used to reference the subgraph from a *NamedAttributeNode* definition. In the case of entity inheritance, multiple subgraph elements have the same name.

The *type* element must be specified when the subgraph corresponds to a subclass of the entity type corresponding to the referencing attribute node.

The *attributeNodes* element lists attributes of the class that must be included. If the subgraph corresponds to a subclass of the class referenced by the corresponding attribute node, only subclass-specific attributes are listed.

```
@Target({})
@Retention(RUNTIME)
public @interface NamedSubgraph {
    String name();
    Class type() default void.class;
    NamedAttributeNode[] attributeNodes();
}
```

10.4. Annotations for Queries

10.4.1. NamedQuery Annotation

The *NamedQuery* annotation is used to specify a named query in the Java Persistence query language.

The *name* element is used to refer to the query when using the *EntityManager* methods that create query objects.

The *query* element must specify a query string in the Java Persistence query language.

The *lockMode* element specifies a lock mode for the results returned by the query. If a lock mode other than *NONE* is specified, the query must be executed within a transaction and the persistence context joined to the transaction.

The *hints* elements may be used to specify query properties and hints. Properties defined by this specification must be observed by the provider; hints defined by this specification should be observed by the provider when possible. Vendor-specific hints that are not recognized by a provider must be ignored.

The *NamedQuery* and *NamedQueries* annotations can be applied to an entity or mapped superclass.

```
@Target({TYPE})
@Retention(RUNTIME)
@Repeatable(NamedQueries.class)
public @interface NamedQuery {
    String name();
    String query();
    LockModeType lockMode() default NONE;
    QueryHint[] hints() default {};
}

@Target({})
@Retention(RUNTIME)
public @interface QueryHint {
    String name();
    String value();
}

@Target({TYPE})
@Retention(RUNTIME)
public @interface NamedQueries {
    NamedQuery[] value ();
}
```

10.4.2. 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 *query* element specifies the native query.

The *resultClass* element refers to the class of the result; the value of the *resultSetMapping* element is the name of a *SqlResultSetMapping* specification, as defined in metadata.

The *hints* elements may be used to specify query properties and hints. Hints defined by this specification should be observed by the provider when possible. Vendor-specific hints that are not recognized by a provider must be ignored.

The *NamedNativeQuery* and *NamedNativeQueries* annotations can be applied to an entity or mapped superclass.

```
@Target({TYPE})
@Retention(RUNTIME)
@Repeatable(NamedNativeQueries.class)
public @interface NamedNativeQuery {
    String name();
    String query();
    QueryHint[] hints() default {};
    Class resultClass() default void.class;
    String resultSetMapping() default "";
}

@Target({TYPE})
@Retention(RUNTIME)
public @interface NamedNativeQueries {
    NamedNativeQuery[] value ();
}
```

10.4.3. NamedStoredProcedureQuery Annotation

The *NamedStoredProcedureQuery* annotation is used to specify a stored procedure, its parameters, and its result type.

The *name* element is the name that is passed as an argument to the *createNamedStoredProcedureQuery* method to create an executable *StoredProcedureQuery* object.

The *procedureName* element is the name of the stored procedure in the database.

The parameters of the stored procedure are specified by the *parameters* element. All parameters must be specified in the order in which they occur in the parameter list of the stored procedure.

The *resultClasses* element refers to the class (or classes) that are used to map the results. The *resultSetMappings* element names one or more result set mappings, as defined by the *SqlResultSetMapping* annotation.

If there are multiple result sets, it is assumed that they will be mapped using the same mechanism—e.g., either all via a set of result class mappings or all via a set of result set mappings. The order of the specification of these mappings must be the same as the order in which the result sets will be returned by the stored procedure invocation. If the stored procedure returns one or more result sets and no *resultClasses* or *resultSetMappings* element is specified, any result set will be returned as a list of type *Object[]* . The combining of different strategies for the mapping of stored procedure result sets is undefined.

The *hints* element may be used to specify query properties and hints. Properties defined by this specification must be observed by the provider. Vendor-specific hints that are not recognized by a provider must be ignored.

The *NamedStoredProcedureQuery* and *NamedStoredProcedureQueries* annotations can be applied to an entity or mapped superclass.

```
@Target(TYPE)
@Retention(RUNTIME)
@Repeatable(NamedStoredProcedureQueries.class)
public @interface NamedStoredProcedureQuery {
    String name();
    String procedureName();
    StoredProcedureParameter[] parameters() default {};
    Class[] resultClasses() default {};
    String[] resultSetMappings() default {};
    QueryHint[] hints() default {};
}

@Target(TYPE)
@Retention(RUNTIME)
public @interface NamedStoredProcedureQueries {
    NamedStoredProcedureQuery [] value;
}
```

All parameters of a named stored procedure query must be specified using the *StoredProcedureParameter* annotation. The *name* element refers to the name of the parameter as defined by the stored procedure in the database. If a parameter name is not specified, it is assumed that the stored procedure uses positional parameters. The *mode* element specifies whether the parameter is an IN, INOUT, OUT, or REF_CURSOR parameter. REF_CURSOR parameters are used by some databases to return result sets from stored procedures. The *type* element refers to the JDBC type for the parameter.

```
@Target({})
@Retention(RUNTIME)
public @interface StoredProcedureParameter {
    String name() default "";
    ParameterMode mode() default ParameterMode.IN;
    Class type();
}

public enum ParameterMode {
    IN,
    INOUT,
    OUT,
    REF_CURSOR
}
```

10.4.4. Annotations for SQL Result Set Mappings

The *SqlResultSetMapping* annotation is used to specify the mapping of the result of a native SQL query or stored procedure.

```
@Target({TYPE})
@Retention(RUNTIME)
@Repeatable(SqlResultSetMappings.class)
public @interface SqlResultSetMapping {
    String name();
    EntityResult[] entities() default {};
    ConstructorResult[] classes() default {};
    ColumnResult[] columns() default {};
}

@Target({TYPE})
@Retention(RUNTIME)
public @interface SqlResultSetMappings {
    SqlResultSetMapping[] value();
}
```

The *name* element is the name given to the result set mapping, and is used to refer to it in the methods of the *Query* and *StoredProcedureQuery* APIs. The *entities*, *classes*, and *columns* elements are used to specify the mapping to entities, constructors, 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 *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.

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.

```
@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 specifies the name of the corresponding column in the SELECT list—i.e., column alias, if applicable.

```
@Target(value={})
@Retention(RUNTIME)
public @interface ConstructorResult {
    Class targetClass();
    ColumnResult[] columns();
}
```

The *targetClass* element specifies the class whose constructor is to be invoked.

The *columns* element specifies the mapping of columns in the SELECT list to the arguments of the intended constructor.

```
@Target({})
@Retention(RUNTIME)
public @interface ColumnResult {
    String name();
    Class type() default void.class;
}
```

The *name* element specifies the name of the column in the SELECT list.

The *type* element specifies the Java type to which the column type is to be mapped. If the *type* element is not specified, the default JDBC type mapping for the column will be used.

10.5. References to EntityManager and EntityManagerFactory

These annotations are used to express dependencies on entity managers and entity manager factories.

10.5.1. PersistenceContext Annotation

The *PersistenceContext* annotation is used to express a dependency on a container-managed entity manager and its associated persistence context.

The *name* element refers to the name by which the entity manager is to be accessed in the environment referencing context, and is not needed when dependency injection is used.

The optional *unitName* element refers to the name of the persistence unit. If the *unitName* element is specified, the persistence unit for the entity manager that is accessible in JNDI must have the same name.

The *type* element specifies whether a transaction-scoped or extended persistence context is to be used. If the *type* element is not specified, a transaction-scoped persistence context is used.

The *synchronizationType* element specifies whether the persistence context is always automatically synchronized with the current transaction or whether the persistence context must be explicitly joined to the current transaction by means of the EntityManager *joinTransaction* method.

The optional *properties* element may be used to specify properties for the container or persistence provider. Properties defined by this specification must be observed by the provider. Vendor specific properties may be included in the set of properties, and are passed to the persistence provider by the container when the entity manager is created. Properties that are not recognized by a vendor must be ignored.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(PersistenceContexts.class)
public @interface PersistenceContext {
    String name() default "";
    String unitName() default "";
    PersistenceContextType type() default TRANSACTION;
    SynchronizationType synchronization() default SYNCHRONIZED;
    PersistenceProperty[] properties() default {};
}

public enum PersistenceContextType {
    TRANSACTION,
    EXTENDED
}

public enum SynchronizationType {
    SYNCHRONIZED,
    UNSYNCHRONIZED
}

@Target({})
@Retention(RUNTIME)
public @interface PersistenceProperty {
    String name();
    String value();
}
```

The *PersistenceContexts* annotation declares one or more *PersistenceContext* annotations. It is used to express a dependency on multiple persistence contexts [97: A dependency on ultiple persistence contexts may be needed, for example, when multiple persistence units are used.].

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface PersistenceContexts {
    PersistenceContext[] value();
}
```

10.5.2. PersistenceUnit Annotation

The *PersistenceUnit* annotation is used to express a dependency on an entity manager factory and its associated persistence unit.

The *name* element refers to the name by which the entity manager factory is to be accessed in the environment referencing context, and is not needed when dependency injection is used.

The optional *unitName* element refers to the name of the persistence unit as defined in the *persistence.xml* file. If the *unitName* element is specified, the persistence unit for the entity manager factory that is accessible in JNDI must have the same name.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(PersistenceUnits.class)
public @interface PersistenceUnit {
    String name() default "";
    String unitName() default "";
}
```

The *PersistenceUnits* annotation declares one or more *PersistenceUnit* annotations. It is used to express a dependency on multiple persistence units [98: Multiple persistence units may be needed, for example, when mapping to multiple databases.].

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface PersistenceUnits {
    PersistenceUnit[] value();
}
```

10.6. Annotations for Type Converter Classes

The *Converter* annotation specifies that the annotated class is a converter and defines its scope. A converter class must be annotated with the *Converter* annotation or defined in the XML descriptor as a converter.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface Converter {
    boolean autoApply() default false;
}
```

If the *autoApply* element is specified as *true*, the persistence provider must automatically apply the converter to all mapped attributes of the specified target type for all entities in the persistence unit except for attributes for which conversion is overridden by means of the *Convert* annotation (or XML equivalent). The *Convert* annotation is described in Section 11.1.10.

In determining whether a converter is applicable to an attribute, the provider must treat primitive types and wrapper types as equivalent.

Note that Id attributes, version attributes, relationship attributes, and attributes explicitly annotated as

Enumerated or *Temporal* (or designated as such via XML) will not be converted.

If *autoApply* is *false*, only those attributes of the target type for which the *Convert* annotation (or corresponding XML element) has been specified will be converted.

Note that if *autoApply* is *true*, the *Convert* annotation may be used to override or disable auto-apply conversion on a per-attribute basis.

If there is more than one converter defined for the same target type, the *Convert* annotation should be used to explicitly specify which converter to use.

Chapter 11. Metadata for Object/Relational Mapping

The object/relational mapping metadata is part of the application domain model contract. It 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 this application dependency upon the object/relational mapping metadata and insure that the semantics and requirements expressed by that mapping are observed.

The use of object/relational mapping metadata to control schema generation is specified in Section 11.2.

11.1. Annotations for Object/Relational Mapping

These annotations and types are in the package *javax.persistence*.

XML metadata may be used as an alternative to these annotations, or to override or augment annotations, as described in Chapter 12.

11.1.1. Access Annotation

The *Access* annotation is used to specify an access type to be applied to an entity class, mapped superclass, or embeddable class, or to a specific attribute of such a class.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface Access {
    AccessType value();
}

public enum AccessType {
    FIELD,
    PROPERTY
}
```

Table 4 lists the annotation elements that may be specified for the *Access* annotation.

Table 4. Access Annotation Elements

Type	Name	Description	Default
AccessType	value	(Required) The access type to be applied to the class or attribute.	

11.1.2. AssociationOverride Annotation

The *AssociationOverride* annotation is used to override a mapping for an entity relationship.

The *AssociationOverride* annotation may be applied to an entity that extends a mapped superclass to override a relationship mapping defined by the mapped superclass. If the *AssociationOverride* annotation is not specified, the association is mapped the same as in the original mapping. When used to override a mapping defined by a mapped superclass, the *AssociationOverride* annotation is applied to the entity class.

The *AssociationOverride* annotation may be used to override a relationship mapping from an embeddable within an entity to another entity when the embeddable is on the owning side of the relationship. When used to override a relationship mapping defined by an embeddable class (including an embeddable class embedded within another embeddable class), the *AssociationOverride* annotation is applied to the field or property containing the embeddable.

When the *AssociationOverride* annotation is used to override a relationship mapping from an embeddable class, the *name* element specifies the referencing relationship field or property within the embeddable class. To override mappings at multiple levels of embedding, a dot (".") notation syntax must be used in the *name* element to indicate an attribute within an embedded attribute. The value of each identifier used with the dot notation is the name of the respective embedded field or property. When the *AssociationOverride* annotation is applied to override the mappings of an embeddable class used as a map value, " *value.* " must be used to prefix the name of the attribute within the embeddable class that is being overridden in order to specify it as part of the map value. [99: The use of map keys that contain embeddables that reference entities is not permitted.]

If the relationship mapping is a foreign key mapping, the *joinColumns* element of the *AssociationOverride* annotation is used. If the relationship mapping uses a join table, the *joinTable* element of the *AssociationOverride* element must be specified to override the mapping of the join table and/or its join columns. [100: Note that *either* the *joinColumns* element *or* the *joinTable* element of the *AssociationOverride* annotation is specified for overriding a given relationship (but never both).]

The *joinColumns* element refers to the table for the class that contains the annotation.

The *foreignKey* element is used to specify or control the generation of a foreign key constraint for the columns corresponding to the *joinColumns* element when table generation is in effect. If both this element and the *foreignKey* element of any of the *joinColumns* elements are specified, the behavior is undefined.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(AssociationOverrides.class)
public @interface AssociationOverride {
    String name();

    JoinColumn[] joinColumns() default {};

    ForeignKey foreignKey() default

    @ForeignKey(PROVIDER_DEFAULT);
    JoinTable joinTable() default @JoinTable;
}
```

Table 5 lists the annotation elements that may be specified for the *AssociationOverride* annotation.

Table 5. AssociationOverride Annotation Elements

Type	Name	Description	Default
String	name	(Required) The name of the relationship property whose mapping is being overridden if property-based access is being used, or the name of the relationship field if field-based access is used.	
JoinColumn[]	joinColumns	The join column(s) being mapped to the persistent attribute(s). The joinColumns element must be specified if a foreign key mapping is used in the overriding of the mapping of the relationship. The joinColumns element must not be specified if a join table is used in the overriding of the mapping of the relationship	

Type	Name	Description	Default
ForeignKey	foreignKey	(Optional) The foreign key constraint specification for the join columns. This is used only if table generation is in effect.	Provider's default
JoinTable	joinTable	The join table that maps the relationship. The joinTable element must be specified if a join table is used in the overriding of the mapping of the relationship. The joinTable element must not be specified if a foreign key mapping is used in the overriding of the mapping of the relationship.	.

Example 1:

```
@MappedSuperclass
public class Employee {
    @Id
    protected Integer id;

    @Version
    protected Integer version;

    @ManyToOne
    protected Address address;

    public Integer getId() { ... }

    public void setId(Integer id) { ... }

    public Address getAddress() { ... }

    public void setAddress(Address address) { ... }
}

@Entity
@AssociationOverride(name="address", joinColumns=@JoinColumn(name="ADDR_ID"))
public class PartTimeEmployee extends Employee {
    // address field mapping overridden to ADDR_ID foreign key
    @Column(name="WAGE")
    protected Float hourlyWage;

    public Float getHourlyWage() { ... }

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

Example 2: Overriding of the mapping for the *phoneNumbers* relationship defined in the *ContactInfo* embeddable class.

```
@Entity
public class Employee {
    @Id
    int id;

    @AssociationOverride(
        name="phoneNumbers",
        joinTable=@JoinTable(
            name="EMPPHONES",
            joinColumns=@JoinColumn(name="EMP"),
            inverseJoinColumns=@JoinColumn(name="PHONE")
        )
    )
    @Embedded
    ContactInfo contactInfo;

    // ...
}

@Embeddable
public class ContactInfo {
    @ManyToOne Address address; // Unidirectional
    @ManyToMany(targetEntity=PhoneNumber.class)
    List phoneNumbers;
}

@Entity
public class PhoneNumber {
    @Id
    int number;

    @ManyToMany(mappedBy="contactInfo.phoneNumbers")
    Collection<Employee> employees;
}
```

11.1.3. AssociationOverrides Annotation

The mappings of multiple relationship properties or fields may be overridden. The *AssociationOverrides* annotation can be used for this purpose.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface AssociationOverrides {
    AssociationOverride[] value();
}
```

Table 6 lists the annotation elements that may be specified for the *AssociationOverrides* annotation.

Table 6. AssociationOverrides Annotation Elements

Type	Name	Description	Default
AssociationOverride[]	value	(Required) The association override mappings that are to be applied to the relationship field or property.	

Example:

```
@MappedSuperclass
public class Employee {
    @Id
    protected Integer id;

    @Version
    protected Integer version;

    @ManyToOne
    protected Address address;

    @OneToOne
    protected Locker locker;

    public Integer getId() { ... }

    public void setId(Integer id) { ... }

    public Address getAddress() { ... }

    public void setAddress(Address address) { ... }

    public Locker getLocker() { ... }

    public void setLocker(Locker locker) { ... }
}

@Entity
@AssociationOverrides({
    @AssociationOverride(name="address", joinColumns=@JoinColumn("ADDR_ID")),
    @AssociationOverride(name="locker", joinColumns=@JoinColumn("LCKR_ID"))})
public PartTimeEmployee { ... }
```

Alternatively:

```
@Entity
@AssociationOverride(name="address", joinColumns=@JoinColumn("ADDR_ID"))
@AssociationOverride(name="locker", joinColumns=@JoinColumn("LCKR_ID"))
public PartTimeEmployee { ... }
```

11.1.4. AttributeOverride Annotation

The *AttributeOverride* annotation is used to override the mapping of a *Basic* (whether explicit or default) property or field or *Id* property or field.

The *AttributeOverride* annotation may be applied to an entity that extends a mapped superclass or to an embedded field or property to override a *Basic* mapping or *Id* mapping defined by the mapped superclass or embeddable class (or embeddable class of one of its attributes).

The *AttributeOverride* annotation may be applied to an element collection containing instances of an embeddable class or to a map collection whose key and/or value is an embeddable class. When the *AttributeOverride* annotation is applied to a map, " *key.* " or " *value.* " must be used to prefix the name of the attribute that is being overridden in order to specify it as part of the map key or map value.

To override mappings at multiple levels of embedding, a dot (".") notation form must be used in the *name* element to indicate an attribute within an embedded attribute. The value of each identifier used with the dot notation is the name of the respective embedded field or property.

If the *AttributeOverride* annotation is not specified, the column is mapped the same as in the original mapping.

Table 7 lists the annotation elements that may be specified for the *AttributeOverride* annotation.

The *column* element refers to the table for the class that contains the annotation.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(AttributeOverrides.class)
public @interface AttributeOverride {
    String name();
    Column column();
}
```

Table 7. AttributeOverride Annotation Elements

Type	Name	Description	Default
String	name	(Required) The name of the property whose mapping is being overridden if property-based access is being used, or the name of the field if field-based access is used.	
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 or mapped superclass.	

Example 1:

```
@MappedSuperclass
public class Employee {
    @Id
    protected Integer id;

    @Version
    protected Integer version;

    protected String address;

    public Integer getId() { ... }

    public void setId(Integer id) { ... }

    public String getAddress() { ... }

    public void setAddress(String address) { ... }
}

@Entity
@AttributeOverride(name="address", column=@Column(name="ADDR"))
public class PartTimeEmployee extends Employee {
    // address field mapping overridden to ADDR
    protected Float wage();

    public Float getHourlyWage() { ... }

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

Example 2:

```
@Embeddable public class Address {
    protected String street;

    protected String city;

    protected String state;

    @Embedded
    protected Zipcode zipcode;
}

@Embeddable
public class Zipcode {
    protected String zip;
    protected String plusFour;
}

@Entity
public class Customer {
    @Id
    protected Integer id;

    protected String name;

    @AttributeOverride(name="state", column=@Column(name="ADDR_STATE"))
    @AttributeOverride(name="zipcode.zip", column= @Column(name="ADDR_ZIP"))
    @Embedded
    protected Address address;

    // ...
}
```

Example 3:

```
@Entity
public class PropertyRecord {
    @EmbeddedId
    PropertyOwner owner;

    @AttributeOverrides(name="key.street", column=@Column(name="STREET_NAME"))
    @AttributeOverride(name="value.size", column=@Column(name="SQUARE_FEET"))
    @AttributeOverride(name="value.tax", column=@Column(name="ASSESSMENT"))
    @ElementCollection
    Map<Address, PropertyInfo> parcels;
}

@Embeddable
public class PropertyInfo {
    Integer parcelNumber;
    Integer size;
    BigDecimal tax;
}
```

11.1.5. AttributeOverrides Annotation

The mappings of multiple properties or fields may be overridden. The *AttributeOverrides* annotation can be used for this purpose.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface AttributeOverrides {
    AttributeOverride[] value();
}
```

Table 8 lists the annotation elements that may be specified for the *AttributeOverrides* annotation.

Table 8. AttributeOverrides Annotation Elements

Type	Name	Description	Default
AttributeOverride[]	value	(Required) The AttributeOverride mappings that are to be applied to the field or property.	

Example:

```
@Embedded
@AttributeOverrides({
    @AttributeOverride(name="startDate", column=@Column(name="EMP_START")),
    @AttributeOverride(name="endDate", column=@Column(name="EMP_END"))
})
public EmploymentPeriod getEmploymentPeriod() { ... }
```

11.1.6. Basic Annotation

The *Basic* annotation is the simplest type of mapping to a database column. The *Basic* annotation can be applied to a persistent property or instance variable of any of the following types: 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*, *java.time.LocalDate*, *java.time.LocalTime*, *java.time.LocalDateTime*, *java.time.OffsetTime*, *java.time.OffsetDateTime*, *byte[]*, *Byte[]*, *char[]*, *Character[]*, enums, and any other type that implements *Serializable*. [101: Mapping of java.time.LocalDate, java.time.LocalTime, java.time.LocalDateTime, java.time.OffsetTime, and java.time.OffsetDateTime types to columns other than those supported by the mappings defined by Appendix B of the JDBC 4.2 specification is not required to be supported by the persistence provider beyond the support required for other serializable types. See [3].] As described in Section 2.8, the use of the *Basic* annotation is optional for persistent fields and properties of these types. If the *Basic* annotation is not specified for such a field or property, the default values of the *Basic* annotation will apply.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Basic {
    FetchType fetch() default EAGER;
    boolean optional() default true;
}
```

Table 9 lists the annotation elements that may be specified for the *Basic* annotation and their default values.

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 must 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. In particular, lazy fetching might only be available for *Basic* mappings for which property-based access is used.

The *optional* element is a hint as to whether the value of the field or property may be null. It is disregarded for primitive types.

Table 9. Basic Annotation Elements

Type	Name	Description	Default
FetchType	fetch	(Optional) Whether the value of the field or property should be lazily loaded or must be eagerly fetched. The EAGER strategy is a requirement on the persistence provider runtime that the value must be eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime.	EAGER
boolean	optional	(Optional) Whether the value of the field or property may be null. This is a hint and is disregarded for primitive types; it may be used in schema generation.	true

Example 1:

```
@Basic
protected String name;
```

Example 2:

```
@Basic(fetch=LAZY)
protected String getName() { return name; }
```

11.1.7. Cacheable Annotation

The *Cacheable* annotation specifies whether an entity should be cached if caching is enabled when the value of the *persistence.xml shared-cache-mode* element is *ENABLE_SELECTIVE* or *DISABLE_SELECTIVE*. The value of the *Cacheable* annotation is inherited by subclasses; it can be overridden by specifying *Cacheable* on a subclass.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface Cacheable {
    boolean value() default true;
}
```

Cacheable(false) means that the entity and its state must not be cached by the provider.

If the *shared-cache-mode* element is not specified in the *persistence.xml* file and the *javax.persistence.sharedCache.mode* property is not specified when the entity manager factory for the persistence unit is created, the semantics of the *Cacheable* annotation are undefined.

Table 10. Cacheable Annotation Elements

Type	Name	Description	Default
boolean	value	(Optional) Whether or not the entity should be cached.	true

11.1.8. CollectionTable Annotation

The *CollectionTable* annotation is used in the mapping of collections of basic or embeddable types. The *CollectionTable* annotation specifies the table that is used for the mapping of the collection and is specified on the collection-valued field or property.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface CollectionTable {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    JoinColumn[] joinColumns() default {};
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
    UniqueConstraint[] uniqueConstraints() default {};
    Index[] indexes() default {};
}
```

By default, the columns of the collection table that correspond to the embeddable class or basic type are derived from the attributes of the embeddable class or from the basic type according to the default values of the *Column* annotation, as described in Section 11.1.9. In the case of a basic type, the column name is derived from the name of the collection-valued field or property. In the case of an embeddable class, the column names are derived from the field or property names of the embeddable class.

To override the default properties of the column used for a basic type, the *Column* annotation is used on the collection-valued attribute in addition to the *ElementCollection* annotation. The value of the

table element of the *Column* annotation defaults to the name of the collection table.

To override these defaults for an embeddable class, the *AttributeOverride* and/or *AttributeOverrides* annotations must be used in addition to the *ElementCollection* annotation. The value of the *table* element of the *Column* annotation used in the *AttributeOverride* annotation defaults to the name of the collection table. If the embeddable class contains references to other entities, the default values for the columns corresponding to those references may be overridden by means of the *AssociationOverride* and/or *AssociationOverrides* annotations.

The *foreignKey* element is used to specify or control the generation of a foreign key constraint for the columns corresponding to the *joinColumns* element when table generation is in effect. If both this element and the *foreignKey* element of any of the *joinColumns* elements are specified, the behavior is undefined. If no *foreignKey* annotation element is specified in either location, the persistence provider's default foreign key strategy will apply.

If the *CollectionTable* annotation is missing, the default values of the *CollectionTable* annotation elements apply.

Table 11 lists the annotation elements that may be specified for the *CollectionTable* annotation and their default values.

Table 11. CollectionTable Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the collection table.	The concatenation of the name of the containing entity and the name of the collection attribute, separated by an underscore.
String	catalog	(Optional) The catalog of the table.	Default catalog.
String	schema	(Optional) The schema of the table.	Default schema for user.

Type	Name	Description	Default
JoinColumn[]	joinColumns	(Optional) The foreign key columns of the collection table which reference the primary table of the entity.	(Default only applies if a single join column is used.) The same defaults as for JoinColumn (i.e., the concatenation of the following: the name of the entity; "_"; the name of the referenced primary key column.) However, if there is more than one join column, a JoinColumn annotation must be specified for each join column using the JoinColumns annotation. Both the name and the referencedColumnName elements must be specified in each such JoinColumn annotation.
ForeignKey	foreignKey	(Optional) The foreign key constraint specification for the join columns. This is used only if table generation is in effect.	Provider's default
UniqueConstraint[]	uniqueConstraints	(Optional) Unique constraints that are to be placed on the table. These are only used if table generation is in effect.	No additional constraints
Index[]	indexes	(Optional) Indexes for the table. These are only used if table generation is in effect.	No additional indexes

Example:

```
@Embeddable
public class Address {
    protected String street;
    protected String city;
    protected String state;

    // ...
}

@Entity public class Person {
    @Id
    protected String ssn;

    protected String name;

    protected Address home;

    // ...

    @ElementCollection // use default table (PERSON_NICKNAMES)
    @Column(name="name", length=50)
    protected Set<String> nickNames = new HashSet();

    // ...
}

@Entity
public class WealthyPerson extends Person {
    @ElementCollection
    @CollectionTable(name="HOMES") // use default join column name
    @AttributeOverrides({
        @AttributeOverride(name="street", column=@Column(name="HOME_STREET")),
        @AttributeOverride(name="city", column=@Column(name="HOME_CITY")),
        @AttributeOverride(name="state", column=@Column(name="HOME_STATE"))
    })
    protected Set<Address> vacationHomes = new HashSet();

    // ...
}
```

11.1.9. Column Annotation

The *Column* annotation is used to specify a mapped column for a persistent property or field.

Table 12 lists the annotation elements that may be specified for the *Column* annotation and their default values.

If no *Column* annotation is specified, the default values in Table 12 apply.

```
@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 table() default "";
    int length() default 255;
    int precision() default 0; // decimal precision
    int scale() default 0; // decimal scale
}
```

Table 12. Column Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the column.	The property or field name.
boolean	unique	(Optional) Whether the column 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 corresponds to only a single column. This constraint applies in addition to any constraint entailed by primary key mapping and to constraints specified at the table level.	false
boolean	nullable	(Optional) Whether the database column is nullable.	true
boolean	insertable	(Optional) Whether the column is included in SQL INSERT statements generated by the persistence provider.	true

Type	Name	Description	Default
boolean	updatable	(Optional) Whether the column is 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	table	(Optional) The name of the table that contains the column. If absent the column is assumed to be in the primary table for the mapped object.	Column is in primary table.
int	length	(Optional) The column length. (Applies only if a string-valued column is used.)	255
int	precision	(Optional) The precision for a decimal (exact numeric) column. (Applies only if a decimal column is used.)	0 (Value must be set by developer.)
int	scale	(Optional) The scale for a decimal (exact numeric) column. (Applies only if a decimal column is used.)	0

Example 1:

```
@Column(name="DESC", nullable=false, length=512)
public String getDescription() {
    return description;
}
```

Example 2:

```
@Column(name="DESC", columnDefinition="CLOB NOT NULL", table="EMP_DETAIL")
@Lob
public String getDescription() {
    return description;
}
```

Example 3:

```
@Column(name="ORDER_COST", updatable=false, precision=12, scale=2)
public BigDecimal getCost() {
    return cost;
}
```

11.1.10. Convert Annotation

The *Convert* annotation is applied directly to an attribute of an entity, mapped superclass, or embeddable class to specify conversion of a Basic attribute or to override the use of a converter that has been specified as *autoApply=true*. When persistent properties are used, the *Convert* annotation is applied to the getter method. It is not necessary to use the *Basic* annotation or corresponding XML element to specify the basic type

The *Convert* annotation may be applied to an entity that extends a mapped superclass to specify or override the conversion mapping for an inherited basic attribute.

```
@Target({METHOD, FIELD, TYPE})
@Retention(RUNTIME)
@Repeatable(Converts.class)
public @interface Convert {
    Class converter() default void.class;
    String attributeName() default "";
    boolean disableConversion() default false;
}
```

Table 13 lists the annotation elements that may be specified for the *Convert* annotation.

Table 13. Convert Annotation Elements

Type	Name	Description	Default
Class	converter	(Optional) The converter to be applied.	No converter

Type	Name	Description	Default
String	attributeName	(Optional) The name of the attribute to convert. Must be specified unless the Convert annotation is applied to an attribute of basic type or to an element collection of basic type. Must not be specified otherwise.	The basic attribute or basic element collection attribute to which the annotation is applied
boolean	disableConversion	(Optional) Whether conversion of the attribute is to be disabled.	false

The *converter* element is used to specify the converter that is to be applied. If an autoApply converter is applicable to the given field or property, the converter specified by the *converter* element will be applied instead.

The *disableConversion* element specifies that any applicable autoApply converter must not be applied.

The behavior is undefined if neither the *converter* element nor the *disableConversion* element has been specified.

The *Convert* annotation should not be used to specify conversion of the following: Id attributes (including the attributes of embedded ids and derived identities), version attributes, relationship attributes, and attributes explicitly annotated (or designated via XML) as *Enumerated* or *Temporal*. Applications that specify such conversions will not be portable.

The *Convert* annotation may be applied to a basic attribute or to an element collection of basic type (in which case the converter is applied to the elements of the collection). In these cases, the *attributeName* element must not be specified.

The *Convert* annotation may be applied to an embedded attribute or to a map collection attribute whose key or value is of embeddable type (in which case the converter is applied to the specified attribute of the embeddable instances contained in the collection). In these cases, the *attributeName* element must be specified.

To override conversion mappings at multiple levels of embedding, a dot (*"."*) notation form must be used in the *attributeName* element to indicate an attribute within an embedded attribute. The value of each identifier used with the dot notation is the name of the respective embedded field or property.

When the *Convert* annotation is applied to a map containing instances of embeddable classes, the *attributeName* element must be specified, and *"key."* or *"value."* must be used to prefix the name of the attribute that is to be converted in order to specify it as part of the map key or map value.

When the *Convert* annotation is applied to a map to specify conversion of a map key of basic type, *"key"*

must be used as the value of the *attributeName* element to specify that it is the map key that is to be converted.

The *Convert* annotation may be applied to an entity class that extends a mapped superclass to specify or override a conversion mapping for an inherited basic or embedded attribute.

Example 1: Convert a basic attribute

```
@Converter
public class BooleanToIntegerConverter implements AttributeConverter<Boolean, Integer> {
... }

@Entity
public class Employee {
    @Id
    long id;

    @Convert(converter=BooleanToIntegerConverter.class)
    boolean fullTime;

    // ...
}
```

Example 2: Auto-apply conversion of a basic attribute

```
@Converter(autoApply=true)
public class EmployeeDateConverter implements
    AttributeConverter<com.acme.EmployeeDate, java.sql.Date> { ... }

@Entity
public class Employee {
    @Id
    long id;

    // ...

    // EmployeeDateConverter is applied automatically
    EmployeeDate startDate;
}
```

Example 3: Disable conversion in the presence of an auto-apply converter

```
@Convert(disableConversion=true)
EmployeeDate lastReview;
```

Example 4: Apply a converter to an element collection of basic type

```
@ElementCollection
// applies to each element in the collection
@Convert(converter=NameConverter.class)
List<String> names;
```

Example 5: Apply a converter to an element collection that is a map of basic values. The converter is applied to the map *value*.

```
@ElementCollection
@Convert(converter=EmployeeNameConverter.class)
Map<String, String> responsibilities;
```

Example 6: Apply a converter to a map key of basic type

```
@OneToMany
@Convert(converter=ResponsibilityCodeConverter.class, attributeName="key")
Map<String, Employee> responsibilities;
```

Example 7: Apply a converter to an embeddable attribute

```
@Embedded
@Convert(converter=CountryConverter.class, attributeName="country")
Address address;
```

Example 8: Apply a converter to a nested embeddable attribute

```
@Embedded
@Convert(converter=CityConverter.class, attributeName="region.city")
Address address;
```

Example 9: Apply a converter to a nested attribute of an embeddable that is a map key of an element collection

```
@Entity
public class PropertyRecord {
    // ...

    @Convert(converter=CityConverter.class, attributeName="key.region.city")
    @ElementCollection
    Map<Address, PropertyInfo> parcels;
}
```

Example 10: Apply a converter to an embeddable that is a map key for a relationship

```
@OneToMany
@Convert(converter=ResponsibilityCodeConverter.class, attributeName="key.jobType")
Map<Responsibility, Employee> responsibilities;
```

Example 11: Override conversion mappings for attributes inherited from a mapped superclass

```
@Entity
@Convert(converter=DateConverter.class, attributeName="startDate")
@Convert(converter=DateConverter.class, attributeName="endDate")
public class FullTimeEmployee extends GenericEmployee { ... }
```

11.1.11. Converts Annotation

The *Converts* annotation can be used to group *Convert* annotations. Multiple converters must not be applied to the same basic attribute.

```
@Target({METHOD, FIELD, TYPE})
@Retention(RUNTIME)
public @interface Converts {
    Convert[] value();
}
```

Table 14 lists the annotation elements that may be specified for the *Converts* annotation.

Table 14. Converts Annotation Elements

Type	Name	Description	Default
Convert[]	value	(Required) The Convert mappings that are to be applied to the entity or the field or property.	

Example: Multiple converters applied to an embedded attribute

```
@Embedded
@Converts({
    @Convert(converter=CountryConverter.class, attributeName="country"),
    @Convert(converter=CityConverter.class, attributeName="region.city")
})
Address address;
```

11.1.12. DiscriminatorColumn Annotation

For the SINGLE_TABLE mapping strategy, and typically also for the JOINED strategy, the persistence provider will use a type discriminator column. The *DiscriminatorColumn* annotation is used to define the discriminator column for the SINGLE_TABLE and JOINED inheritance mapping strategies.

The strategy and the discriminator column are only specified in the root of an entity class hierarchy or subhierarchy in which a different inheritance strategy is applied. [102: The combination of inheritance strategies within a single entity inheritance hierarchy is not defined by this specification.]

The *DiscriminatorColumn* annotation can be specified on an entity class (including on an abstract entity class).

If the *DiscriminatorColumn* annotation is missing, and a discriminator column is required, the name of the discriminator column defaults to "DTYPE" and the discriminator type to STRING.

Table 15 lists the annotation elements that may be specified for the *DiscriminatorColumn* annotation and their default values.

The supported discriminator types are defined by the *DiscriminatorType* enum:

```
public enum DiscriminatorType { STRING, CHAR, INTEGER };
```

The type of the discriminator column, if specified in the optional *columnDefinition* element, must be consistent with the discriminator type.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface DiscriminatorColumn {
    String name() default "DTYPE";
    DiscriminatorType discriminatorType() default STRING;
    String columnDefinition() default "";
    int length() default 31;
}
```

Table 15. DiscriminatorColumn Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of column to be used for the discriminator.	"DTYPE"
DiscriminatorType	discriminatorType	(Optional) The type of object/column to use as a class discriminator.	DiscriminatorType.STRING
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.
int	length	(Optional) The column length for String-based discriminator types. Ignored for other discriminator types.	31

Example:

```
@Entity
@Table(name="CUST")
@DiscriminatorColumn(name="DISC", discriminatorType=STRING, length=20)
public class Customer { ... }

@Entity
public class ValuedCustomer extends Customer { ... }
```

11.1.13. DiscriminatorValue Annotation

The *DiscriminatorValue* annotation is used to specify the value of the discriminator column for entities of the given type. The *DiscriminatorValue* annotation can only be specified on a concrete entity class. If the *DiscriminatorValue* annotation is not specified and a discriminator column is used, a provider-specific function will be used to generate a value representing the entity type.

The inheritance strategy and the discriminator column are only specified in the root of an entity class hierarchy or subhierarchy in which a different inheritance strategy is applied. The discriminator value, if not defaulted, should be specified for each entity class in the hierarchy.

Table 16 lists the annotation elements that may be specified for the *DiscriminatorValue* annotation and their default values.

The discriminator value must be consistent in type with the discriminator type of the specified or

defaulted discriminator column. If the discriminator type is an integer, the value specified must be able to be converted to an integer value (e.g., "*1*").

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface DiscriminatorValue {
    String value();
}
```

Table 16. DiscriminatorValue Annotation Elements

Type	Name	Description	Default
String	value	(Optional) The value that indicates that the row is an entity of the annotated entity type.	If the DiscriminatorValue annotation is not specified, a provider-specific function to generate a value representing the entity type is used for the value of the discriminator column. If the DiscriminatorType is STRING, the discriminator value default is the entity name.

Example:

```
@Entity
@Table(name="CUST")
@Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorColumn(name="DISC", discriminatorType=STRING,length=20)
@DiscriminatorValue("CUSTOMER")
public class Customer { ... }

@Entity
@DiscriminatorValue("VCUSTOMER")
public class ValuedCustomer extends Customer { ... }
```

11.1.14. ElementCollection Annotation

The *ElementCollection* annotation defines a collection of instances of a basic type or embeddable class. The *ElementCollection* annotation (or equivalent XML element) must be specified if the collection is to

be mapped by means of a collection table. [103: If it is not specified, the rules of Section 2.8 apply.]

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface ElementCollection {
    Class targetClass() default void.class;
    FetchType fetch() default LAZY;
}
```

Table 17 lists the annotation elements that may be specified for the *ElementCollection* annotation and their default values.

Table 17. ElementCollection Annotation Elements

Type	Name	Description	Default
Class	targetClass	(Optional) The basic or embeddable class that is the element type of the collection. Optional only if the collection field or property is defined using Java generics. Must be specified otherwise.	The parameterized type of the collection when defined using generics.
FetchType	fetch	(Optional) Whether the collection should be lazily loaded or must be eagerly fetched. The EAGER strategy is a requirement on the persistence provider runtime that the collection elements must be eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime.	LAZY

Example:

```
@Entity public class Person {
    @Id
    protected String ssn;

    protected String name;

    @ElementCollection
    protected Set<String> nickNames = new HashSet();

    // ...
}
```

11.1.15. Embeddable Annotation

The *Embeddable* annotation is used to specify a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity.

```
@Documented
@Target({TYPE})
@Retention(RUNTIME)
public @interface Embeddable {

}
```

Example 1:

```
@Embeddable
public class EmploymentPeriod {
    @Temporal(DATE)
    java.util.Date startDate;

    @Temporal(DATE)
    java.util.Date endDate;

    // ...
}
```

Example 2:

```
@Embeddable
public class PhoneNumber {
    protected String areaCode;
    protected String localNumber;

    @ManyToOne
    PhoneServiceProvider provider;

    // ...
}

@Entity
public class PhoneServiceProvider {
    @Id
    protected String name;

    // ...
}
```

Example 3:

```
@Embeddable
public class Address {
    protected String street;
    protected String city;
    protected String state;

    @Embedded
    protected Zipcode zipcode;
}

@Embeddable
public class Zipcode {
    protected String zip;
    protected String plusFour;
}
```

11.1.16. Embedded Annotation

The *Embedded* annotation is used to specify a persistent field or property of an entity or embeddable class whose value is an instance of an embeddable class. [104: If the embeddable class is used as a primary key, the *EmbeddedId* rather than the *Embedded* annotation is used.] Each of the persistent properties or fields of the embedded object is mapped to the database table for the entity or embeddable class. The embeddable class must be annotated as *Embeddable*. [105: Use of the *Embedded*

annotation is not required. See Section 2.8.]

The *AttributeOverride*, *AttributeOverrides*, *AssociationOverride*, and *AssociationOverrides* annotations may be used to override mappings declared or defaulted by the embeddable class.

Implementations are not required to support embedded objects that are mapped across more than one table (e.g., split across primary and secondary tables or multiple secondary tables).

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Embedded {}
```

Example:

```
@Embedded
@AttributeOverrides({
    @AttributeOverride(name="startDate", column=@Column(name="EMP_START")),
    @AttributeOverride(name="endDate", column=@Column(name="EMP_END"))
})
public EmploymentPeriod getEmploymentPeriod() { ... }
```

11.1.17. EmbeddedId Annotation

The *EmbeddedId* annotation is applied to a persistent field or property of an entity class or mapped superclass to denote a composite primary key that is an embeddable class. The embeddable class must be annotated as *Embeddable*. [106: Note that the *Id* annotation is not used in the embeddable class.] Relationship mappings defined within an embedded id class are not supported.

There must be only one *EmbeddedId* annotation and no *Id* annotation when the *EmbeddedId* annotation is used.

The *AttributeOverride* annotation may be used to override the column mappings declared within the embeddable class.

The *MapsId* annotation may be used in conjunction with the *EmbeddedId* annotation to specify a derived primary key. See Section 2.4.1 and Section 11.1.37.

If the entity has a derived primary key, the *AttributeOverride* annotation may only be used to override those attributes of the embedded id that do not correspond to the relationship to the parent entity.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface EmbeddedId {}
```

Example 1:

```
@Entity public class Employee {
    @EmbeddedId
    protected EmployeePK empPK;

    String name;

    @ManyToOne
    Set<Department> dept;

    // ...
}
```

Example 2:

```
@Embeddable
public class DependentId {
    String name;
    EmployeeId empPK; // corresponds to PK type of Employee
}

@Entity
public class Dependent {
    // default column name for "name" attribute is overridden
    @AttributeOverride(name="name", @Column(name="dep_name"))
    @EmbeddedId
    DependentId id;

    // ...

    @MapsId("empPK")
    @ManyToOne
    Employee emp;
}
```

11.1.18. Enumerated Annotation

An *Enumerated* annotation specifies that a persistent property or field should be persisted as a enumerated type. The *Enumerated* annotation may be used in conjunction with the *Basic* annotation. The *Enumerated* annotation may be used in conjunction with the _ElementCollection_ [107: If the element collection is a Map, this applies to the map value.] annotation when the element collection value is of basic type.

An enum can be mapped as either a string or an integer [108: Mapping of enum values that contain

state is not supported.]. The *EnumType* enum defines the mapping for enumerated types.

```
public enum EnumType {
    ORDINAL,
    STRING
}
```

If the enumerated type is not specified or the *Enumerated* annotation is not used, the enumerated type is assumed to be *ORDINAL* unless a converter is being applied.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Enumerated {
    EnumType value() default ORDINAL;
}
```

Table 18 lists the annotation elements that may be specified for the *Enumerated* annotation and their default values.

Table 18. Enumerated Annotation Elements

Type	Name	Description	Default
EnumType	value	(Optional) The type used in mapping an enum type.	ORDINAL

Example:

```
public enum EmployeeStatus {FULL_TIME, PART_TIME, CONTRACT}
public enum SalaryRate {JUNIOR, SENIOR, MANAGER, EXECUTIVE}

@Entity
public class Employee {
    // ...

    public EmployeeStatus getStatus() {...}

    @Enumerated(STRING)
    public SalaryRate getPayScale() {...}

    // ...
}
```

If the status property is mapped to a column of integer type, and the payscale property to a column of

varchar type, an instance that has a status of *PART_TIME* and a pay rate of *JUNIOR* will be stored with *STATUS* set to 1 and *PAYSCALE* set to *"JUNIOR"*.

11.1.19. ForeignKey Annotation

The *ForeignKey* annotation is used to specify the handling of foreign key constraints when schema generation is in effect. If this annotation is not specified, the persistence provider's default foreign key strategy will be used.

```
@Target({})
@Retention(RUNTIME)
public @interface ForeignKey {
    String name() default "";
    ConstraintMode value() default CONSTRAINT;
    String foreignKeyDefinition() default "";
}
```

The *name* element specifies a name for the foreign key constraint.

The *ConstraintMode* enum is used to control the application of constraints.

```
public enum ConstraintMode {CONSTRAINT, NO_CONSTRAINT, PROVIDER_DEFAULT}
```

The enum values have the following semantics: A value of *CONSTRAINT* will cause the persistence provider to generate a foreign key constraint. A value of *NO_CONSTRAINT* will result in no constraint being generated. A value of *PROVIDER_DEFAULT* will result in the provider's default behavior (which may or may not result in the generation of a constraint for any given join column or set of join columns).

The syntax used in the *foreignKeyDefinition* element should follow the SQL syntax used by the target database for foreign key constraints. For example, this may be similar to the following:

```
FOREIGN KEY (<COLUMN expression> {, <COLUMN expression>}... )
REFERENCES <TABLE identifier> [ (<COLUMN expression> {, <COLUMN expression>}... ) ]
[ ON UPDATE <referential action> ]
[ ON DELETE <referential action> ]
```

If the *ForeignKey* annotation is specified with a *ConstraintMode* value of *CONSTRAINT*, but the *foreignKeyDefinition* element is not specified, the provider will generate a foreign key constraint whose update and delete actions it determines most appropriate for the join column(s) to which the foreign key constraint is applied

Table 19 lists the annotation elements that may be specified for the *ForeignKey* annotation.

Table 19. ForeignKey Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the foreign key constraint.	A provider-generated name.
ConstraintMode	value	(Optional) Whether to generate a constraint.	CONSTRAINT
String	foreignKeyDefinition	(Optional) The foreign key constraint definition.	Provider-default. If the value of the ConstraintMode element is NO_CONSTRAINT, the provider must not generate a foreign key constraint.

11.1.20. GeneratedValue Annotation

The *GeneratedValue* annotation provides for the specification of generation strategies for the values of primary keys. The *GeneratedValue* annotation may be applied to a primary key property or field of an entity or mapped superclass in conjunction with the *Id* annotation. [109: Portable applications should not use the *GeneratedValue* annotation on other persistent fields or properties.] The use of the *GeneratedValue* annotation is only required to be supported for simple primary keys. Use of the *GeneratedValue* annotation is not supported for derived primary keys.

Table 20 lists the annotation elements that may be specified for the *GeneratedValue* annotation and their default values.

The types of primary key generation are defined by the *GenerationType* enum:

```
public enum GenerationType { TABLE, SEQUENCE, IDENTITY, AUTO };
```

The *TABLE* generator type value indicates that the persistence provider must assign primary keys for the entity using an underlying database table to ensure uniqueness.

The *SEQUENCE* and *IDENTITY* values specify the use of a database sequence or identity column, respectively. [110: Note that SEQUENCE and IDENTITY are not portable across all databases.]

The further specification of table generators and sequence generators is described in Section 11.1.48 and Section 11.1.51.

The *AUTO* value indicates that the persistence provider should pick an appropriate strategy for the particular database. The *AUTO* generation strategy may expect a database resource to exist, or it may attempt to create one. A vendor may provide documentation on how to create such resources in the event that it does not support schema generation or cannot create the schema resource at runtime.

This specification does not define the exact behavior of these strategies.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface GeneratedValue {
    GenerationType strategy() default AUTO;
    String generator() default "";
}
```

Table 20. GeneratedValue Annotation Elements

Type	Name	Description	Default
GenerationType	strategy	(Optional) The primary key generation strategy that the persistence provider must use to generate the annotated entity primary key.	GenerationType.AUTO
String	generator	(Optional) The name of the primary key generator to use as specified in the SequenceGenerator or TableGenerator annotation.	Default primary key generator supplied by persistence provider.

Example 1:

```
@Id
@GeneratedValue(strategy=SEQUENCE, generator="CUST_SEQ")
@Column(name="CUST_ID")
public Long getId() { return id; }
```

Example 2:

```
@Id
@GeneratedValue(strategy=TABLE, generator="CUST_GEN")
@Column(name="CUST_ID")
Long id;
```

11.1.21. Id Annotation

The *Id* annotation specifies the primary key property or field of an entity. The *Id* annotation may be applied in an entity or mapped superclass.

The field or property to which the *Id* annotation is applied should be one of the following types: any Java primitive type; any primitive wrapper type; *java.lang.String*; *java.util.Date*; *java.sql.Date*; *java.math.BigDecimal*; _java.math.BigInteger_ [111: Primary keys using types other than these will not be portable. In general, floating point types should never be used in primary keys.]. See Section 2.4.

The mapped column for the primary key of the entity is assumed to be 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 primary key property or field.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Id {}
```

Example:

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

11.1.22. IdClass Annotation

The *IdClass* annotation is applied to an entity class or a mapped superclass to specify a composite primary key class that is mapped to multiple fields or properties of the entity.

The names of the fields or properties in the primary key class and the primary key fields or properties of the entity must correspond and their types must match according to the rules specified in Section 2.4 and Section 2.4.1.

The *Id* annotation must also be applied to the corresponding fields or properties of the entity.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface IdClass {
    Class value();
}
```

Table 21 lists the annotation elements that may be specified for the *IdClass* annotation.

Table 21. IdClass Annotation Elements

Type	Name	Description	Default
Class	value	(Required) The composite primary key class.	

Example:

```
@IdClass(com.acme.EmployeePK.class)
@Entity
public class Employee {
    @Id
    String empName;

    @Id
    Date birthDay;

    // ...
}
```

11.1.23. Index Annotation

The *Index* annotation is used in schema generation. Note that it is not necessary to specify an index for a primary key, as the primary key index will be created automatically, however, the Index annotation may be used to specify the ordering of the columns in the index for the primary key.

```
@Target({})
@Retention(RUNTIME)
public @interface Index {
    String name() default "";
    String columnList();
    boolean unique() default false;
}
```

The syntax of the *columnList* element is a *column_list*, as follows:

```
column::= index_column [,index_column]*
index_column::= column_name [ASC | DESC]
```

The persistence provider must observe the specified ordering of the columns.

If *ASC* or *DESC* is not specified, *ASC* (ascending order) is assumed.

Table 22 lists the annotation elements that may be specified for the *Index* annotation.

Table 22. Index Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the index.	A provider-generated name.

Type	Name	Description	Default
String	columnList	(Required) The names of the columns to be included in the index.	
boolean	unique	(Optional) Whether the index is unique.	false

11.1.24. Inheritance Annotation

The *Inheritance* annotation defines the inheritance strategy to be used for an entity class hierarchy. It is specified on the entity class that is the root of the entity class hierarchy.

If the *Inheritance* annotation is not specified or if no inheritance type is specified for an entity class hierarchy, the SINGLE_TABLE mapping strategy is used.

Support for the combination of inheritance strategies is not required by this specification. Portable applications should only use a single inheritance strategy within an entity hierarchy.

The three inheritance mapping strategies are the single table per class hierarchy, joined subclass, and table per concrete class strategies. See Section 2.12 for a more detailed discussion of inheritance strategies.

The inheritance strategy options are defined by the *InheritanceType* enum:

```
public enum InheritanceType { SINGLE_TABLE, JOINED, TABLE_PER_CLASS };
```

Support for the TABLE_PER_CLASS mapping strategy is optional in this release.

Table 23 lists the annotation elements that may be specified for the *Inheritance* annotation and their default values.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface Inheritance {
    InheritanceType strategy() default SINGLE_TABLE;
}
```

Table 23. Inheritance Annotation Elements

Type	Name	Description	Default
InheritanceType	strategy	(Optional) The inheritance strategy to use for the entity inheritance hierarchy.	InheritanceType.SINGLE_TABLE

Example:

```
@Entity
@Inheritance(strategy=JOINED)
public class Customer { ... }

@Entity
public class ValuedCustomer extends Customer { ... }
```

11.1.25. JoinColumn Annotation

The *JoinColumn* annotation is used to specify a column for joining an entity association or element collection.

Table 24 lists the annotation elements that may be specified for the *JoinColumn* annotation and their default values.

If the *JoinColumn* annotation itself is defaulted, a single join column is assumed and the default values described in Table 24 apply.

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.

Support for referenced columns that are not primary key columns of the referenced table is optional. Applications that use such mappings will not be portable.

The *foreignKey* annotation element is used to specify or control the generation of a foreign key constraint when schema generation is in effect. If this element is not specified, the persistence provider's default foreign key strategy will apply.

If more than one *JoinColumn* annotation is applied to a field or property, both the *name* and the *referencedColumnName* elements must be specified in each such *JoinColumn* annotation.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(JoinColumns.class)
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 table() default "";
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
}
```

Table 24. 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 using a foreign key mapping strategy, the foreign key column is in the table of the source entity or embeddable. If the join is for a unidirectional OneToMany mapping using a foreign key mapping strategy, the foreign key is in the table of the target entity. If the join is for a ManyToMany mapping or for a OneToOne or bidirectional ManyToOne/OneToMany mapping using a join table, the foreign key is in a join table. If the join is for an element collection, the foreign key is in a collection table.	(Default only applies if a single join column is used.) The concatenation of the following: the name of the referencing relationship property or field of the referencing entity or embeddable class; "*"; the name of the referenced primary key column. If there is no such referencing relationship property or field in the entity, or if the join is for an element collection, the join column name is formed as the concatenation of the following: the name of the 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 entity relationship mappings other than the cases described below, the referenced column is in the table of the target entity. When used with a unidirectional OneToMany foreign key mapping, the referenced column is in the table of the source 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. When used in a collection table mapping, the referenced column is in the table of the entity containing the collection.	(Default 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

Type	Name	Description	Default
boolean	insertable	(Optional) Whether the column is included in SQL INSERT statements generated by the persistence provider.	true
boolean	updatable	(Optional) Whether the column is 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 for the column.
String	table	(Optional) The name of the table that contains the column.	If the join is for a OneToOne or ManyToOne mapping using a foreign key mapping strategy, the name of the table of the source entity or embeddable. If the join is for a unidirectional OneToMany mapping using a foreign key mapping strategy, the name of the table of the target entity. If the join is for a ManyToMany mapping or for a OneToOne or bidirectional ManyToOne/ OneToMany mapping using a join table, the name of the join table. If the join is for an element collection, the name of the collection table.
ForeignKey	foreignKey	(Optional) The foreign key constraint for the join column. This is used only if table generation is in effect.	Provider's default

Example 1:

```
@ManyToOne
@JoinColumn(name="ADDR_ID")
public Address getAddress() { return address; }
```

Example 2: Unidirectional One-to-Many association using a foreign key mapping.

In Customer class:

```
@OneToMany
@JoinColumn(name="CUST_ID") // join column is in table for Order
public Set<Order> getOrders() { return orders; }
```

11.1.26. JoinColumns Annotation

Composite foreign keys are supported by means of the *JoinColumns* annotation. The *JoinColumns* annotation groups *JoinColumn* annotations for the same relationship.

When the *JoinColumns* annotation is used, both the *name* and the *referencedColumnName* elements must be specified in each of the grouped *JoinColumn* annotations.

The *foreignKey* annotation element is used to specify or control the generation of a foreign key constraint when schema generation is in effect. If both this element and the *foreignKey* element of any of the *JoinColumn* elements referenced by the *value* element are specified, the behavior is undefined. If no *foreignKey* annotation element is specified in either location, the persistence provider's default foreign key strategy will apply.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface JoinColumns {
    JoinColumn[] value();
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
}
```

Table 25 lists the annotation elements that may be specified for the *JoinColumns* annotation.

Table 25. JoinColumns Annotation Elements

Type	Name	Description	Default
JoinColumn[]	value	(Required) The join columns that map the relationship.	

Type	Name	Description	Default
ForeignKey	foreignKey	(Optional) The foreign key constraint specification for the join columns. This is used only if table generation is in effect.	Provider's default

Example:

```
@ManyToOne
@JoinColumns({
    @JoinColumn(name="ADDR_ID", referencedColumnName="ID"),
    @JoinColumn(name="ADDR_ZIP", referencedColumnName="ZIP")
})
public Address getAddress() { return address; }
```

11.1.27. JoinTable Annotation

The *JoinTable* annotation is used in the mapping of entity associations. A *JoinTable* annotation is specified on the owning side of the association. A join table is typically used in the mapping of many-to-many and unidirectional one-to-many associations. It may also be used to map bidirectional many-to-one/one-to-many associations, unidirectional many-to-one relationships, and one-to-one associations (both bidirectional and unidirectional).

Table 26 lists the annotation elements that may be specified for the *JoinTable* annotation and their default values.

If the *JoinTable* annotation is not explicitly specified for the mapping of a many-to-many or unidirectional one-to-many relationship, 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.

The *foreignKey* element is used to specify or control the generation of a foreign key constraint for the columns corresponding to the *joinColumns* element when table generation is in effect. If both this element and the *foreignKey* element of any of the *joinColumns* elements are specified, the behavior is undefined. If no *foreignKey* annotation element is specified in either location, the persistence provider's default foreign key strategy will apply. The *inverseForeignKey* element applies to the generation of a foreign key constraint for the columns corresponding to the *inverseJoinColumns* element, and similar rules apply.

When a join table is used in mapping a relationship with an embeddable class on the owning side of the relationship, the containing entity rather than the embeddable class is considered the owner of the relationship.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface JoinTable {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    JoinColumn[] joinColumns() default {};
    JoinColumn[] inverseJoinColumns() default {};
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
    ForeignKey inverseForeignKey() default @ForeignKey(PROVIDER_DEFAULT);
    UniqueConstraint[] uniqueConstraints() default {};
    Index[] indexes() default {};
}
```

Table 26. JoinTable Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the join table.	The concatenated names of the two associated primary entity tables (owning side first), separated by an underscore.
String	catalog	(Optional) The catalog of the table.	Default catalog.
String	schema	(Optional) The schema of the table.	Default schema for user.
JoinColumn[]	joinColumns	(Optional) 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 same defaults as for JoinColumn.
JoinColumn[]	inverseJoinColumns	(Optional) 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 same defaults as for JoinColumn.

Type	Name	Description	Default
ForeignKey	foreignKey	(Optional) The foreign key constraint specification for the join columns. This is used only if table generation is in effect.	Provider's default.
ForeignKey	inverseForeignKey	(Optional) The foreign key constraint specification for the inverse join columns. This is used only if table generation is in effect.	Provider's default.
UniqueConstraint[]	uniqueConstraints	(Optional) Unique constraints that are to be placed on the table. These are only used if table generation is in effect.	No additional constraints
Index[]	indexes	(Optional) Indexes for the table. These are only used if table generation is in effect.	No additional indexes

Example:

```
@JoinTable(
    name="CUST_PHONE",
    joinColumns=@JoinColumn(name="CUST_ID", referencedColumnName="ID"),
    inverseJoinColumns=@JoinColumn(name="PHONE_ID", referencedColumnName="ID")
)
```

11.1.28. 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. Portable applications should use the *Lob* annotation when mapping to a database Lob type. The *Lob* annotation may be used in conjunction with the *Basic* annotation or with the _ElementCollection_ [112: If the element collection is a Map, this applies to the map value.] annotation when the element collection value is of basic type. A Lob may be either a binary or character type. The Lob type is inferred from the type of the persistent field or property and, except for string and character types, defaults to Blob.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Lob {

}
```

Example 1:

```
@Lob
@Basic(fetch=EAGER)
@Column(name="REPORT")
protected String report;
```

Example 2:

```
@Lob @Basic(fetch=LAZY)
@Column(name="EMP_PIC", columnDefinition="BLOB NOT NULL")
protected byte[] pic;
```

11.1.29. 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. If the association is bidirectional, either side may be designated as the owning side. If the relationship is bidirectional, the non-owning side must use the *mappedBy* element of the *ManyToMany* annotation to specify the relationship field or property of the owning side.

The join table for the relationship, if not defaulted, is specified on the owning side.

The *ManyToMany* annotation may be used within an embeddable class contained within an entity class to specify a relationship to a collection of entities [113: The *ManyToMany* annotation must not be used within an embeddable class used in an element collection.]. If the relationship is bidirectional and the entity containing the embeddable class is the owner of the relationship, the non-owning side must use the *mappedBy* element of the *ManyToMany* annotation to specify the relationship field or property of the embeddable class. The dot (".") notation syntax must be used in the *mappedBy* element to indicate the relationship attribute within the embedded attribute. The value of each identifier used with the dot notation is the name of the respective embedded field or property.

Table 27 lists these annotation elements that may be specified for the *ManyToMany* annotation and their default values.

The *cascade* element specifies the set of cascadable operations that are propagated to the associated entity. The operations that are cascadable are defined by the *CascadeType* enum:

```
public enum CascadeType {ALL, PERSIST, MERGE, REMOVE, REFRESH, DETACH};
```

The value *cascade=ALL* is equivalent to *cascade=\{PERSIST, MERGE, REMOVE, REFRESH, DETACH}*.

When the collection is a *java.util.Map*, the *cascade* element applies to the map *value*.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface ManyToMany {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default LAZY;
    String mappedBy() default "";
}
```

The *EAGER* strategy is a requirement on the persistence provider runtime that the associated entity must 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 27. ManyToMany Annotation Elements

Type	Name	Description	Default
Class	targetEntity	(Optional) The entity class that is the target of the association. Optional only if the collection-valued relationship property is defined using Java generics. Must be specified otherwise.	The parameterized type of the collection when defined using generics.
CascadeType[]	cascade	(Optional) The operations that must be cascaded to the target of the association.	No operations are cascaded.

Type	Name	Description	Default
FetchType	fetch	(Optional) Whether the association should be lazily loaded or must be eagerly fetched. The EAGER strategy is a requirement on the persistence provider runtime that the associated entities must be eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime.	LAZY
String	mappedBy	The field or property that owns the relationship. Required unless the relationship is unidirectional.	

Example 1:

In Customer class:

```
@ManyToMany
@JoinTable(name="CUST_PHONES")
public Set<PhoneNumber> getPhones() { return phones; }
```

In PhoneNumber class:

```
@ManyToMany(mappedBy="phones")
public Set<Customer> getCustomers() { return customers; }
```

Example 2:

In Customer class:

```
@ManyToMany(targetEntity=com.acme.PhoneNumber.class)
public Set getPhones() { return phones; }
```

In PhoneNumber class:

```
@ManyToMany(targetEntity=com.acme.Customer.class, mappedBy="phones")
public Set getCustomers() { return customers; }
```

Example 3:

In Customer class:

```
@ManyToMany
@JoinTable(
    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(mappedBy="phones")
public Set<Customer> getCustomers() { return customers; }
```

Example 4:

Embeddable class used by the Employee entity specifies a many-to-many relationship.

```
@Entity
public class Employee {
    @Id
    int id;

    @Embedded
    ContactInfo contactInfo;

    // ...
}

@Embeddable
public class ContactInfo {
    @ManyToOne
    Address address; // Unidirectional

    @ManyToMany
    List<PhoneNumber> phoneNumbers; // Bidirectional
}

@Entity
public class PhoneNumber {
    @Id
    int phNumber;

    @ManyToMany(mappedBy="contactInfo.phoneNumbers")
    Collection<Employee> employees;
}
```

11.1.30. 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 *ManyToOne* annotation may be used within an embeddable class to specify a relationship from the embeddable class to an entity class. If the relationship is bidirectional, the non-owning *OneToMany* entity side must use the *mappedBy* element of the *OneToMany* annotation to specify the relationship field or property of the embeddable field or property on the owning side of the relationship. The dot (".") notation syntax must be used in the *mappedBy* element to indicate the relationship attribute within the embedded attribute. The value of each identifier used with the dot notation is the name of the respective embedded field or property.

Table 28 lists the annotation elements that may be specified for the *ManyToOne* annotation and their default values.

```
@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 operations that can be cascaded are defined by the *CascadeType* enum, defined in Section 11.1.29.

The *EAGER* strategy is a requirement on the persistence provider runtime that the associated entity must 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 28. 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 must be cascaded to the target of the association.	No operations are cascaded.
FetchType	fetch	(Optional) Whether the association should be lazily loaded or must be eagerly fetched. The EAGER strategy is a requirement on the persistence provider runtime that the associated entity must be eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime.	EAGER
boolean	optional	(Optional) Whether the association is optional. If set to false then a non-null relationship must always exist.	true

Example 1:

```
@ManyToOne(optional=false)
@JoinColumn(name="CUST_ID", nullable=false, updatable=false)
public Customer getCustomer() { return customer; }
```

Example 2:

```
@Entity
public class Employee {
    @Id
    int id;

    @Embedded
    JobInfo jobInfo;

    // ...
}

@Embeddable
public class JobInfo {
    String jobDescription;

    @ManyToOne
    ProgramManager pm; // Bidirectional
}

@Entity
public class ProgramManager {
    @Id
    int id;

    @OneToMany(mappedBy="jobInfo.pm")
    Collection<Employee> manages;
}
```

11.1.31. MapKey Annotation

The *MapKey* annotation is used to specify the map key for associations of type *java.util.Map* when the map key is itself the primary key or a persistent field or property of the entity that is the value of the 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 the *name* element is not specified, the primary key of the associated entity 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.

If a persistent field or property other than the primary key is used as a map key, it is expected to be unique within the context of the relationship.

The *MapKeyClass* annotation is not used when *MapKey* is specified and vice versa.

Table 29 lists the annotation elements that may be specified for the *MapKey* annotation.

Table 29. MapKey Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the persistent field or property that is used as the map key.	The primary key is used as the map key.

Example 1:

```
@Entity
public class Department {

    // ...

    @OneToMany(mappedBy="department")
    @MapKey // map key is primary key
    public Map<Integer, Employee> getEmployees() { ... }

    // ...
}

@Entity
public class Employee {

    // ...

    @Id public Integer getEmpId() { ... }
    @ManyToOne
    @JoinColumn(name="dept_id")
    public Department getDepartment() { ... }

    // ...
}
```

Example 2:

```
@Entity
public class Department {
    // ...

    @OneToMany(mappedBy="department")
    @MapKey(name="name")
    public Map<String, Employee> getEmployees() { ... }

    // ...
}

@Entity
public class Employee {
    @Id
    public Integer getEmpId() { ... }

    // ...

    public String getName() { ... }

    // ...

    @ManyToOne
    @JoinColumn(name="dept_id")
    public Department getDepartment() { ... }

    // ...
}
```

11.1.32. MapKeyClass Annotation

The *MapKeyClass* annotation is used to specify the type of the map key for associations of type *java.util.Map* . The map key can be a basic type, an embeddable class, or an entity. If the map is specified using Java generics, the *MapKeyClass* annotation and associated type need not be specified; otherwise they must be specified.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface MapKeyClass {
    Class value();
}
```

The *MapKeyClass* annotation is used in conjunction with *ElementCollection* or one of the collection-valued relationship annotations (*OneToMany* or *ManyToMany*).

The *MapKey* annotation is not used when *MapKeyClass* is specified and vice versa.

Table 30 lists the annotation elements that may be specified for the *MapKeyClass* annotation.

Table 30. MapKeyClass Annotation Elements

Type	Name	Description	Default
Class	value	(Required) The type of the map key.	

Example 1:

```
@Entity
public class Item {
    @Id
    int id;

    // ...

    @ElementCollection(targetClass=String.class)
    @MapKeyClass(String.class)
    Map images; // map from image name to image filename

    // ...
}
```

Example 2:

```
// MapKeyClass and target type of relationship can be defaulted
@Entity
public class Item {
    @Id
    int id;

    // ...

    @ElementCollection
    Map<String, String> images;

    // ...
}
```

Example 3:

```
@Entity
public class Company {
    @Id
    int id;

    // ...

    @OneToMany(targetEntity=com.example.VicePresident.class)
    @MapKeyClass(com.example.Division.class)
    Map organization;
}
```

Example 4:

```
// MapKeyClass and target type of relationship are defaulted
@Entity
public class Company {
    @Id
    int id;

    // ...

    @OneToMany
    Map<Division, VicePresident> organization;
}
```

11.1.33. MapKeyColumn Annotation

The *MapKeyColumn* annotation is used to specify the mapping for the key column of a map whose map key is a basic type. If the *name* element is not specified, it defaults to the concatenation of the following: the name of the referencing relationship field or property; "_"; "*KEY*".

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface MapKeyColumn {
    String name() default "";
    boolean unique() default false;
    boolean nullable() default false;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String table() default "";
    int length() default 255;
    int precision() default 0; // decimal precision
    int scale() default 0; // decimal scale
}
```

If no *MapKeyColumn* annotation is specified, the default values in Table 31 apply.

Table 31. MapKeyColumn Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the map key column. The table in which it is found depends upon the context. If the map key is for an element collection, the map key column is in the collection table for the map value. If the map key is for a ManyToMany entity relationship or for a OneToMany entity relationship using a join table, the map key column is in a join table. If the map key is for a OneToMany entity relationship using a foreign key mapping strategy, the map key column is in the table of the entity that is the value of the map.	The concatenation of the following: the name of the referencing property or field name; "_"; "*KEY*".

Type	Name	Description	Default
boolean	unique	(Optional) Whether the column 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 corresponds to only a single column. This constraint applies in addition to any constraint entailed by primary key mapping and to constraints specified at the table level.	false
boolean	nullable	(Optional) Whether the database column is nullable.	true
boolean	insertable	(Optional) Whether the column is included in SQL INSERT statements generated by the persistence provider.	true
boolean	updatable	(Optional) Whether the column is 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.

Type	Name	Description	Default
String	table	(Optional) The name of the table that contains the column.	If the map key is for an element collection, the name of the collection table for the map value. If the map key is for a OneToMany or ManyToMany entity relationship using a join table, the name of the join table for the map. If the map key is for a OneToMany entity relationship using a foreign key mapping strategy, the name of the primary table of the entity that is the value of the map.
int	length	(Optional) The column length. (Applies only if a string-valued column is used.)	255
int	precision	(Optional) The precision for a decimal (exact numeric) column. (Applies only if a decimal column is used.)	0 (Value must be set by developer.)
int	scale	(Optional) The scale for a decimal (exact numeric) column. (Applies only if a decimal column is used.)	0

Example:

```
@Entity
public class Item {
    @Id
    int id;

    // ...

    @ElementCollection
    @MapKeyColumn(name="IMAGE_NAME")
    @Column(name="IMAGE_FILENAME")
    @CollectionTable(name="IMAGE_MAPPING")
    Map<String, String> images; // map from image name to filename

    // ...
}
```

11.1.34. MapKeyEnumerated Annotation

The *MapKeyEnumerated* annotation is used to specify the enum type for a map key whose basic type is an enumerated type.

The *MapKeyEnumerated* annotation can be applied to an element collection or relationship of type *java.util.Map* , in conjunction with the *ElementCollection*, *OneToMany*, or *ManyToMany* annotation. If the map is specified using Java generics, the *MapKeyClass* annotation and associated type need not be specified; otherwise they must be specified.

If the enumerated type is not specified or the *MapKeyEnumerated* annotation is not used, the enumerated type is assumed to be *ORDINAL*.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface MapKeyEnumerated {
    EnumType value() default ORDINAL;
}
```

Table 32 lists the annotation elements that may be specified for the *MapKeyEnumerated* annotation and their default values. The *EnumType* enum is defined in Section 11.1.18.

Table 32. MapKeyEnumerated Annotation Elements

Type	Name	Description	Default
EnumType	value	(Optional) The type used in mapping an enum type.	ORDINAL

11.1.35. MapKeyJoinColumn Annotation

The *MapKeyJoinColumn* annotation is used to specify a mapping to an entity that is a map key. The map key join column is in the collection table, join table, or table of the target entity that is used to represent the map.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(MapKeyJoinColumns.class)
public @interface MapKeyJoinColumn {
    String name() default "";
    String referencedColumnName() default "";
    boolean unique() default false;
    boolean nullable() default false;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String table() default "";
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
}
```

Table 33 lists the annotation elements that may be specified for the *MapKeyJoinColumn* annotation and their default values.

If no *MapKeyJoinColumn* annotation is specified, a single join column is assumed and the default values described below (and in Table 33) apply.

The *name* annotation element defines the name of the foreign key column. The remaining annotation elements (other than *referencedColumnName*) refer to this column.

If there is a single map key join column, and if the name annotation member is missing, the map key join column name is formed as the concatenation of the following: the name of the referencing relationship property or field of the referencing entity or embeddable; "_"; "*KEY*".

If the *referencedColumnName* element is missing, the foreign key is assumed to refer to the primary key of the referenced table. Support for referenced columns that are not primary key columns of the referenced table is optional. Applications that use such mappings will not be portable.

The *foreignKey* element is used to specify or control the generation of a foreign key constraint for the map key join column when table generation is in effect. If the *foreignKey* element is not specified, the persistence provider's default foreign key strategy will be used.

If more than one *MapKeyJoinColumn* annotation is applied to a field or property, both the *name* and the *referencedColumnName* elements must be specified in each such *MapKeyJoinColumn* annotation.

Table 33. MapKeyJoinColumn Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the foreign key column for the map key. The table in which it is found depends upon the context. If the join is for a map key for an element collection, the foreign key column is in the collection table for the map value. If the join is for a map key for a ManyToMany entity relationship or for a OneToMany entity relationship using a join table, the foreign key column is in a join table. If the join is for a OneToMany entity relationship using a foreign key mapping strategy, the foreign key column for the map key is in the table of the entity that is the value of the map.	(Default only applies if a single join column is used.) The concatenation of the following: the name of the referencing relationship property or field of the referencing entity or embeddable class; "_"; "KEY".
String	referencedColumnName	(Optional) The name of the column referenced by this foreign key column. The referenced column is in the table of the target entity.	(Default 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.	false
boolean	nullable	(Optional) Whether the foreign key column is nullable.	true

Type	Name	Description	Default
boolean	insertable	(Optional) Whether the column is included in SQL INSERT statements generated by the persistence provider.	true
boolean	updatable	(Optional) Whether the column is 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 for the column.
String	table	(Optional) The name of the table that contains the foreign key column. If the join is for a map key for an element collection, the foreign key column is in the collection table for the map value. If the join is for a map key for a ManyToMany entity relationship or for a OneToMany entity relationship using a join table, the foreign key column is in a join table. If the join is for a OneToMany entity relationship using a foreign key mapping strategy, the foreign key column for the map key is in the table of the entity that is the value of the map.	If the map is for an element collection, the name of the collection table for the map value. If the map is for a OneToMany or ManyToMany entity relationship using a join table, the name of the join table for the map. If the map is for a OneToMany entity relationship using a foreign key mapping strategy, the name of the primary table of the entity that is the value of the map.
ForeignKey	foreignKey	(Optional) The foreign key constraint specification for the join column. This is used only if table generation is in effect.	Provider’s default

Example 1:

```
@Entity
public class Company {
    @Id
    int id;

    // ...

    @OneToMany // unidirectional
    @JoinTable(
        name="COMPANY_ORGANIZATION",
        joinColumns=@JoinColumn(name="COMPANY"),
        inverseJoinColumns=@JoinColumn(name="VICEPRESIDENT")
    )
    @MapKeyJoinColumn(name="DIVISION")
    Map<Division, VicePresident> organization;
}
```

Example 2:

```
@Entity
public class VideoStore {
    @Id
    int id;

    String name;

    Address location;

    // ...

    @ElementCollection
    @CollectionTable(name="INVENTORY", joinColumns=@JoinColumn(name="STORE"))
    @Column(name="COPIES_IN_STOCK")
    @MapKeyJoinColumn(name="MOVIE", referencedColumnName="ID")
    Map<Movie, Integer> videoInventory;

    // ...
}

@Entity
public class Movie {
    @Id
    long id;

    String title;

    // ...
}
```

Example 3:

```
@Entity
public class Student {
    @Id
    int studentId;

    // ...

    @ManyToMany // students and courses are also many-many
    @JoinTable(
        name="ENROLLMENTS",
        joinColumns=@JoinColumn(name="STUDENT"),
        inverseJoinColumns=@JoinColumn(name="SEMESTER")
    )
    @MapKeyJoinColumn(name="COURSE")
    Map<Course, Semester> enrollment;

    // ...
}
```

11.1.36. MapKeyJoinColumns Annotation

Composite map keys referencing entities are supported by means of the *MapKeyJoinColumns* annotation. The *MapKeyJoinColumns* annotation groups *MapKeyJoinColumn* annotations.

When the *MapKeyJoinColumns* annotation is used, both the *name* and the *referencedColumnName* elements must be specified in each of the grouped *MapKeyJoinColumn* annotations.

The *foreignKey* element is used to specify or control the generation of a foreign key constraint for the columns corresponding to the *MapKeyJoinColumn* elements referenced by the *value* element when table generation is in effect. If both this element and the *foreignKey* element of any of the *MapKeyJoinColumn* elements are specified, the behavior is undefined. If no *foreignKey* annotation element is specified in either location, the persistence provider's default foreign key strategy will apply.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface MapKeyJoinColumns {
    MapKeyJoinColumn[] value();
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
}
```

Table 34 lists the annotation elements that may be specified for the *MapKeyJoinColumns* annotation.

Table 34. MapKeyJoinColumns Annotation Elements

Type	Name	Description	Default
MapKeyJoinColumn[]	value	(Required) The map key join columns that are used to map to the entity that is the map key.	
ForeignKey	foreignKey	(Optional) The foreign key constraint specification for the join columns. This is used only if table generation is in effect.	Provider's default

11.1.37. MapKeyTemporal Annotation

The *MapKeyTemporal* annotation is used to specify the temporal type for a map key whose basic type is a temporal type.

The *MapKeyTemporal* annotation can be applied to an element collection or relationship of type *java.util.Map* , in conjunction with the *ElementCollection*, *OneToMany*, or *ManyToMany* annotation. If the map is specified using Java generics, the *MapKeyClass* annotation and associated type need not be specified; otherwise they must be specified.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface MapKeyTemporal {
    TemporalType value();
}
```

Table 35 lists the annotation elements that may be specified for the *MapKeyTemporal* annotation and their default values. The *TemporalType* enum is defined in Section 11.1.53.

Table 35. MapKeyTemporal Annotation Elements

Type	Name	Description	Default
TemporalType	value	(Required) The type used in mapping java.util.Date or java.util.Calendar.	

11.1.38. MappedSuperclass Annotation

The *MappedSuperclass* annotation designates a class whose mapping information is applied to the entities that inherit from it. A mapped superclass has no separate table defined for it.

A class designated with the *MappedSuperclass* annotation 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 mapped superclass itself. 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*, *AttributeOverrides*, *AssociationOverride*, and *AssociationOverrides* annotations.

```
@Documented
@Target(TYPE)
@Retention(RUNTIME)
public @interface MappedSuperclass {}
```

11.1.39. MapsId Annotation

The *MapsId* annotation is used to designate a *ManyToOne* or *OneToOne* relationship attribute that provides the mapping for an *EmbeddedId* primary key, an attribute within an *EmbeddedId* primary key, or a simple primary key of the parent entity.

The *value* element specifies the attribute within a composite key to which the relationship attribute corresponds. If the entity's primary key is of the same Java type as the primary key of the entity referenced by the relationship, the *value* attribute is not specified.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface MapsId {
    String value() default "";
}
```

Table 36 lists the annotation elements that may be specified for the *MapsId* annotation.

Table 36. MapsId Annotation Elements

Type	Name	Description	Default
String	value	(Optional) The name of the attribute within the composite key to which the relationship attribute corresponds.	The relationship maps the entity's primary key.

Example:

```
// parent entity has simple primary key
@Entity
public class Employee {
    @Id
    long empId;

    String name;

    // ...
}

// dependent entity uses EmbeddedId for composite key
@Embeddable
public class DependentId {
    String name;
    long empid; // corresponds to PK type of Employee
}

@Entity
public class Dependent {
    @EmbeddedId
    DependentId id;

    // ...

    @MapsId("empid") // maps the empid attribute of embedded id
    @ManyToOne
    Employee emp;
}
```

11.1.40. OneToMany Annotation

A *OneToMany* annotation defines a many-valued association with one-to-many multiplicity.

Table 37 lists the annotation elements that may be specified for the *OneToMany* annotation and their default values.

If the collection is defined using generics to specify the element type, the associated target entity class need not be specified; otherwise it must be specified.

The *OneToMany* annotation may be used within an embeddable class contained within an entity class to specify a relationship to a collection of entities [114: The *OneToMany* annotation must not be used within an embeddable class used in an element collection.]. If the relationship is bidirectional, the *mappedBy* element must be used to specify the relationship field or property of the entity that is the owner of the relationship.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface OneToMany {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default LAZY;
    String mappedBy() default "";
    boolean orphanRemoval() default false;
}
```

The operations that can be cascaded are defined by the *CascadeType* enum, defined in Section 11.1.29.

When the collection is a *java.util.Map*, the *cascade* element and the *orphanRemoval* element apply to the map *value*.

If *orphanRemoval* is *true* and an entity that is the target of the relationship is removed from the relationship (either by removal from the collection or by setting the relationship to null), the remove operation will be applied to the entity being orphaned. If the entity being orphaned is a detached, new, or removed entity, the semantics of *orphanRemoval* do not apply.

If *orphanRemoval* is *true* and the remove operation is applied to the source entity, the remove operation will be cascaded to the relationship target in accordance with the rules of Section 3.2.3, (and hence it is not necessary to specify *cascade=REMOVE* for the relationship) [115: If the parent is detached or new or was previously removed before the orphan was associated with it, the remove operation is not applied to the entity being orphaned.].

The remove operation is applied at the time of the flush operation. The *orphanRemoval* functionality is intended for entities that are privately "owned" by their parent entity. Portable applications must otherwise not depend upon a specific order of removal, and must not reassign an entity that has been orphaned to another relationship or otherwise attempt to persist it.

The default mapping for unidirectional one-to-many relationships uses a join table as is described in Section 2.10.5. Unidirectional one-to-many relationships may be implemented using one-to-many foreign key mappings, using the *JoinColumn* and *JoinColumns* annotations.

Table 37. 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-valued relationship property is defined using Java generics. Must be specified otherwise.	The parameterized type of the collection when defined using generics.
CascadeType[]	cascade	(Optional) The operations that must be cascaded to the target of the association.	No operations are cascaded.
FetchType	fetch	(Optional) Whether the association should be lazily loaded or must be eagerly fetched. The EAGER strategy is a requirement on the persistence provider runtime that the associated entities must be eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime.	LAZY
String	mappedBy	The field or property that owns the relationship. Required unless the relationship is unidirectional.	
boolean	orphanRemoval	(Optional) Whether to apply the remove operation to entities that have been removed from the relationship and to cascade the remove operation to those entities.	false

Example 1: One-to-Many association using generics

In Customer class:

```
@OneToMany(cascade=ALL, mappedBy="customer", orphanRemoval=true)
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",
    orphanRemoval=true
)
public Set getOrders() { return orders; }
```

In Order class:

```
@ManyToOne
@JoinColumn(name="CUST_ID", nullable=false)
protected Customer customer;
```

Example 3: Unidirectional One-to-Many association using a foreign key mapping

In Customer class:

```
@OneToMany(orphanRemoval=true)
@JoinColumn(name="CUST_ID") // join column is in table for Order
public Set<Order> getOrders() { return orders; }
```

11.1.41. 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.

If the relationship is bidirectional, the *mappedBy* element must be used to specify the relationship field or property of the entity that is the owner of the relationship.

The *OneToOne* annotation may be used within an embeddable class to specify a relationship from the embeddable class to an entity class. If the relationship is bidirectional and the entity containing the embeddable class is on the owning side of the relationship, the non-owning side must use the *mappedBy* element of the *OneToOne* annotation to specify the relationship field or property of the embeddable class. The dot (".") notation syntax must be used in the *mappedBy* element to indicate the relationship attribute within the embedded attribute. The value of each identifier used with the dot notation is the name of the respective embedded field or property.

Table 38 lists the annotation elements that may be specified for the *OneToOne* annotation and their default values.

```
@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 "";
    boolean orphanRemoval() default false;
}
```

The operations that can be cascaded are defined by the *CascadeType* enum, defined in Section 11.1.29.

If *orphanRemoval* is *true* and an entity that is the target of the relationship is removed from the relationship (by setting the relationship to null), the remove operation will be applied to the entity being orphaned. If the entity being orphaned is a detached, new, or removed entity, the semantics of *orphanRemoval* do not apply.

If *orphanRemoval* is *true* and the remove operation is applied to the source entity, the remove operation will be cascaded to the relationship target in accordance with the rules of Section 3.2.3, (and hence it is not necessary to specify *cascade=REMOVE* for the relationship) [116: If the parent is detached or new or was previously removed before the orphan was associated with it, the remove operation is not applied to the entity being orphaned.].

The remove operation is applied at the time of the flush operation. The *orphanRemoval* functionality is intended for entities that are privately "owned" by their parent entity. Portable applications must otherwise not depend upon a specific order of removal, and must not reassign an entity that has been orphaned to another relationship or otherwise attempt to persist it.

Table 38. OneToOne 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 must be cascaded to the target of the association.	No operations are cascaded.
FetchType	fetch	(Optional) Whether the association should be lazily loaded or must be eagerly fetched. The EAGER strategy is a requirement on the persistence provider runtime that the associated entity must be eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime.	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 or property that owns the relationship. The mappedBy element is only specified on the inverse (non-owning) side of the association.	
boolean	orphanRemoval	(Optional) Whether to apply the remove operation to entities that have been removed from the relationship and to cascade the remove operation to those entities.	false

Example 1: 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 2: One-to-one association where both source and target share the same primary key values.

On Employee class:

```
@Entity
public class Employee {
    @Id
    Integer id;

    @OneToOne(orphanRemoval=true)
    @MapsId
    EmployeeInfo info;

    // ...
}
```

On EmployeeInfo class:

```
@Entity
public class EmployeeInfo {
    @Id
    Integer id;

    //...
}
```

Example 3: One-to-one association from an embeddable class to another entity.

```
@Entity
public class Employee {
    @Id
    int id;

    @Embedded
    LocationDetails location;

    // ...
}

@Embeddable
public class LocationDetails {
    int officeNumber;

    @OneToOne
    ParkingSpot parkingSpot;

    // ...
}

@Entity
public class ParkingSpot {
    @Id
    int id;

    String garage;

    @OneToOne(mappedBy="location.parkingSpot")
    Employee assignedTo;

    // ...
}
```

11.1.42. OrderBy Annotation

The *OrderBy* annotation specifies the ordering the elements of a collection-valued association or element collection are to have when the association or collection is retrieved.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface OrderBy {
    String value() default "";
}
```

The syntax of the *value* ordering element is an orderby_list, as follows:

```
orderby_list ::= orderby_item [,orderby_item]*
orderby_item ::= [property_or_field_name] [ASC | DESC]
```

If orderby_list is not specified or if *ASC* or *DESC* is not specified, *ASC* (ascending order) is assumed.

If the ordering element is not specified for an entity association, ordering by the primary key of the associated entity is assumed. [117: If the primary key is a composite primary key, the precedence of ordering among the attributes within the primary key is not futher defined. To assign such a precedence within these attributes, each of the individual attributes must be specified as an orderby_item.]

A property or field name specified as an *orderby_item* must correspond to a basic persistent property or field of the associated class or embedded class within it. The properties or fields used in the ordering must correspond to columns for which comparison operators are supported.

The dot (".") notation is used to refer to an attribute within an embedded attribute. The value of each identifier used with the dot notation is the name of the respective embedded field or property.

The *OrderBy* annotation may be applied to an element collection. When *OrderBy* is applied to an element collection of basic type, the ordering will be by value of the basic objects and the *property_or_field_name* is not used. [118: In all other cases when OrderBy is applied to an element collection, the *property_or_field_name* must be specified.] When specifying an ordering over an element collection of embeddable type, the dot notation must be used to specify the attribute or attributes that determine the ordering.

The *OrderBy* annotation is not used when an order column is specified. See Section 11.1.43.

Table 39 lists the annotation elements that may be specified for the *OrderBy* annotation.

Table 39. OrderBy Annotation Elements

Type	Name	Description	Default
String	value	(Optional) The list of attributes (optionally qualified with ASC or DESC) whose values are used in the ordering.	Ascending ordering by the primary key.

Example 1:

```
@Entity
public class Course {
    // ...

    @ManyToMany
    @OrderBy("lastname ASC")
    public List<Student> getStudents() { ... };

    // ...
}
```

Example 2:

```
@Entity
public class Student {
    // ...

    @ManyToMany(mappedBy="students")
    @OrderBy // PK is assumed
    public List<Course> getCourses() { ... };

    // ...
}
```

Example 3:

```
@Entity
public class Person {
    // ...

    @ElementCollection
    @OrderBy("zipcode.zip, zipcode.plusFour")
    public Set<Address> getResidences() { ... };

    // ...
}

@Embeddable
public class Address {
    protected String street;
    protected String city;
    protected String state;

    @Embedded
    protected Zipcode zipcode;
}

@Embeddable
public class Zipcode {
    protected String zip;
    protected String plusFour;
}
```

11.1.43. OrderColumn Annotation

The *OrderColumn* annotation specifies a column that is used to maintain the persistent order of a list. The persistence provider is responsible for maintaining the order upon retrieval and in the database. The persistence provider is responsible for updating the ordering upon flushing to the database to reflect any insertion, deletion, or reordering affecting the list. The *OrderColumn* annotation may be specified on a one-to-many or many-to-many relationship or on an element collection. The *OrderColumn* annotation is specified on the side of the relationship that references the collection that is to be ordered. The order column is not visible as part of the state of the entity or embeddable class. [119: The *OrderBy* annotation should be used for ordering that is visible as persistent state and maintained by the application.]

The *OrderBy* annotation is not used when *OrderColumn* is specified.

Table 40 lists the annotation elements that may be specified for the *OrderColumn* annotation and their default values.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface OrderColumn {
    String name() default "";
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
}
```

If *name* is not specified, the column name is the concatenation of the following: the name of the referencing relationship property or field of the referencing entity or embeddable class; " _ "; " *ORDER* ".

The order column must be of integral type. The persistence provider must maintain a contiguous (non-sparse) ordering of the values of the order column when updating the association or element collection. The order column value for the first element of the list must be 0.

Table 40. OrderColumn Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the ordering column.	The concatenation of the name of the referencing property or field; " _ "; " *ORDER* ".
boolean	nullable	(Optional) Whether the database column is nullable.	true
boolean	insertable	(Optional) Whether the column is included in SQL INSERT statements generated by the persistence provider.	true
boolean	updatable	(Optional) Whether the column is 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.

Example 1:

```
@Entity
public class CreditCard {
    @Id
    long ccNumber;

    @OneToMany // unidirectional
    @OrderColumn
    List<CardTransaction> transactionHistory;

    // ...
}
```

Example 2:

```
@Entity
public class Course {
    // ...

    @ManyToMany
    @JoinTable(name="COURSE_ENROLLMENT")
    public Set<Student> getStudents() { ... };

    // ...

    @ManyToMany // unidirectional
    @JoinTable(name="WAIT_LIST")
    @OrderColumn(name="WAITLIST_ORDER")
    public List<Student> getWaitList()  { ... }
}

@Entity
public class Student {
    // ...

    @ManyToMany(mappedBy="students")
    public Set<Course> getCourses() { ... };

    // ...
}
```

Example of querying the ordered list:

```
SELECT w
FROM course c JOIN c.waitlist w
WHERE c.name = "geometry" AND INDEX(w) = 0
```

11.1.44. PrimaryKeyJoinColumn Annotation

The *PrimaryKeyJoinColumn* annotation specifies a primary key column that is 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; it is used within a *SecondaryTable* annotation to join a secondary table to a primary table; and it may be used in a *OneToOne* mapping in which the primary key of the referencing entity is used as a foreign key [120: It is not expected that a database foreign key be defined for the OneToOne mapping, as the OneToOne relationship may be defined as "optional=true".] to the referenced entity [121: The derived id mechanisms described in Section 2.4.1.1 are now to be preferred over PrimaryKeyJoinColumn for the OneToOne mapping case.].

The *foreignKey* element is used to specify or control the generation of a foreign key constraint for the primary key join column when table generation is in effect. If the *foreignKey* element is not specified, the persistence provider's default foreign key strategy will apply.

Table 41 lists the annotation elements that may be specified for the *PrimaryKeyJoinColumn* annotation and their default values.

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)
@Repeatable(PrimaryKeyJoinColumns.class)
public @interface PrimaryKeyJoinColumn {
    String name() default "";
    String referencedColumnName() default "";
    String columnDefinition() default "";
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
}
```

Table 41. PrimaryKeyJoinColumn Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the primary key column of the current table.	The same name as the primary key column of the primary table of the superclass (JOINED mapping strategy); the same name as the primary key column of the primary table (SecondaryTable mapping); or the same name as the primary key column for the table for the referencing entity (OneToOne mapping).
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 primary table of the superclass (JOINED mapping strategy); the same name as the primary key column of the primary table (SecondaryTable mapping); or the same name as the primary key column of the table for the referenced entity (OneToOne mapping).
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.
ForeignKey	foreignKey	(Optional) The foreign key constraint specification for the join column. This is used only if table generation is in effect.	Provider's default

Example: Customer and ValuedCustomer subclass

```
@Entity
@Table(name="CUST")
@Inheritance(strategy=JOINED)
@DiscriminatorValue("CUST")
public class Customer { ... }

@Entity
@Table(name="VCUST")
@DiscriminatorValue("VCUST")
@PrimaryKeyJoinColumn(name="CUST_ID")
public class ValuedCustomer extends Customer { ... }
```

11.1.45. PrimaryKeyJoinColumns Annotation

Composite foreign keys are supported by means of the *PrimaryKeyJoinColumns* annotation. The *PrimaryKeyJoinColumns* annotation groups *PrimaryKeyJoinColumn* annotations.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface PrimaryKeyJoinColumns {
    PrimaryKeyJoinColumn[] value();
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
}
```

The *foreignKey* element is used to specify or control the generation of a foreign key constraint for the columns corresponding to the *PrimaryKeyJoinColumn* elements referenced by the *value* element when table generation is in effect. If both this element and the *foreignKey* element of any of the *PrimaryKeyJoinColumn* elements are specified, the behavior is undefined. If no *foreignKey* annotation element is specified in either location, the persistence provider's default foreign key strategy will apply.

Table 42 lists the annotation elements that may be specified for the *PrimaryKeyJoinColumns* annotation.

Table 42. PrimaryKeyJoinColumns Annotation Elements

Type	Name	Description	Default
PrimaryKeyJoinColumn[]	value	(Required) The primary key join columns.	

Type	Name	Description	Default
ForeignKey	foreignKey	(Optional) The foreign key constraint specification for the join columns. This is used only if table generation is in effect.	Provider's default

Example 1: ValuedCustomer subclass

```
@Entity
@Table(name="VCUST")
@DiscriminatorValue("VCUST")
@PrimaryKeyJoinColumns({
    @PrimaryKeyJoinColumn(name="CUST_ID", referencedColumnName="ID"),
    @PrimaryKeyJoinColumn(name="CUST_TYPE", referencedColumnName="TYPE")
})
public class ValuedCustomer extends Customer { ... }
```

Example 2: OneToOne relationship between Employee and EmployeeInfo classes. [122: Note that the derived identity mechanisms described in Section 2.4.1.1 is now preferred to the use of PrimaryKeyJoinColumn for this case.]

```
public class EmpPK {
    public Integer id;
    public String name;
}

@Entity
@IdClass(com.acme.EmpPK.class)
public class Employee {
    @Id
    Integer id;

    @Id
    String name;

    @OneToOne
    @PrimaryKeyJoinColumns({
        @PrimaryKeyJoinColumn(name="ID", referencedColumnName="EMP_ID"),
        @PrimaryKeyJoinColumn(name="NAME", referencedColumnName="EMP_NAME")
    })
    EmployeeInfo info;

    // ...
}

@Entity
@IdClass(com.acme.EmpPK.class)
public class EmployeeInfo {
    @Id
    @Column(name="EMP_ID")
    Integer id;

    @Id
    @Column(name="EMP_NAME")
    String name;

    // ...
}
```

11.1.46. SecondaryTable Annotation

The *SecondaryTable* annotation is used to specify a secondary table for the annotated entity class.

If no *SecondaryTable* annotation is specified, it is assumed that all persistent fields or properties of the entity are mapped to the primary table. Specifying one or more secondary tables indicates that the data for the entity class is stored across multiple tables.

Table 43 lists the annotation elements that may be specified for the *SecondaryTable* annotation and their default values.

If no primary key join columns are specified, the join columns are assumed to reference the primary key columns of the primary table, and have the same names and types as the referenced primary key columns of the primary table.

The *foreignKey* element is used to specify or control the generation of a foreign key constraint for the columns corresponding to the *pkJoinColumns* element when table generation is in effect. If both this element and the *foreignKey* element of any of the *pkJoinColumns* elements are specified, the behavior is undefined. If no *foreignKey* annotation element is specified in either location, the persistence provider's default foreign key strategy will apply.

```
@Target({TYPE})
@Retention(RUNTIME)
@Repeatable(SecondaryTables.class)
public @interface SecondaryTable {
    String name();
    String catalog() default "";
    String schema() default "";
    PrimaryKeyJoinColumn[] pkJoinColumns() default {};
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
    UniqueConstraint[] uniqueConstraints() default {};
    Index[] indexes() default {};
}
```

Table 43. 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
PrimaryKeyJoinColumn[]	pkJoinColumns	(Optional) The columns that are used to join with the primary table.	Column(s) of the same name(s) as the primary key column(s) in the primary table
ForeignKey	foreignKey	(Optional) The foreign key constraint for the join column. This is used only if table generation is in effect.	Provider's default

Type	Name	Description	Default
UniqueConstraint[]	uniqueConstraints	(Optional) Unique constraints that are to 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 JoinColumn annotations and constraints entailed by primary key mappings.	No additional constraints
Index[]	indexes	(Optional) Indexes for the table. These are only used if table generation is in effect.	No additional indexes

Example 1: Single secondary table with a single primary key column.

```
@Entity
@Table(name="CUSTOMER")
@SecondaryTable(
    name="CUST_DETAIL",
    pkJoinColumns=@PrimaryKeyJoinColumn(name="CUST_ID")
)
public class Customer { ... }
```

Example 2: Single secondary table with multiple primary key columns.

```
@Entity
@Table(name="CUSTOMER")
@SecondaryTable(
    name="CUST_DETAIL",
    pkJoinColumns={
        @PrimaryKeyJoinColumn(name="CUST_ID"),
        @PrimaryKeyJoinColumn(name="CUST_TYPE")
    })
public class Customer { ... }
```

11.1.47. SecondaryTables Annotation

The *SecondaryTables* annotation can be used to specify multiple secondary tables for an entity.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface SecondaryTables {
    SecondaryTable[] value();
}
```

Table 44 lists the annotation elements that may be specified for the *SecondaryTables* annotation.

Table 44. SecondaryTables Annotation Elements

Type	Name	Description	Default
SecondaryTable[]	value	(Required) The secondary tables that are used to map the entity class.	

Example 1: 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 2: Multiple secondary tables with differently named primary key columns.

```
@Entity
@Table(name="EMPLOYEE")
@SecondaryTables({
    @SecondaryTable(
        name="EMP_DETAIL",
        pkJoinColumns=@PrimaryKeyJoinColumn(name="EMPL_ID")),
    @SecondaryTable(
        name="EMP_HIST",
        pkJoinColumns=@PrimaryKeyJoinColumn(name="EMPLOYEE_ID"))
})
public class Employee { ... }
```

11.1.48. SequenceGenerator Annotation

The *SequenceGenerator* annotation defines a primary key generator that may be referenced by name when a generator element is specified for the *GeneratedValue* annotation. A sequence generator may be specified on the entity class or on the primary key field or property. The scope of the generator name is global to the persistence unit (across all generator types).

Table 45 lists the annotation elements that may be specified for the *SequenceGenerator* annotation and their default values.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(SequenceGenerators.class)
public @interface SequenceGenerator {
    String name();
    String sequenceName() default "";
    String catalog() default "";
    String schema() default "";
    int initialValue() default 1;
    int allocationSize() default 50;
}
```

Table 45. SequenceGenerator Annotation Elements

Type	Name	Description	Default
String	name	(Required) A unique generator name that can be referenced by one or more classes to be the generator for primary key values.	
String	sequenceName	(Optional) The name of the database sequence object from which to obtain primary key values.	A provider- chosen value
String	catalog	(Optional) The catalog of the sequence generator.	Default catalog
String	schema	(Optional) The schema of the sequence generator.	Default schema for user
int	initialValue	(Optional) The value from which the sequence object is to start generating.	1

Type	Name	Description	Default
int	allocationSize	(Optional) The amount to increment by when allocating sequence numbers from the sequence.	50

Example:

```
@SequenceGenerator(name="EMP_SEQ", allocationSize=25)
```

11.1.49. SequenceGenerators Annotation

The SequenceGenerators annotation can be used to specify multiple sequence generators.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface SequenceGenerators {
    SequenceGenerator[] value();
}
```

Table 46. SequenceGenerators Annotation Elements

Type	Name	Description	Default
SequenceGenerator[]	value	(Required) The sequence generator mappings	

11.1.50. Table Annotation

The *Table* annotation specifies the primary table for the annotated entity. Additional tables may be specified by using the *SecondaryTable* or *SecondaryTables* annotation. [123: When a joined inheritance strategy is used, the Table annotation is used to specify a primary table for the subclass-specific state if the default is not used.]

Table 47 lists the annotation elements that may be specified for the *Table* annotation and their default values.

If no *Table* annotation is specified for an entity class, the default values defined in Table 47 apply.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface Table {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    UniqueConstraint[] uniqueConstraints() default {};
    Index[] indexes() default {};
}
```

Table 47. Table Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the table.	Entity name
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 are to 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 and constraints entailed by primary key mappings.	No additional constraints
Index[]	indexes	(Optional) Indexes for the table. These are only used if table generation is in effect.	No additional indexes

Example:

```
@Entity
@Table(name="CUST", schema="RECORDS")
public class Customer { ... }
```

11.1.51. TableGenerator Annotation

The *TableGenerator* annotation defines a primary key generator that may be referenced by name when a generator element is specified for the *GeneratedValue* annotation. A table generator may be specified on the entity class or on the primary key field or property. The scope of the generator name is global to the persistence unit (across all generator types).

Section 6.5.5 lists the annotation elements that may be specified for the *TableGenerator* annotation and their default values.

The *table* element specifies the name of the table that is used by the persistence provider to store generated primary key values for entities. An entity type will typically use its own row in the table for the generation of primary key values. The primary key values are normally positive integers.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(TableGenerators.class)
public @interface TableGenerator {
    String name();
    String table() default "";
    String catalog() default "";
    String schema() default "";
    String pkColumnName() default "";
    String valueColumnName() default "";
    String pkColumnValue() default "";
    int initialValue() default 0;
    int allocationSize() default 50;
    UniqueConstraint[] uniqueConstraints() default {};
    Index[] indexes() default {};
}
```

Table 48. TableGenerator Annotation Elements

Type	Name	Description	Default
String	name	(Required) A unique generator name that can be referenced by one or more classes to be the generator for primary key values.	
String	table	(Optional) Name of table that stores the generated primary key values.	Name is chosen by persistence provider
String	catalog	(Optional) The catalog of the table.	Default catalog

Type	Name	Description	Default
String	schema	(Optional) The schema of the table.	Default schema for user
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 value used to initialize the column that stores the last value generated.	0
int	allocationSize	(Optional) The amount to increment by when allocating numbers from the generator.	50
UniqueConstraint[]	uniqueConstraints	(Optional) Unique constraints that are to be placed on the table. These are only used if table generation is in effect. These constraints apply in addition to primary key constraints.	No additional constraints
Index[]	indexes	(Optional) Indexes for the table. These are only used if table generation is in effect.	No additional indexes

Example 1:

```
@Entity
public class Employee {
    // ...

    @TableGenerator(
        name="empGen",
        table="ID_GEN",
        pkColumnName="GEN_KEY",
        valueColumnName="GEN_VALUE",
        pkColumnValue="EMP_ID",
        allocationSize=1)
    @Id
    @GeneratedValue(strategy=TABLE, generator="empGen")
    int id;

    // ...
}
```

Example 2:

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

    @TableGenerator(
        name="addressGen",
        table="ID_GEN",
        pkColumnName="GEN_KEY",
        valueColumnName="GEN_VALUE",
        pkColumnValue="ADDR_ID")
    @Id
    @GeneratedValue(strategy=TABLE, generator="addressGen")
    int id;

    // ...
}
```

11.1.52. TableGenerators Annotation

The TableGenerators annotation can be used to specify multiple table generators.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface TableGenerators {
    TableGenerator[] value();
}
```

Table 49. TableGenerators Annotation Elements

Type	Name	Description	Default
TableGenerator[]	value	(Required) The table generator mappings	

11.1.53. Temporal Annotation

The *Temporal* annotation must be specified for persistent fields or properties of type *java.util.Date* and *java.util.Calendar* unless a converter is being applied. It may only be specified for fields or properties of these types.

The *Temporal* annotation may be used in conjunction with the *Basic* annotation, the *Id* annotation, or the _ElementCollection_ [124: If the element collection is a Map, this applies to the map value.] annotation (when the element collection value is of such a temporal type).

The *TemporalType* enum defines the mapping for these temporal types.

```
public enum TemporalType {
    DATE, //java.sql.Date
    TIME, //java.sql.Time
    TIMESTAMP //java.sql.Timestamp
}

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Temporal {
    TemporalType value();
}
```

Table 50 lists the annotation elements that may be specified for the *Temporal* annotation and their default values.

Table 50. Temporal Annotation Elements

Type	Name	Description	Default
TemporalType	value	(Required) The type used in mapping java.util.Date or java.util.Calendar.	

Example:

```
@Embeddable
public class EmploymentPeriod {
    @Temporal(DATE)
    java.util.Date startDate;

    @Temporal(DATE)
    java.util.Date endDate;

    // ...
}
```

11.1.54. Transient Annotation

The *Transient* annotation is used to annotate a property or field of an entity class, mapped superclass, or embeddable class. It specifies that the property or field is not persistent.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Transient {}
```

Example:

```
@Entity
public class Employee {
    @Id
    int id;

    @Transient
    User currentUser;

    // ...
}
```

11.1.55. UniqueConstraint Annotation

The *UniqueConstraint* annotation is used to specify that a unique constraint is to be included in the generated DDL for a primary or secondary table.

Table 51 lists the annotation elements that may be specified for the *UniqueConstraint* annotation.

```
@Target({})
@Retention(RUNTIME)
public @interface UniqueConstraint {
    String name() default "";
    String[] columnNames();
}
```

Table 51. UniqueConstraint Annotation Elements

Type	Name	Description	Default
String	name	(Optional) Constraint name.	A provider-chosen name.
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 { ... }
```

11.1.56. Version Annotation

The *Version* annotation specifies the version field or property of an entity class that serves as its optimistic lock value. The version is used to ensure integrity when performing the merge operation and for optimistic concurrency control.

Only a single *Version* property or field should be used per class; applications that use more than one *Version* property or field will not 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 will not be portable.

In general, fields or properties that are specified with the *Version* annotation should not be updated by

the application. [125: See, however, Section 4.10.]

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(name="OPTLOCK")
protected int getVersionNum() { return versionNum; }
```

11.2. Object/Relational Metadata Used in Schema Generation

The following annotations and XML elements define or control the generation of database objects. If schema generation is in effect, the persistence provider must observe the mapping information specified by these annotations and their corresponding XML elements. Unless otherwise specified, all elements of these annotations are observed in the schema generation process.

- *CollectionTable*
- *Column*
- *DiscriminatorColumn*
- *EmbeddedId*
- *Enumerated, MapKeyEnumerated*
- *ForeignKey*
- *GeneratedValue*
- *Id*
- *Index*
- *Inheritance*
- *JoinColumn*
- *JoinTable*
- *Lob*
- *MapKeyColumn*

- *MapKeyJoinColumn*
- *OrderColumn*
- *PrimaryKeyJoinColumn*
- *SecondaryTable*
- *SequenceGenerator*
- *Table*
- *TableGenerator*
- *Temporal, MapKeyTemporal*
- *UniqueConstraint*
- *Version*

In some cases, these annotations and elements may be specified explicitly, while in other cases they may be implied by the default values of other annotations or elements. For example, by default a table is generated corresponding to an entity and bears the same name as that assigned to the entity (which in turn may have been defaulted from the name of the entity class).

The naming of database objects is determined by the defaulting rules and the explicit names used in annotations and/or XML. The names of database objects must be treated in conformance with the requirements of Section 2.13.

The metadata annotations and corresponding XML elements that result in generated objects are as follows.

11.2.1. Table-level elements

The following annotations (and corresponding XML elements) specify the creation of tables. The rules for their naming, columns, and other properties are defined in the referenced sections of this specification:

11.2.1.1. Table

By default, a table is created for every top-level entity and, by default, includes columns corresponding to the basic and embedded attributes of the entity and the foreign keys to the tables of related entities. These columns include columns that result from the use of mapped superclasses, if any. The *SecondaryTable* annotation, in conjunction with the use of the *table* element of the *Column* and *JoinColumn* annotations, is used to override this mapping to partition the state of an entity across multiple tables.

The mapping of the columns of a table is controlled by the *Column* and *JoinColumn* annotations. When entity state is inherited from a mapped superclass, the *AttributeOverride* and *AssociationOverride* annotations may be used to further control the column-level mapping of inherited state. The ordering of the columns is not defined by this specification. When it is desirable to control the ordering of

columns, DDL scripts should be provided.

See Section 11.1.49 for additional rules that apply to the generation of tables. For the treatment of column-level mappings, see further below.

11.2.1.2. Inheritance

The *Inheritance* annotation defines the inheritance strategy for an entity hierarchy. The inheritance strategy determines whether the table for a top-level entity includes columns for entities that inherit from the entity and whether it includes a discriminator column, or whether separate tables are created for each entity type that inherits from the top-level entity. See Section 2.12 and Section 11.1.24 for rules pertaining to the treatment of entity inheritance.

11.2.1.3. SecondaryTable

A secondary table is created to partition the mapping of entity state across multiple tables. See Section 11.1.46 for the rules that apply to the generation of secondary tables.

11.2.1.4. CollectionTable

A collection table is created for the mapping of an element collection. See Section 11.1.8 for the rules that apply to the generation of collection tables. The *Column*, *AttributeOverride*, and *AssociationOverride* annotations may be used to override *CollectionTable* mappings, as described in Section 11.1.9, Section 11.1.4, and Section 11.1.2 respectively.

11.2.1.5. JoinTable

By default, join tables are created for the mapping of many-to-many relationships and unidirectional one-to-many relationships. See Section 2.10.4, Section 2.10.5.1, and Section 2.10.5.2 for the defaults that apply in such cases. Join tables may also be used to map bidirectional many-to-one/one-to-many associations, unidirectional many-to-one relationships, and one-to-one relationships (both bidirectional and unidirectional). See Section 11.1.27 for the rules that apply to the generation of join tables. The *AssociationOverride* annotation may be used to override join table mappings.

11.2.1.6. TableGenerator

Table generator tables are used to store generated primary key values. See Section 11.1.51 for the rules pertaining to table generators.

11.2.2. Column-level elements

The following annotations and corresponding XML elements control the mapping of columns in generated tables.

The exact mapping of Java language types to database-specific types is not defined by this specification, as databases vary in the specific types that they support. In general, however, an implementation of this specification should conform to the “Standard Mapping from Java Types to JDBC Types” as defined

by the JDBC specification [3]. Unless otherwise explicitly specified, however, VARCHAR and VARBINARY mappings should be used in preference to CHAR and BINARY mappings. Applications that are sensitive to the exact database mappings that are generated should use the *columnDefinition* element of the *Column* annotation or include DDL files that specify how the database schema is to be generated.

11.2.2.1. Column

The following elements of the *Column* annotation are used in schema generation:

- *name*
- *unique*
- *nullable*
- *columnDefinition*
- *table*
- *length* (string-valued columns only)
- *precision* (exact numeric (decimal/numeric) columns only)
- *scale* (exact numeric (decimal/numeric) columns only)

See Section 11.1.9 for the rules that apply to these elements and column creation. The *AttributeOverride* annotation may be used to override column mappings.

11.2.2.2. MapKeyColumn

The *MapKeyColumn* annotation specifies the mapping for a key column of a map when the key is of basic type. The following elements of the *MapKeyColumn* annotation are used in schema generation:

- *name*
- *unique*
- *nullable*
- *columnDefinition*
- *table*
- *length* (string-valued columns only)
- *precision* (exact numeric (decimal/numeric) columns only)
- *scale* (exact numeric (decimal/numeric) columns only)

See Section 11.1.33 for the rules that apply to these elements and map key column creation. The *AttributeOverride* annotation may be used to override map key column mappings.

11.2.2.3. Enumerated, MapKeyEnumerated

The *Enumerated* and *MapKeyEnumerated* annotations control whether string- or integer-valued

columns are generated for basic attributes of enumerated types and therefore impact the default column mappings for these types. See Section 11.1.18 and Section 11.1.34. The *Column* and *MapKeyColumn* annotations may be used to further control the column mappings for attributes of enumerated types.

11.2.2.4. Temporal, MapKeyTemporal

The *Temporal* and *MapKeyTemporal* annotations control whether date-, time-, or timestamp-value columns are generated for basic attributes of temporal types, and therefore impact the default column mappings for these types. See Section 11.1.53 and Section 11.1.37. The *Column* and *MapKeyColumn* annotations may be used to further control the column mappings for attributes of temporal types.

11.2.2.5. Lob

The *Lob* annotation specifies that a persistent attribute is to be persisted to a database large object type. See Section 11.1.28. In general, however, the treatment of the *Lob* annotation is provider-dependent. Applications that are sensitive to the exact mapping that is used should use the *columnDefinition* element of the *Column* annotation or include DDL files that specify how the database schema is to be generated.

11.2.2.6. OrderColumn

The *OrderColumn* annotation specifies the generation of a column that is used to maintain the persistent ordering of a list that is represented in an element collection, one-to-many, or many-to-many relationship.

The following elements of the *OrderColumn* annotation are used in schema generation:

- *name*
- *nullable*
- *columnDefinition*

See Section 11.1.43 for the rules that pertain to the generation of order columns.

11.2.2.7. DiscriminatorColumn

A discriminator column is generated for the SINGLE_TABLE mapping strategy and may optionally be generated by the provider for use with the JOINED inheritance strategy. The *DiscriminatorColumn* annotation may be used to control the mapping of the discriminator column. See Section 11.1.12 for the rules that pertain to discriminator columns.

11.2.2.8. Version

The *Version* annotation specifies the generation of a column to serve as an entity's optimistic lock. See Section 11.1.56 for rules that pertain to the version column. The *Column* annotation may be used to further control the column mapping for a version attribute.

11.2.3. Primary Key mappings

Primary keys may be represented by basic or embedded attributes and/or may correspond to foreign key attributes. The *Id* and *EmbeddedId* annotations define attributes whose corresponding columns are the constituents of database primary keys.

11.2.3.1. Id

The *Id* annotation (which may be used used in conjunction with the *IdClass* annotation) is used to specify attributes whose database columns correspond to a primary key. Use of the *Id* annotation results in the creation of a primary key consisting of the corresponding column or columns. Rules for the *Id* annotation are described in Section 11.1.21 and Section 2.4.

The *Column* annotation may be used to further control the column mapping for an *Id* attribute that is applied to a basic type. If the *Id* column was defined in a mapped superclass, the *AttributeOverride* annotation may be used to control the column mapping.

The *JoinColumn* annotation may be used to further control the column mappings for an *Id* attribute that is applied to a relationship that corresponds to a foreign key. If the *Id* attribute was defined in a mapped superclass, the *AssociationOverride* annotation may be used to control the column mapping.

11.2.3.2. EmbeddedId

The *EmbeddedId* annotation specifies an embedded attribute whose corresponding columns correspond to a database primary key. Use of the *EmbeddedId* annotation results in the creation of a primary key consisting of the corresponding columns. Rules for the *EmbeddedId* annotation are described in Section 11.1.17 and Section 2.4.

The *Column* annotation may be used to control the column mapping for an embeddable class. If the *EmbeddedId* attribute is defined in a mapped superclass, the *AttributeOverride* annotation may be used to control the column mappings.

If an *EmbeddedId* attribute corresponds to a relationship attribute, the *MapsId* annotation must be used, and the column mapping is determined by the join column for the relationship. See Section 2.4.1.

11.2.3.3. GeneratedValue

The *GeneratedValue* annotation indicates a primary key whose value is to be generated by the provider. If a strategy is indicated, the provider must use it if it is supported by the target database. Note that specification of the AUTO strategy may result in the provider creating a database object for Id generation (e.g., a database sequence). Rules for the *GeneratedValue* annotation are described in Section 11.1.20. The *GeneratedValue* annotation may only be portably used for simple (i.e., non-composite) primary keys.

11.2.4. Foreign Key Column Mappings

11.2.4.1. JoinColumn

The *JoinColumn* annotation is typically used in specifying a foreign key mapping. In general, the foreign key definitions created will be provider-dependent and database-dependent. Applications that are sensitive to the exact mapping that is used should use the *foreignKey* element of the *JoinColumn* annotation or include DDL files that specify how the database schemas are to be generated.

The following elements of the *JoinColumn* annotation are used in schema generation:

- *name*
- *referencedColumnName*
- *unique*
- *nullable*
- *columnDefinition*
- *table*
- *foreignKey*

See Section 11.1.25 for rules that apply to these elements and join column creation, and sections Section 2.10 and Section 11.1.8 for the rules that apply for the default mappings of foreign keys for relationships and element collections. The *AssociationOverride* annotation may be used to override relationship mappings. The *PrimaryKeyJoinColumn* annotation is used to join secondary tables and may be used in the mapping of one-to-one relationships. See Section 11.2.4.3 below.

11.2.4.2. MapKeyJoinColumn

The *MapKeyJoinColumn* annotation is to specify foreign key mappings to entities that are map keys in map-valued element collections or relationships. In general, the foreign key definitions created should be expected to be provider-dependent and database-dependent. Applications that are sensitive to the exact mapping that is used should use the *foreignKey* element of the *MapKeyJoinColumn* annotation or include DDL files that specify how the database schemas are to be generated.

The following elements of the *MapKeyJoinColumn* annotation are used in schema generation:

- *name*
- *referencedColumnName*
- *unique*
- *nullable*
- *columnDefinition*
- *table*
- *foreignKey*

See Section 11.1.35 for rules that apply to these elements and map key join column creation. The

AssociationOverride annotation may be used to override such mappings.

11.2.4.3. PrimaryKeyJoinColumn

The *PrimaryKeyJoinColumn* annotation specifies that a primary key column is to be used as a foreign key. This annotation is used in the specification of the JOINED mapping strategy and for joining a secondary table to a primary table in a OneToOne relationship mapping. In general, the foreign key definitions created should be expected to be provider-dependent and database-dependent. Applications that are sensitive to the exact mapping that is used should use the *foreignKey* element of the *PrimaryKeyJoinColumn* annotation or include DDL files that specify how the database schemas are to be generated. See Section 11.1.44 for rules pertaining to the *PrimaryKeyJoinColumn* annotation.

11.2.4.4. ForeignKey

The *ForeignKey* annotation may be used within the *JoinColumn*, *JoinColumns*, *MapKeyJoinColumn*, *MapKeyJoinColumns*, *PrimaryKeyJoinColumn*, *PrimaryKeyJoinColumns*, *CollectionTable*, *JoinTable*, *SecondaryTable*, and *AssociationOverride* annotations to specify or override a foreign key constraint. See Section 11.1.19.

11.2.5. Other Elements

11.2.5.1. SequenceGenerator

The *SequenceGenerator* annotation creates a database sequence to be used for Id generation. The use of generators is limited to those databases that support them. See Section 11.1.48.

11.2.5.2. Index

The *Index* annotation generates an index consisting of the specified columns. The ordering of the names in the *columnList* element specified in the *Index* annotation must be observed by the provider when creating the index. See Section 11.1.23.

11.2.5.3. UniqueConstraint

The *UniqueConstraint* annotation generates a unique constraint for the given table. Databases typically implement unique constraints by creating unique indexes. The ordering of the *columnNames* specified in the *UniqueConstraint* annotation must be observed by the provider when creating the constraint. See Section 11.1.55. The *unique* element of the *Column*, *JoinColumn*, *MapKeyColumn*, and *MapKeyJoinColumn* annotations is equivalent to the use of the *UniqueConstraint* annotation when only one column is to be included in the constraint.

11.3. Examples of the Application of Annotations for Object/Relational Mapping

11.3.1. Examples of Simple Mappings

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = 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 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;

    @Id
    @GeneratedValue(strategy = 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;
```

```
    @Id
    @GeneratedValue(strategy = 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 void setCustomer(Customer cust) {
        this.cust = cust;
    }
}

@Entity
@Table(name = "DLVY_SVC")
public class DeliveryService {
```

```
    private String serviceName;
    private int priceCategory;
    private Collection customers;

    @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(name = "CUST_DLVRY")
    public Collection getCustomers() {
        return customers;
    }

    public setCustomers(Collection customers) {
        this.customers = customers;
    }
}
```

11.3.2. A More Complex Example

```
/***** Employee class *****/
@Entity
@Table(name = "EMPL")
@SecondaryTable(name = "EMP_SALARY",
                pkJoinColumns = @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;

    @Id
    @GeneratedValue(strategy = 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(cascade = PERSIST, mappedBy = "employees")
    @JoinTable(
        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", table = "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;

    @Id
    @GeneratedValue(strategy = IDENTITY)
    public Integer getId() {
        return id;
    }

    protected void setId(Integer id) {
        this.id = id;
    }

    @Version
    @Column(name = "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;

    @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", nullable = false)
    public Employee getEmployee() {
        return employee;
    }

    public void setEmployee(Employee employee) {
        this.employee = employee;
    }
}

/***** Project class *****/
@Entity
@Inheritance(strategy = JOINED)
@DiscriminatorValue("Proj")
@DiscriminatorColumn(name = "DISC")
public class Project implements Serializable {
    private Integer projId;
    private int version;
    private String name;
    private Set<Employee> employees;

    @Id
    @GeneratedValue(strategy = 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")
@DiscriminatorValue("GovtProj")
@PrimaryKeyJoinColumn(name = "GOV_PROJ_ID", referencedColumnName = "ID")
public class GovernmentProject extends Project {
    private String fileInfo;

    @Column(name = "INFO")
    public String getFileInfo() {
        return fileInfo;
    }

    public void setFileInfo(String fileInfo) {
        this.fileInfo = fileInfo;
```

```
    }
}

/***** CovertProject subclass *****/
@Entity
@Table(name = "C_PROJECT")
@DiscriminatorValue("CovProj")
@PrimaryKeyJoinColumn(name = "COV_PROJ_ID", referencedColumnName = "ID")
public class CovertProject extends Project {
    private String classified;

    public CovertProject() {
        super();
    }

    public CovertProject(String classified) {
        this();
        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;

    @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 12. XML Object/Relational Mapping Descriptor

The XML object/relational mapping descriptor serves as both an alternative to and an overriding mechanism for Java language metadata annotations.

12.1. Use of the XML Descriptor

The XML schema for the object relational/mapping descriptor is contained in Section 12.3. The root element of this schema is the *entity-mappings* element. The absence or present of the *xml-mapping-metadata-complete* subelement contained in the *persistence-unit-defaults* subelement of the *entity-mappings* element controls whether the XML object/relational mapping descriptor is used to selectively override annotation values or whether it serves as a complete alternative to Java language metadata annotations.

If the *xml-mapping-metadata-complete* subelement is specified, the complete set of mapping metadata for the persistence unit is contained in the XML mapping files for the persistence unit, and any persistence annotations on the classes are ignored.

If *xml-mapping-metadata-complete* is specified and XML elements are omitted, the default values apply. These default values are the same as the corresponding defaults when annotations are used, except in the cases specified in Section 12.2 below. When the *xml-mapping-metadata-complete* element is specified, any *metadata-complete* attributes specified within the *entity*, *mapped-superclass*, and *embeddable* elements are ignored.

If the *xml-mapping-metadata-complete* subelement is not specified, the XML descriptor overrides the values set or defaulted by the use of annotations, as described below.

The mapping files used by the application developer must conform to the XML schema defined in Section 12.3 or to the object/relational mapping schema defined in a previous version of this specification [1], [8].

The Java Persistence persistence provider must support use of older versions of the object/relational mapping schema as well as the object/relational mapping schema defined in Section 12.3, whether singly or in combination when multiple mapping files are used.

12.2. XML Overriding Rules

This section defines the rules that apply when the XML descriptor is used to override annotations, and the rules pertaining to the interaction of XML elements specified as subelements of the *persistence-unit-defaults*, *entity-mappings*, *entity*, *mapped-superclass*, and *embeddable* elements.

12.2.1. persistence-unit-defaults Subelements

12.2.1.1. schema

The *schema* subelement applies to all entities, tables, secondary tables, join tables, collection tables, table generators, and sequence generators in the persistence unit.

The *schema* subelement is overridden by any *schema* subelement of the *entity-mappings* element; any *schema* element explicitly specified in the *Table* or *SecondaryTable* annotation on an entity or any *schema* attribute on any *table* or *secondary-table* subelement defined within an *entity* element; any *schema* element explicitly specified in a *TableGenerator* annotation or *table-generator* subelement; any *schema* element explicitly specified in a *SequenceGenerator* annotation or *sequence-generator* subelement; any *schema* element explicitly specified in a *JoinTable* annotation or *join-table* subelement; and any *schema* element explicitly specified in a *CollectionTable* annotation or *collection-table* subelement.

12.2.1.2. catalog

The *catalog* subelement applies to all entities, tables, secondary tables, join tables, collection tables, table generators, and sequence generators in the persistence unit.

The *catalog* subelement is overridden by any *catalog* subelement of the *entity-mappings* element; any *catalog* element explicitly specified in the *Table* or *SecondaryTable* annotation on an entity or any *catalog* attribute on any *table* or *secondary-table* subelement defined within an *entity* XML element; any *catalog* element explicitly specified in a *TableGenerator* annotation or *table-generator* subelement; any *catalog* element explicitly specified in a *SequenceGenerator* annotation or *sequence-generator* subelement; any *catalog* element explicitly specified in a *JoinTable* annotation or *join-table* subelement; and any *catalog* element explicitly specified in a *CollectionTable* annotation or *collection-table* subelement.

12.2.1.3. delimited-identifiers

The *delimited-identifiers* subelement applies to the naming of database objects, as described in Section 2.13. It specifies that all database table-, schema-, and column-level identifiers in use for the persistence unit be treated as delimited identifiers.

The *delimited-identifiers* subelement cannot be overridden in this release.

12.2.1.4. access

The *access* subelement applies to all managed classes in the persistence unit.

The *access* subelement is overridden by the use of any annotations specifying mapping information on the fields or properties of the entity class; by any *Access* annotation on the entity class, mapped superclass, or embeddable class; by any *access* subelement of the *entity-mappings* element; by any *Access* annotation on a field or property of an entity class, mapped superclass, or embeddable class; by any *access* attribute defined within an *entity*, *mapped-superclass*, or *embeddable* XML element, or by

any *access* attribute defined within an *id*, *embedded-id*, *version*, *basic*, *embedded*, *many-to-one*, *one-to-one*, *one-to-many*, *many-to-many*, or *element-collection* element.

12.2.1.5. cascade-persist

The *cascade-persist* subelement applies to all relationships in the persistence unit.

Specifying this subelement adds the cascade persist option to all relationships in addition to any settings specified in annotations or XML.

The *cascade-persist* subelement cannot be overridden in this release.

The ability to override the *cascade-persist* of the *persistence-unit-defaults* element will be added in a future release of this specification.

12.2.1.6. entity-listeners

The *entity-listeners* subelement defines default entity listeners for the persistence unit. These entity listeners are called before any other entity listeners for an entity unless the entity listener order is overridden within a *mapped-superclass* or *entity* element, or the *ExcludeDefaultListeners* annotation is present on the entity or mapped superclass or the *exclude-default-listeners* subelement is specified within the corresponding *entity* or *mapped-superclass* XML element.

12.2.2. Other Subelements of the entity-mappings element

12.2.2.1. package

The *package* subelement specifies the package of the classes listed within the subelements and attributes of the same mapping file only. The *package* subelement is overridden if the fully qualified class name is specified for a class and the two disagree.

12.2.2.2. schema

The *schema* subelement applies only to the entities, tables, secondary tables, join tables, collection tables, table generators, and sequence generators listed within the same mapping file.

The *schema* subelement is overridden by any *schema* element explicitly specified in the *Table*, *SecondaryTable* , *JoinTable*, or *CollectionTable* annotation on an entity listed within the mapping file or any *schema* attribute on any *table* or *secondary-table* subelement defined within the *entity* element for such an entity, or by any *schema* attribute on any *join-table* or *collection-table* subelement of an attribute defined within the *attributes* subelement of the *entity* element for such an entity, or by the *schema* attribute of any *table-generator* or *sequence-generator* element within the mapping file.

12.2.2.3. catalog

The *catalog* subelement applies only to the entities, tables, secondary tables, join tables, collection tables, table generators, and sequence generators listed within the same mapping file.

The *catalog* subelement is overridden by any *catalog* element explicitly specified in the *Table*, *SecondaryTable* , *JoinTable*, or *CollectionTable* annotation on an entity listed within the mapping file or any *catalog* attribute on any *table* or *secondary-table* subelement defined within the *entity* element for such an entity, or by any *catalog* attribute on any *join-table* or *collection-table* subelement of an attribute defined within the *attributes* subelement of the *entity* element for such an entity, or by the *catalog* attribute of any *table-generator* or *sequence-generator* element within the mapping file.

12.2.2.4. access

The *access* subelement applies to the managed classes listed within the same mapping file.

The *access* subelement is overridden by the use of any annotations specifying mapping information on the fields or properties of the entity class; by any *Access* annotation on the entity class, mapped superclass, or embeddable class; by any *Access* annotation on a field or property of an entity class, mapped superclass, or embeddable class; by any *access* attribute defined within an *entity*, *mapped-superclass*, or *embeddable* XML element, or by any *access* attribute defined within an *id*, *embedded-id*, *version*, *basic*, *embedded*, *many-to-one*, *one-to-one*, *one-to-many*, *many-to-many*, or *element-collection* element.

12.2.2.5. sequence-generator

The generator defined by the *sequence-generator* subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain generators of the same name.

The generator defined is added to any generators defined in annotations. If a generator of the same name is defined in annotations, the generator defined by this subelement overrides that definition.

12.2.2.6. table-generator

The generator defined by the *table-generator* subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain generators of the same name.

The generator defined is added to any generators defined in annotations. If a generator of the same name is defined in annotations, the generator defined by this subelement overrides that definition.

12.2.2.7. named-query

The named query defined by the *named-query* subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain named queries of the same name.

The named query defined is added to the named queries defined in annotations. If a named query of the same name is defined in annotations, the named query defined by this subelement overrides that definition.

12.2.2.8. named-native-query

The named native query defined by the *named-native-query* subelement applies to the persistence unit.

It is undefined if multiple mapping files for the persistence unit contain named queries of the same name.

The named native query defined is added to the named native queries defined in annotations. If a named query of the same name is defined in annotations, the named query defined by this subelement overrides that definition.

12.2.2.9. named-stored-procedure-query

The named stored procedure query defined by the *named-stored-procedure-query* subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain named stored procedure queries of the same name.

The named stored procedure query defined is added to the named stored procedure queries defined in annotations. If a named stored procedure query of the same name is defined in annotations, the named stored procedure query defined by this subelement overrides that definition.

12.2.2.10. sql-result-set-mapping

The SQL result set mapping defined by the *sql-result-set-mapping* subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain SQL result set mappings of the same name.

The SQL result set mapping defined is added to the SQL result set mappings defined in annotations. If a SQL result set mapping of the same name is defined in annotations, the SQL result set mapping defined by this subelement overrides that definition.

12.2.2.11. entity

The *entity* subelement defines an entity of the persistence unit. It is undefined if multiple mapping files for the persistence unit contain entries for the same entity.

The entity class may or may not have been annotated as *Entity*. The subelements and attributes of the *entity* element override as specified in Section 12.2.3.

12.2.2.12. mapped-superclass

The *mapped-superclass* subelement defines a mapped superclass of the persistence unit. It is undefined if multiple mapping files for the persistence unit contain entries for the same mapped superclass.

The mapped superclass may or may not have been annotated as *MappedSuperclass*. The subelements and attributes of the *mapped-superclass* element override as specified in Section 12.2.4.

12.2.2.13. embeddable

The *embeddable* subelement defines an embeddable class of the persistence unit. It is undefined if multiple mapping files for the persistence unit contain entries for the same embeddable class.

The *embeddable* class may or may not have been annotated as *Embeddable*. The subelements and attributes of the *embeddable* element override as specified in Section 12.2.5.

12.2.2.14. converter

The converter defined by the *converter* subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain converters for the same target type.

The converter defined is added to the converters defined in annotations. If a converter for the same target type is defined in annotations, the converter defined by this subelement overrides that definition.

12.2.3. entity Subelements and Attributes

These apply only to the entity for which they are subelements or attributes, unless otherwise specified below.

12.2.3.1. metadata-complete

If the *metadata-complete* attribute of the *entity* element is specified as *true*, any annotations on the entity class (and its fields and properties) are ignored. When *metadata-complete* is specified as *true* and XML attributes or sub-elements of the *entity* element are omitted, the default values for those attributes and elements are applied.

12.2.3.2. access

The *access* attribute defines the access type for the entity. The *access* attribute overrides any access type specified by the *persistence-unit-defaults* element or *entity-mappings* element for the given entity. The access type for a field or property of the entity may be overridden by specifying by overriding the mapping for that field or property using the appropriate XML subelement, as described in Section 12.2.3.26 below.

Caution must be exercised in overriding an access type that was specified or defaulted using annotations, as doing so may cause applications to break.

12.2.3.3. cacheable

The *cacheable* attribute defines whether the entity should be cached or must not be cached when the *shared-cache-mode* element of the *persistence.xml* file is specified as *ENABLE_SELECTIVE* or *DISABLE_SELECTIVE*. If the *Cacheable* annotation was specified for the entity, its value is overridden by this attribute. The value of the *cacheable* attribute is inherited by subclasses (unless otherwise overridden for a subclass by the *Cacheable* annotation or *cacheable* XML attribute).

12.2.3.4. name

The name attribute defines the entity name. The *name* attribute overrides the value of the entity name defined by the *name* element of the *Entity* annotation (whether explicitly specified or defaulted).

Caution must be exercised in overriding the entity name, as doing so may cause applications to break.

12.2.3.5. table

The *table* subelement overrides any *Table* annotation (including defaulted *Table* values) on the entity. If a *table* subelement is present, and attributes or subelements of that *table* subelement are not explicitly specified, their default values are applied.

12.2.3.6. secondary-table

The *secondary-table* subelement overrides all *SecondaryTable* and *SecondaryTables* annotations (including defaulted *SecondaryTable* values) on the entity. If a *secondary-table* subelement is present, and attributes or subelements of that *secondary - table* subelement are not explicitly specified, their default values are applied.

12.2.3.7. primary-key-join-column

The *primary-key-join-column* subelement of the entity element specifies a primary key column that is used to join the table of an entity subclass to the primary table for the entity when the joined strategy is used. The *primary-key-join-column* subelement overrides all *PrimaryKeyJoinColumn* and *PrimaryKeyJoinColumns* annotations (including defaulted *PrimaryKeyJoinColumn* values) on the entity. If a *primary-key-join-column* subelement is present, and attributes or subelements of that *primary-key-join-column* subelement are not explicitly specified, their default values are applied.

12.2.3.8. id-class

The *id-class* subelement overrides any *IdClass* annotation specified on the entity.

12.2.3.9. inheritance

The *inheritance* subelement overrides any *Inheritance* annotation (including defaulted *Inheritance* values) on the entity. If an *inheritance* subelement is present, and the *strategy* attribute is not explicitly specified, its default value is applied.

This element applies to the entity and its subclasses (unless otherwise overridden for a subclass by an annotation or XML element).

Support for the combination of inheritance strategies is not required by this specification. Portable applications should use only a single inheritance strategy within an entity hierarchy.

12.2.3.10. discriminator-value

The *discriminator-value* subelement overrides any *DiscriminatorValue* annotations (including defaulted *DiscriminatorValue* values) on the entity.

12.2.3.11. discriminator-column

The *discriminator-column* subelement overrides any *DiscriminatorColumn* annotation (including defaulted *DiscriminatorColumn* values) on the entity. If a *discriminator-column* subelement is present, and attributes of that *discriminator-column* subelement are not explicitly specified, their default values are applied.

This element applies to the entity and its subclasses (unless otherwise overridden for a subclass by an annotation or XML element).

12.2.3.12. sequence-generator

The generator defined by the *sequence-generator* subelement is added to any generators defined in annotations and any other generators defined in XML. If a generator of the same name is defined in annotations, the generator defined by this subelement overrides that definition. If a *sequence-generator* subelement is present, and attributes or subelements of that *sequence-generator* subelement are not explicitly specified, their default values are applied.

The generator defined by the *sequence-generator* subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain generators of the same name.

12.2.3.13. table-generator

The generator defined by the *table-generator* subelement is added to any generators defined in annotations and any other generators defined in XML. If a generator of the same name is defined in annotations, the generator defined by this subelement overrides that definition. If a *table-generator* subelement is present, and attributes or subelements of that *table-generator* subelement are not explicitly specified, their default values are applied.

The generator defined by the *table-generator* subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain generators of the same name.

12.2.3.14. attribute-override

The *attribute-override* subelement is additive to any *AttributeOverride* or *AttributeOverrides* annotations on the entity. It overrides any *AttributeOverride* elements for the same attribute name. If an *attribute-override* subelement is present, and attributes or subelements of that *attribute-override* subelement are not explicitly specified, their default values are applied.

12.2.3.15. association-override

The *association-override* subelement is additive to any *AssociationOverride* or *AssociationOverrides a* nnotations on the entity. It overrides any *AssociationOverride* elements for the same attribute name. If an *association-override* subelement is present, and attributes or subelements of that *association-override* subelement are not explicitly specified, their default values are applied.

12.2.3.16. convert

The *convert* subelement is additive to any *Convert* or *Converts* annotations on the entity. It overrides any *Convert* annotation for the same attribute name. If a *convert* subelement is present, and attributes or subelements of that *convert* subelement are not explicitly specified, their default values are applied.

12.2.3.17. named-entity-graph

The *named-entity-graph* subelement is additive to any *NamedEntityGraph* annotations on the entity. It overrides any *NamedEntityGraph* annotation with the same name.

12.2.3.18. named-query

The named query defined by the *named-query* subelement is added to any named queries defined in annotations, and any other named queries defined in XML. If a named query of the same name is defined in annotations, the named query defined by this subelement overrides that definition. If a *named-query* subelement is present, and attributes or subelements of that *named-query* subelement are not explicitly specified, their default values are applied.

The named query defined by the *named-query* subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain named queries of the same name.

12.2.3.19. named-native-query

The named query defined by the *named-native-query* subelement is added to any named queries defined in annotations, and any other named queries defined in XML. If a named query of the same name is defined in annotations, the named query defined by this subelement overrides that definition. If a *named-native-query* subelement is present, and attributes or subelements of that *named-native-query* subelement are not explicitly specified, their default values are applied.

The named native query defined by the *named-native-query* subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain named queries of the same name.

12.2.3.20. named-stored-procedure-query

The named stored procedure query defined by the *named-stored-procedure-query* subelement is added to any named stored procedure queries defined in annotations, and any other named stored procedure queries defined in XML. If a named stored procedure query of the same name is defined in annotations, the named stored procedure query defined by this subelement overrides that definition. If a *named-stored-procedure-query* subelement is present, and attributes or subelements of that *named-stored-procedure-query* subelement are not explicitly specified, their default values are applied.

The named stored procedure query defined by the *named-stored-procedure-query* subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain named stored procedure queries of the same name.

12.2.3.21. sql-result-set-mapping

The SQL result set mapping defined by the *sql-result-set-mapping* is added to the SQL result set mappings defined in annotations, and any other SQL result set mappings defined in XML. If a SQL result set mapping of the same name is defined in annotations, the SQL result set mapping defined by this subelement overrides that definition. If a *sql-result-set-mapping* subelement is present, and attributes or subelements of that *sql-result-set-mapping* subelement are not explicitly specified, their default values are applied.

The SQL result set mapping defined by the *sql-result-set-mapping* subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain SQL result set mappings of the same name.

12.2.3.22. exclude-default-listeners

The *exclude-default-listeners* subelement applies whether or not the *ExcludeDefaultListeners* annotation was specified on the entity.

This element causes the default entity listeners to be excluded for the entity and its subclasses.

12.2.3.23. exclude-superclass-listeners

The *exclude-superclass-listeners* subelement applies whether or not the *ExcludeSuperclassListeners* annotation was specified on the entity.

This element causes any superclass listeners to be excluded for the entity and its subclasses.

12.2.3.24. entity-listeners

The *entity-listeners* subelement overrides any *EntityListeners* annotation on the entity.

These listeners apply to the entity and its subclasses unless otherwise excluded.

12.2.3.25. pre-persist, post-persist, pre-remove, post-remove, pre-update, post-update, post-load

These subelements override any lifecycle callback methods defined by the corresponding annotations on the entity.

12.2.3.26. attributes

The *attributes* element groups the mapping subelements for the fields and properties of the entity. It may be sparsely populated to include only a subset of the fields and properties. If the value of *metadata-complete* is *true*, the remainder of the attributes will be defaulted according to the default rules. If *metadata-complete* is not specified, or is *false*, the mappings for only those properties and fields that are explicitly specified will be overridden.

id

The *id* subelement overrides the mapping for the specified field or property. If an *id* subelement is present, and attributes or subelements of that *id* subelement are not explicitly specified, their default values are applied.

embedded-id

The *embedded-id* subelement overrides the mapping for the specified field or property. If an *embedded-id* subelement is present, and attributes or subelements of that *embedded-id* subelement are not explicitly specified, their default values are applied.

basic

The *basic* subelement overrides the mapping for the specified field or property. If a *basic* subelement is present, and attributes or subelements of that *basic* subelement are not explicitly specified, their default values are applied.

version

The *version* subelement overrides the mapping for the specified field or property. If a *version* subelement is present, and attributes or subelements of that *version* subelement are not explicitly specified, their default values are applied.

many-to-one

The *many-to-one* subelement overrides the mapping for the specified field or property. If a *many-to-one* subelement is present, and attributes or subelements of that *many-to-one* subelement are not explicitly specified, their default values are applied.

one-to-many

The *one-to-many* subelement overrides the mapping for the specified field or property. If a *one-to-many* subelement is present, and attributes or subelements of that *one-to-many* subelement are not explicitly specified, their default values are applied.

one-to-one

The *one-to-one* subelement overrides the mapping for the specified field or property. If a *one-to-one* subelement is present, and attributes or subelements of that *one-to-one* subelement are not explicitly specified, their default values are applied.

many-to-many

The *many-to-many* subelement overrides the mapping for the specified field or property. If a *many-to-many* subelement is present, and attributes or subelements of that *many-to-many* subelement are not explicitly specified, their default values are applied.

element-collection

The *element-collection* subelement overrides the mapping for the specified field or property. If an *element-collection* subelement is present, and attributes or subelements of that *element-collection* subelement are not explicitly specified, their default values are applied.

embedded

The *embedded* subelement overrides the mapping for the specified field or property. If an *embedded* subelement is present, and attributes or subelements of that *embedded* subelement are not explicitly specified, their default values are applied.

transient

The *transient* subelement overrides the mapping for the specified field or property.

12.2.4. mapped-superclass Subelements and Attributes

These apply only to the mapped-superclass for which they are subelements or attributes, unless otherwise specified below.

12.2.4.1. metadata-complete

If the *metadata-complete* attribute of the *mapped-superclass* element is specified as *true*, any annotations on the mapped superclass (and its fields and properties) are ignored. When *metadata-complete* is specified as *true* and attributes or sub-elements of the *mapped-superclass* element are omitted, the default values for those attributes and elements are applied.

12.2.4.2. access

The *access* attribute defines the access type for the mapped superclass. The *access* attribute overrides any access type specified by the *persistence-unit-defaults* element or *entity-mappings* element for the given mapped superclass. The access type for a field or property of the mapped superclass may be overridden by specifying by overriding the mapping for that field or property using the appropriate XML subelement, as described in [a17126] below.

Caution must be exercised in overriding an access type that was specified or defaulted using annotations, as doing so may cause applications to break.

12.2.4.3. id-class

The *id-class* subelement overrides any *IdClass* annotation specified on the mapped superclass.

12.2.4.4. exclude-default-listeners

The *exclude-default-listeners* subelement applies whether or not the *ExcludeDefaultListeners* annotation was specified on the mapped superclass.

This element causes the default entity listeners to be excluded for the mapped superclass and its subclasses.

12.2.4.5. exclude-superclass-listeners

The *exclude-superclass-listeners* subelement applies whether or not the *ExcludeSuperclassListeners* annotation was specified on the mapped superclass.

This element causes any superclass listeners to be excluded for the mapped superclass and its subclasses.

12.2.4.6. entity-listeners

The *entity-listeners* subelement overrides any *EntityListeners* annotation on the mapped superclass.

These listeners apply to the mapped superclass and its subclasses unless otherwise excluded.

12.2.4.7. pre-persist, post-persist, pre-remove, post-remove, pre-update, post-update, post-load

These subelements override any lifecycle callback methods defined by the corresponding annotations on the mapped superclass.

12.2.4.8. attributes

The *attributes* element groups the mapping subelements for the fields and properties defined by the mapped superclass. It may be sparsely populated to include only a subset of the fields and properties. If the value of *metadata-complete* is *true*, the remainder of the attributes will be defaulted according to the default rules. If *metadata-complete* is not specified, or is *false*, the mappings for only those properties and fields that are explicitly specified will be overridden.

id

The *id* subelement overrides the mapping for the specified field or property. If an *id* subelement is present, and attributes or subelements of that *id* subelement are not explicitly specified, their default values are applied.

embedded-id

The *embedded-id* subelement overrides the mapping for the specified field or property. If an *embedded-id* subelement is present, and attributes or subelements of that *embedded-id* subelement are not explicitly specified, their default values are applied.

basic

The *basic* subelement overrides the mapping for the specified field or property. If a *basic* subelement is present, and attributes or subelements of that *basic* subelement are not explicitly specified, their default values are applied.

version

The *version* subelement overrides the mapping for the specified field or property. If a *version* subelement is present, and attributes or subelements of that *version* subelement are not explicitly specified, their default values are applied.

many-to-one

The *many-to-one* subelement overrides the mapping for the specified field or property. If a *many-to-one* subelement is present, and attributes or subelements of that *many-to-one* subelement are not explicitly specified, their default values are applied.

one-to-many

The *one-to-many* subelement overrides the mapping for the specified field or property. If a *one-to-many* subelement is present, and attributes or subelements of that *one-to-many* subelement are not explicitly specified, their default values are applied.

one-to-one

The *one-to-one* subelement overrides the mapping for the specified field or property. If a *one-to-one* subelement is present, and attributes or subelements of that *one-to-one* subelement are not explicitly specified, their default values are applied.

many-to-many

The *many-to-many* subelement overrides the mapping for the specified field or property. If a *many-to-many* subelement is present, and attributes or subelements of that *many-to-many* subelement are not explicitly specified, their default values are applied.

element-collection

The *element-collection* subelement overrides the mapping for the specified field or property. If an *element-collection* subelement is present, and attributes or subelements of that *element-collection* subelement are not explicitly specified, their default values are applied.

embedded

The *embedded* subelement overrides the mapping for the specified field or property. If an *embedded* subelement is present, and attributes or subelements of that *embedded* subelement are not explicitly specified, their default values are applied.

transient

The *transient* subelement overrides the mapping for the specified field or property.

12.2.5. embeddable Subelements and Attributes

These apply only to the embeddable for which they are subelements or attributes.

12.2.5.1. metadata-complete

If the *metadata-complete* attribute of the *embeddable* element is specified as *true*, any annotations on the embeddable class (and its fields and properties) are ignored. When *metadata-complete* is specified as *true* and attributes and sub-elements of the *embeddable* element are omitted, the default values for those attributes and elements are applied.

12.2.5.2. access

The *access* attribute defines the access type for the embeddable class. The *access* attribute overrides any access type specified by the *persistence-unit-defaults* element or *entity-mappings* element for the given embeddable class. The access type for a field or property of the embeddable class may be overridden by specifying by overriding the mapping for that field or property using the appropriate XML subelement, as described in Section 12.2.5.3 below.

Caution must be exercised in overriding an access type that was specified or defaulted using annotations, as doing so may cause applications to break.

12.2.5.3. attributes

The *attributes* element groups the mapping subelements for the fields and properties defined by the embeddable class. It may be sparsely populated to include only a subset of the fields and properties. If the value of *metadata-complete* is *true*, the remainder of the attributes will be defaulted according to the default rules. If *metadata-complete* is not specified, or is *false*, the mappings for only those properties and fields that are explicitly specified will be overridden.

basic

The *basic* subelement overrides the mapping for the specified field or property. If a *basic* subelement is present, and attributes or subelements of that *basic* subelement are not explicitly specified, their default values are applied.

many-to-one

The *many-to-one* subelement overrides the mapping for the specified field or property. If a *many-to-one* subelement is present, and attributes or subelements of that *many-to-one* subelement are not explicitly specified, their default values are applied.

one-to-many

The *one-to-many* subelement overrides the mapping for the specified field or property. If a *one-to-many* subelement is present, and attributes or subelements of that *one-to-many* subelement are not explicitly specified, their default values are applied.

one-to-one

The *one-to-one* subelement overrides the mapping for the specified field or property. If a *one-to-one* subelement is present, and attributes or subelements of that *one-to-one* subelement are not explicitly specified, their default values are applied.

many-to-many

The *many-to-many* subelement overrides the mapping for the specified field or property. If a *many-to-many* subelement is present, and attributes or subelements of that *many-to-many* subelement are not explicitly specified, their default values are applied.

element-collection

The *element-collection* subelement overrides the mapping for the specified field or property. If an *element-collection* subelement is present, and attributes or subelements of that *element-collection* subelement are not explicitly specified, their default values are applied.

embedded

The *embedded* subelement overrides the mapping for the specified field or property. If an *embedded* subelement is present, and attributes or subelements of that *embedded* subelement are not explicitly specified, their default values are applied.

transient

The *transient* subelement overrides the mapping for the specified field or property.

12.3. XML Schema

This section provides the XML object/relational mapping schema for use with the persistence API.

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- persistence.xml schema -->
<xsd:schema targetNamespace="https://xmlns.jakarta.ee/xml/ns/persistence"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:persistence="https://xmlns.jakarta.ee/xml/ns/persistence"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  version="3.0">

  <xsd:annotation>
    <xsd:documentation>
      @(#)persistence_3_0.xsd 3.0  February 10, 2020
    </xsd:documentation>
  </xsd:annotation>
```

```
<xsd:annotation>
  <xsd:documentation>

Copyright (c) 2008, 2020 Oracle and/or its affiliates. All rights reserved.

This program and the accompanying materials are made available under the
terms of the Eclipse Public License v. 2.0 which is available at
http://www.eclipse.org/legal/epl-2.0,
or the Eclipse Distribution License v. 1.0 which is available at
http://www.eclipse.org/org/documents/edl-v10.php.

SPDX-License-Identifier: EPL-2.0 OR BSD-3-Clause

  </xsd:documentation>
</xsd:annotation>

 <xsd:annotation>
   <xsd:documentation><![CDATA[

   This is the XML Schema for the persistence configuration file.
   The file must be named "META-INF/persistence.xml" in the
   persistence archive.

   Persistence configuration files must indicate
   the persistence schema by using the persistence namespace:

   https://xmlns.jakarta.ee/xml/ns/persistence

   and indicate the version of the schema by
   using the version element as shown below:

    <persistence xmlns="https://xmlns.jakarta.ee/xml/ns/persistence"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="https://xmlns.jakarta.ee/xml/ns/persistence
        https://xmlns.jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
      version="3.0">
        ...
    </persistence>

  ]]></xsd:documentation>
</xsd:annotation>

<xsd:simpleType name="versionType">
  <xsd:restriction base="xsd:token">
    <xsd:pattern value="[0-9]+(\.[0-9]+)*"/>
  </xsd:restriction>
</xsd:simpleType>
```

```
<!-- **************************************************** -->

<xsd:element name="persistence">
  <xsd:complexType>
    <xsd:sequence>

      <!-- **************************************************** -->

      <xsd:element name="persistence-unit"
                   minOccurs="1" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:annotation>
            <xsd:documentation>

              Configuration of a persistence unit.

            </xsd:documentation>
          </xsd:annotation>
          <xsd:sequence>

          <!-- **************************************************** -->

            <xsd:element name="description" type="xsd:string"
                         minOccurs="0">
              <xsd:annotation>
                <xsd:documentation>

                  Description of this persistence unit.

                </xsd:documentation>
              </xsd:annotation>
            </xsd:element>

            <!-- **************************************************** -->

            <xsd:element name="provider" type="xsd:string"
                         minOccurs="0">
              <xsd:annotation>
                <xsd:documentation>

                  Provider class that supplies EntityManagers for this
                  persistence unit.

                </xsd:documentation>
              </xsd:annotation>
            </xsd:element>
```

```
<!-- **************************************************** -->

<xsd:element name="jta-data-source" type="xsd:string"
             minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      The container-specific name of the JTA datasource to use.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<!-- **************************************************** -->

<xsd:element name="non-jta-data-source" type="xsd:string"
             minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      The container-specific name of a non-JTA datasource to use.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<!-- **************************************************** -->

<xsd:element name="mapping-file" type="xsd:string"
             minOccurs="0" maxOccurs="unbounded">
  <xsd:annotation>
    <xsd:documentation>

      File containing mapping information. Loaded as a resource
      by the persistence provider.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<!-- **************************************************** -->

<xsd:element name="jar-file" type="xsd:string"
             minOccurs="0" maxOccurs="unbounded">
  <xsd:annotation>
    <xsd:documentation>

      Jar file that is to be scanned for managed classes.
```

```
    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<!-- **************************************************** -->

<xsd:element name="class" type="xsd:string"
             minOccurs="0" maxOccurs="unbounded">
  <xsd:annotation>
    <xsd:documentation>

      Managed class to be included in the persistence unit and
      to scan for annotations.  It should be annotated
      with either @Entity, @Embeddable or @MappedSuperclass.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<!-- **************************************************** -->

<xsd:element name="exclude-unlisted-classes" type="xsd:boolean"
             default="true" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      When set to true then only listed classes and jars will
      be scanned for persistent classes, otherwise the
      enclosing jar or directory will also be scanned.
      Not applicable to Java SE persistence units.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<!-- **************************************************** -->

<xsd:element name="shared-cache-mode"
             type="persistence:persistence-unit-caching-type"
             minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      Defines whether caching is enabled for the
      persistence unit if caching is supported by the
      persistence provider. When set to ALL, all entities
      will be cached. When set to NONE, no entities will
```

```
      be cached. When set to ENABLE_SELECTIVE, only entities
      specified as cacheable will be cached. When set to
      DISABLE_SELECTIVE, entities specified as not cacheable
      will not be cached. When not specified or when set to
      UNSPECIFIED, provider defaults may apply.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<!-- **************************************************** -->

<xsd:element name="validation-mode"
             type="persistence:persistence-unit-validation-mode-type"
             minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      The validation mode to be used for the persistence unit.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>


<!-- **************************************************** -->

<xsd:element name="properties" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      A list of standard and vendor-specific properties
      and hints.

    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="property"
                   minOccurs="0" maxOccurs="unbounded">
        <xsd:annotation>
          <xsd:documentation>
            A name-value pair.
          </xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
          <xsd:attribute name="name" type="xsd:string"
                         use="required"/>
```

```
                    <xsd:attribute name="value" type="xsd:string"
                                   use="required"/>
                  </xsd:complexType>
                </xsd:element>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>

        </xsd:sequence>

        <!-- ********************************************************* -->

        <xsd:attribute name="name" type="xsd:string" use="required">
          <xsd:annotation>
            <xsd:documentation>

              Name used in code to reference this persistence unit.

            </xsd:documentation>
          </xsd:annotation>
        </xsd:attribute>

        <!-- ********************************************************* -->

        <xsd:attribute name="transaction-type"
                       type="persistence:persistence-unit-transaction-type">
          <xsd:annotation>
            <xsd:documentation>

              Type of transactions used by EntityManagers from this
              persistence unit.

            </xsd:documentation>
          </xsd:annotation>
        </xsd:attribute>

      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="version" type="persistence:versionType"
                 fixed="3.0" use="required"/>
</xsd:complexType>
</xsd:element>

<!-- ************************************************** -->

<xsd:simpleType name="persistence-unit-transaction-type">
  <xsd:annotation>
```

```
      <xsd:documentation>

        public enum PersistenceUnitTransactionType {JTA, RESOURCE_LOCAL};

      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="JTA"/>
      <xsd:enumeration value="RESOURCE_LOCAL"/>
    </xsd:restriction>
  </xsd:simpleType>

<!-- **************************************************** -->

  <xsd:simpleType name="persistence-unit-caching-type">
    <xsd:annotation>
      <xsd:documentation>

        public enum SharedCacheMode { ALL, NONE, ENABLE_SELECTIVE, DISABLE_SELECTIVE,
UNSPECIFIED};

      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="ALL"/>
      <xsd:enumeration value="NONE"/>
      <xsd:enumeration value="ENABLE_SELECTIVE"/>
      <xsd:enumeration value="DISABLE_SELECTIVE"/>
      <xsd:enumeration value="UNSPECIFIED"/>
    </xsd:restriction>
  </xsd:simpleType>

<!-- **************************************************** -->

  <xsd:simpleType name="persistence-unit-validation-mode-type">
    <xsd:annotation>
      <xsd:documentation>

        public enum ValidationMode { AUTO, CALLBACK, NONE};

      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="AUTO"/>
      <xsd:enumeration value="CALLBACK"/>
      <xsd:enumeration value="NONE"/>
    </xsd:restriction>
  </xsd:simpleType>
```

```
</xsd:schema>
```

Related Documents

- [a19493]JSR-220: Enterprise JavaBeans, v. 3.0. Java Persistence API. *http://jcp.org/en/jsr/detail?id=220*.
- [a19494]SQL 2003, Part 2, Foundation (SQL/Foundation). ISO/IEC 9075-2:2003.
- [a19496]JDBC 4.2 Specification. http://jcp.org/en/jsr/detail?id=221.
- [a19497]Enterprise JavaBeans, v. 2.1. *http://jcp.org/en/jsr/detail?id=153*.
- [a19498]JSR-380: Bean Validation,
  1. 2.0. *http://jcp.org/en/jsr/detail?id=380*.
- [a19499]JSR-366: Java Platform, Enterprise Edition 8 (Java EE 8) Specification. *http://jcp.org/en/jsr/detail?id=366.*
- [a19500]JSR-365: Context and Dependency Injection for Java EE, v 2.0. *http://jcp.org/en/jsr/detail?id=365.*
- [a19501]JSR-317: Java Persistence 2.0. *http://jcp.org/en/jsr/detail?id=317*.

Appendix A: Revision History

This appendix lists the significant changes that have been made during the development of the Java Persistence 2.2 specification.

A.1. Maintenance Release Draft

Created document from Java Persistence 2.1 Final Release specification.

The following annotations have been marked *@Repeatable*:

- AssociationOverride
- AttributeOverride
- Convert
- JoinColumn
- MapKeyJoinColumn
- NamedEntityGraph
- NamedNativeQuery
- NamedQuery
- NamedStoredProcedureQuery
- PersistenceContext
- PersistenceUnit
- PrimaryKeyJoinColumn
- SecondaryTable
- SqlResultSetMapping
- SequenceGenerator
- TableGenerator

Added SequenceGenerators and TableGenerators annotations.

Added support for CDI injection into AttributeConverter classes.

Added support for the mapping of the following java.time types:

- java.time.LocalDate
- java.time.LocalTime
- java.time.LocalDateTime
- java.time.OffsetTime

- java.time.OffsetDateTime

Added default Stream getResultStream() method to Query interface.

Added default Stream<X> getResultStream() method to TypedQuery interface.

Replaced reference to JAR file specification in persistence provider bootstrapping section with more general reference to Java SE service provider requirements.

Updated persistence.xml and orm.xml schemas to 2.2 versions.

Updated Related Documents.

A.2. Jakarta Persistence 3.0

Created document from Java Persistence 2.2 Final Release specification.

The document was converted to *Asciidoc* format.

Packages of all API classes were changed to *jakarta.persistence*. These changes are reflected in the specification document.