PART VI

Persistence

Part VI explores the Java Persistence API. This part contains the following chapters:

◆ ◆ ◆ CHAPTER 32

Introduction to the Java Persistence API

The Java Persistence API provides Java developers with an object/relational mapping facility for managing relational data in Java applications. Java Persistence consists of four areas:

- The Java Persistence API
- The query language
- The Java Persistence Criteria API
- Object/relational mapping metadata

The following topics are addressed here:

Entities

An entity is a lightweight persistence domain object. Typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in that table. The primary programming artifact of an entity is the entity class, although entities can use helper classes.

The persistent state of an entity is represented through either persistent fields or persistent properties. These fields or properties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store.

Requirements for Entity Classes

An entity class must follow these requirements.

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

Persistent Fields and Properties in Entity Classes

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

- Java primitive types
- `java.lang.String`
- Other serializable types, including:
  - Wrappers of Java primitive types
  - `java.math.BigInteger`
  - `java.math.BigDecimal`
  - `java.util.Date`
  - `java.util.Calendar`
  - `java.sql.Date`
  - `java.sql.Time`
  - `java.sql.TimeStamp`
  - User-defined serializable types
  - `byte[]`
  - `Byte[]`
  - `char[]`
  - `Character[]`
- Enumerated types

- Other entities and/or collections of entities
- Embeddable classes

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

Persistent Fields

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

Persistent Properties

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

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

```
Type getProperty()
void setProperty(Type type)
```

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

Using Collections in Entity Fields and Properties

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

- `java.util.Collection`
- `java.util.Set`

- `java.util.List`
- `java.util.Map`

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

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

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

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

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

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

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

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

- The `Map` key or value may be a basic Java programming language type, an embeddable class, or an entity.
- When the `Map` value is an embeddable class or basic type, use the `@ElementCollection` annotation.
- When the `Map` value is an entity, use the `@OneToMany` or `@ManyToMany` annotation.
- Use the `Map` type on only one side of a bidirectional relationship.

If the key type of a `Map` is a Java programming language basic type, use the annotation `javax.persistence.MapKeyColumn` to set the column mapping for the key. By default, the `name`

attribute of `@MapKeyColumn` is of the form *RELATIONSHIP-FIELD/PROPERTY-NAME*`_KEY`. For example, if the referencing relationship field name is `image`, the default `name` attribute is `IMAGE_KEY`.

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

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

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

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

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

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

Validating Persistent Fields and Properties

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

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

Bean Validation constraints are annotations applied to the fields or properties of Java programming language classes. Bean Validation provides a set of constraints as well as an API for defining custom constraints. Custom constraints can be specific combinations of the default

constraints, or new constraints that don't use the default constraints. Each constraint is associated with at least one validator class that validates the value of the constrained field or property. Custom constraint developers must also provide a validator class for the constraint.

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

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

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

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

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

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

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

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

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

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

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

Primary Keys in Entities

Each entity has a unique object identifier. A customer entity, for example, might be identified by a customer number. The unique identifier, or *primary key*, enables clients to locate a particular entity instance. Every entity must have a primary key. An entity may have either a simple or a composite primary key.

Simple primary keys use the `javax.persistence.Id` annotation to denote the primary key property or field.

Composite primary keys are used when a primary key consists of more than one attribute, which corresponds to a set of single persistent properties or fields. Composite primary keys must be defined in a primary key class. Composite primary keys are denoted using the `javax.persistence.EmbeddedId` and `javax.persistence.IdClass` annotations.

The primary key, or the property or field of a composite primary key, must be one of the following Java language types:

- Java primitive types
- Java primitive wrapper types
- `java.lang.String`
- `java.util.Date` (the temporal type should be `DATE`)
- `java.sql.Date`
- `java.math.BigDecimal`
- `java.math.BigInteger`

Floating-point types should never be used in primary keys. If you use a generated primary key, only integral types will be portable.

A primary key class must meet these requirements.

- The access control modifier of the class must be `public`.
- The properties of the primary key class must be `public` or `protected` if property-based access is used.
- The class must have a public default constructor.
- The class must implement the `hashCode()` and `equals(Object other)` methods.
- The class must be serializable.
- A composite primary key must be represented and mapped to multiple fields or properties of the entity class or must be represented and mapped as an embeddable class.
- If the class is mapped to multiple fields or properties of the entity class, the names and types of the primary key fields or properties in the primary key class must match those of the entity class.

The following primary key class is a composite key, and the `orderId` and `itemId` fields together uniquely identify an entity:

```
public final class LineItemKey implements Serializable {
    public Integer orderId;
    public int itemId;

    public LineItemKey() {}

    public LineItemKey(Integer orderId, int itemId) {
        this.orderId = orderId;
        this.itemId = itemId;
    }

    public boolean equals(Object otherOb) {
        if (this == otherOb) {
            return true;
        }
        if (!(otherOb instanceof LineItemKey)) {
            return false;
        }
        LineItemKey other = (LineItemKey) otherOb;
        return (
                    (orderId==null?other.orderId==null:orderId.equals
                    (other.orderId)
                    )
                    &&
                    (itemId == other.itemId)
                );
    }

    public int hashCode() {
        return (
                    (orderId==null?0:orderId.hashCode())
                    ^
                    ((int) itemId)
                );
```

```
    }

    public String toString() {
        return "" + orderId + "-" + itemId;
    }
}
```

Multiplicity in Entity Relationships

Multiplicities are of the following types: one-to-one, one-to-many, many-to-one, and many-to-many:

- **One-to-one**: Each entity instance is related to a single instance of another entity. For example, to model a physical warehouse in which each storage bin contains a single widget, `StorageBin` and `Widget` would have a one-to-one relationship. One-to-one relationships use the `javax.persistence.OneToOne` annotation on the corresponding persistent property or field.
- **One-to-many**: An entity instance can be related to multiple instances of the other entities. A sales order, for example, can have multiple line items. In the order application, `Order` would have a one-to-many relationship with `LineItem`. One-to-many relationships use the `javax.persistence.OneToMany` annotation on the corresponding persistent property or field.
- **Many-to-one**: Multiple instances of an entity can be related to a single instance of the other entity. This multiplicity is the opposite of a one-to-many relationship. In the example just mentioned, the relationship to `Order` from the perspective of `LineItem` is many-to-one. Many-to-one relationships use the `javax.persistence.ManyToOne` annotation on the corresponding persistent property or field.
- **Many-to-many**: The entity instances can be related to multiple instances of each other. For example, each college course has many students, and every student may take several courses. Therefore, in an enrollment application, `Course` and `Student` would have a many-to-many relationship. Many-to-many relationships use the `javax.persistence.ManyToMany` annotation on the corresponding persistent property or field.

Direction in Entity Relationships

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

Bidirectional Relationships

In a *bidirectional* relationship, each entity has a relationship field or property that refers to the other entity. Through the relationship field or property, an entity class's code can access its

related object. If an entity has a related field, the entity is said to "know" about its related object. For example, if `Order` knows what `LineItem` instances it has and if `LineItem` knows what `Order` it belongs to, they have a bidirectional relationship.

Bidirectional relationships must follow these rules.

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

Unidirectional Relationships

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

Queries and Relationship Direction

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

Cascade Operations and Relationships

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

The `javax.persistence.CascadeType` enumerated type defines the cascade operations that are applied in the `cascade` element of the relationship annotations. Table 32–1 lists the cascade operations for entities.

TABLE 32–1 Cascade Operations for Entities

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

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

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

Orphan Removal in Relationships

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

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

The following example will cascade the remove operation to the orphaned `order` entity when the `customer` entity is deleted:

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

Embeddable Classes in Entities

Embeddable classes are used to represent the state of an entity but don't have a persistent identity of their own, unlike entity classes. Instances of an embeddable class share the identity of the entity that owns it. Embeddable classes exist only as the state of another entity. An entity may have single-valued or collection-valued embeddable class attributes.

Embeddable classes have the same rules as entity classes but are annotated with the `javax.persistence.Embeddable` annotation instead of `@Entity`.

The following embeddable class, `ZipCode`, has the fields `zip` and `plusFour`:

```
@Embeddable
public class ZipCode {
  String zip;
  String plusFour;
...
}
```

This embeddable class is used by the `Address` entity:

```
@Entity
public class Address {
  @Id
  protected long id
  String street1;
  String street2;
  String city;
  String province;
  @Embedded
  ZipCode zipCode;
  String country;
...
}
```

Entities that own embeddable classes as part of their persistent state may annotate the field or property with the `javax.persistence.Embedded` annotation but are not required to do so.

Embeddable classes may themselves use other embeddable classes to represent their state. They may also contain collections of basic Java programming language types or other embeddable classes. Embeddable classes may also contain relationships to other entities or collections of entities. If the embeddable class has such a relationship, the relationship is from the target entity or collection of entities to the entity that owns the embeddable class.

Entity Inheritance

Entities support class inheritance, polymorphic associations, and polymorphic queries. Entity classes can extend non-entity classes, and non-entity classes can extend entity classes. Entity classes can be both abstract and concrete.

The `roster` example application demonstrates entity inheritance, as described in "Entity Inheritance in the `roster` Application" on page 617.

Abstract Entities

An abstract class may be declared an entity by decorating the class with `@Entity`. Abstract entities are like concrete entities but cannot be instantiated.

Abstract entities can be queried just like concrete entities. If an abstract entity is the target of a query, the query operates on all the concrete subclasses of the abstract entity:

```
@Entity
public abstract class Employee {
    @Id
    protected Integer employeeId;
    ...
}
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}
@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
}
```

Mapped Superclasses

Entities may inherit from superclasses that contain persistent state and mapping information but are not entities. That is, the superclass is not decorated with the `@Entity` annotation and is not mapped as an entity by the Java Persistence provider. These superclasses are most often used when you have state and mapping information common to multiple entity classes.

Mapped superclasses are specified by decorating the class with the annotation `javax.persistence.MappedSuperclass`:

```
@MappedSuperclass
public class Employee {
    @Id
    protected Integer employeeId;
```

```
    ...
}
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}
@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
    ...
}
```

Mapped superclasses cannot be queried and can't be used in `EntityManager` or `Query` operations. You must use entity subclasses of the mapped superclass in `EntityManager` or `Query` operations. Mapped superclasses can't be targets of entity relationships. Mapped superclasses can be abstract or concrete.

Mapped superclasses do not have any corresponding tables in the underlying datastore. Entities that inherit from the mapped superclass define the table mappings. For instance, in the preceding code sample, the underlying tables would be `FULLTIMEEMPLOYEE` and `PARTTIMEEMPLOYEE`, but there is no `EMPLOYEE` table.

Non-Entity Superclasses

Entities may have non-entity superclasses, and these superclasses can be either abstract or concrete. The state of non-entity superclasses is nonpersistent, and any state inherited from the non-entity superclass by an entity class is nonpersistent. Non-entity superclasses may not be used in `EntityManager` or `Query` operations. Any mapping or relationship annotations in non-entity superclasses are ignored.

Entity Inheritance Mapping Strategies

You can configure how the Java Persistence provider maps inherited entities to the underlying datastore by decorating the root class of the hierarchy with the annotation `javax.persistence.Inheritance`. The following mapping strategies are used to map the entity data to the underlying database:

- A single table per class hierarchy
- A table per concrete entity class
- A "join" strategy, whereby fields or properties that are specific to a subclass are mapped to a different table than the fields or properties that are common to the parent class

The strategy is configured by setting the `strategy` element of `@Inheritance` to one of the options defined in the `javax.persistence.InheritanceType` enumerated type:

```
public enum InheritanceType {
    SINGLE_TABLE,
    JOINED,
    TABLE_PER_CLASS
};
```

The default strategy, `InheritanceType.SINGLE_TABLE`, is used if the `@Inheritance` annotation is not specified on the root class of the entity hierarchy.

The Single Table per Class Hierarchy Strategy

With this strategy, which corresponds to the default `InheritanceType.SINGLE_TABLE`, all classes in the hierarchy are mapped to a single table in the database. This table has a *discriminator column* containing a value that identifies the subclass to which the instance represented by the row belongs.

The discriminator column, whose elements are shown in Table 32–2, can be specified by using the `javax.persistence.DiscriminatorColumn` annotation on the root of the entity class hierarchy.

TABLE 32–2 @DiscriminatorColumn Elements

Type	Name	Description
`String`	`name`	The name of the column to be used as the discriminator column. The default is `DTYPE`. This element is optional.
`DiscriminatorType`	`discriminatorType`	The type of the column to be used as a discriminator column. The default is `DiscriminatorType.STRING`. This element is optional.
`String`	`columnDefinition`	The SQL fragment to use when creating the discriminator column. The default is generated by the Persistence provider and is implementation-specific. This element is optional.
`String`	`length`	The column length for `String`-based discriminator types. This element is ignored for non-`String` discriminator types. The default is 31. This element is optional.

The `javax.persistence.DiscriminatorType` enumerated type is used to set the type of the discriminator column in the database by setting the `discriminatorType` element of `@DiscriminatorColumn` to one of the defined types. `DiscriminatorType` is defined as:

```
public enum DiscriminatorType {
    STRING,
    CHAR,
    INTEGER
};
```

If `@DiscriminatorColumn` is not specified on the root of the entity hierarchy and a discriminator column is required, the Persistence provider assumes a default column name of `DTYPE` and column type of `DiscriminatorType.STRING`.

The `javax.persistence.DiscriminatorValue` annotation may be used to set the value entered into the discriminator column for each entity in a class hierarchy. You may decorate only concrete entity classes with `@DiscriminatorValue`.

If `@DiscriminatorValue` is not specified on an entity in a class hierarchy that uses a discriminator column, the Persistence provider will provide a default, implementation-specific value. If the `discriminatorType` element of `@DiscriminatorColumn` is `DiscriminatorType.STRING`, the default value is the name of the entity.

This strategy provides good support for polymorphic relationships between entities and queries that cover the entire entity class hierarchy. However, this strategy requires the columns that contain the state of subclasses to be nullable.

The Table per Concrete Class Strategy

In this strategy, which corresponds to `InheritanceType.TABLE_PER_CLASS`, each concrete class is mapped to a separate table in the database. All fields or properties in the class, including inherited fields or properties, are mapped to columns in the class's table in the database.

This strategy provides poor support for polymorphic relationships and usually requires either SQL `UNION` queries or separate SQL queries for each subclass for queries that cover the entire entity class hierarchy.

Support for this strategy is optional and may not be supported by all Java Persistence API providers. The default Java Persistence API provider in the GlassFish Server does not support this strategy.

The Joined Subclass Strategy

In this strategy, which corresponds to `InheritanceType.JOINED`, the root of the class hierarchy is represented by a single table, and each subclass has a separate table that contains only those fields specific to that subclass. That is, the subclass table does not contain columns for inherited fields or properties. The subclass table also has a column or columns that represent its primary key, which is a foreign key to the primary key of the superclass table.

This strategy provides good support for polymorphic relationships but requires one or more join operations to be performed when instantiating entity subclasses. This may result in poor performance for extensive class hierarchies. Similarly, queries that cover the entire class hierarchy require join operations between the subclass tables, resulting in decreased performance.

Some Java Persistence API providers, including the default provider in the GlassFish Server, require a discriminator column that corresponds to the root entity when using the joined subclass strategy. If you are not using automatic table creation in your application, make sure

that the database table is set up correctly for the discriminator column defaults, or use the `@DiscriminatorColumn` annotation to match your database schema. For information on discriminator columns, see "The Single Table per Class Hierarchy Strategy" on page 593.

Managing Entities

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

The `EntityManager` Interface

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

Container-Managed Entity Managers

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

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

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

```
@PersistenceContext
EntityManager em;
```

Application-Managed Entity Managers

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

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

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

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

```
@PersistenceUnit
EntityManagerFactory emf;
```

Then obtain an `EntityManager` from the `EntityManagerFactory` instance:

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

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

```
@Resource
UserTransaction utx;
```

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

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

```
@PersistenceContext
EntityManagerFactory emf;
EntityManager em;
```

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

Finding Entities Using the `EntityManager`

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

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

Managing an Entity Instance's Lifecycle

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

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

Persisting Entity Instances

New entity instances become managed and persistent either by invoking the `persist` method or by a cascading `persist` operation invoked from related entities that have the `cascade=PERSIST` or `cascade=ALL` elements set in the relationship annotation. This means that the entity's data is stored to the database when the transaction associated with the `persist` operation is completed. If the entity is already managed, the `persist` operation is ignored,

although the `persist` operation will cascade to related entities that have the `cascade` element set to `PERSIST` or `ALL` in the relationship annotation. If `persist` is called on a removed entity instance, the entity becomes managed. If the entity is detached, either `persist` will throw an `IllegalArgumentException`, or the transaction commit will fail.

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

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

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

Removing Entity Instances

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

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

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

Synchronizing Entity Data to the Database

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

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

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

Persistence Units

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

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

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

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

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

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

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

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

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

Querying Entities

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

- The Java Persistence query language (JPQL) is a simple, string-based language similar to SQL used to query entities and their relationships. See Chapter 34, "The Java Persistence Query Language," for more information.
- The Criteria API is used to create typesafe queries using Java programming language APIs to query for entities and their relationships. See Chapter 35, "Using the Criteria API to Create Queries," for more information.

Both JPQL and the Criteria API have advantages and disadvantages.

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

Criteria queries allow you to define the query in the business tier of the application. Although this is also possible using JPQL dynamic queries, Criteria queries provide better performance because JPQL dynamic queries must be parsed each time they are called. Criteria queries are typesafe and therefore don't require casting, as JPQL queries do. The Criteria API is just another Java programming language API and doesn't require developers to learn the syntax of another

query language. Criteria queries are typically more verbose than JPQL queries and require the developer to create several objects and perform operations on those objects before submitting the query to the entity manager.

Further Information about Persistence

For more information about the Java Persistence API, see

- Java Persistence 2.0 API specification:

  `http://jcp.org/en/jsr/detail?id=317`
- EclipseLink, the Java Persistence API implementation in the GlassFish Server:

  `http://www.eclipse.org/eclipselink/jpa.php`
- EclipseLink team blog:

  `http://eclipselink.blogspot.com/`
- EclipseLink wiki documentation:

  `http://wiki.eclipse.org/EclipseLink`

◆ ◆ ◆ CHAPTER 33

Running the Persistence Examples

This chapter explains how to use the Java Persistence API. The material here focuses on the source code and settings of three examples. The first example, `order`, is an application that uses a stateful session bean to manage entities related to an ordering system. The second example, `roster`, is an application that manages a community sports system. The third example, `address-book`, is a web application that stores contact data. This chapter assumes that you are familiar with the concepts detailed in Chapter 32, "Introduction to the Java Persistence API."

The following topics are addressed here:

The order Application

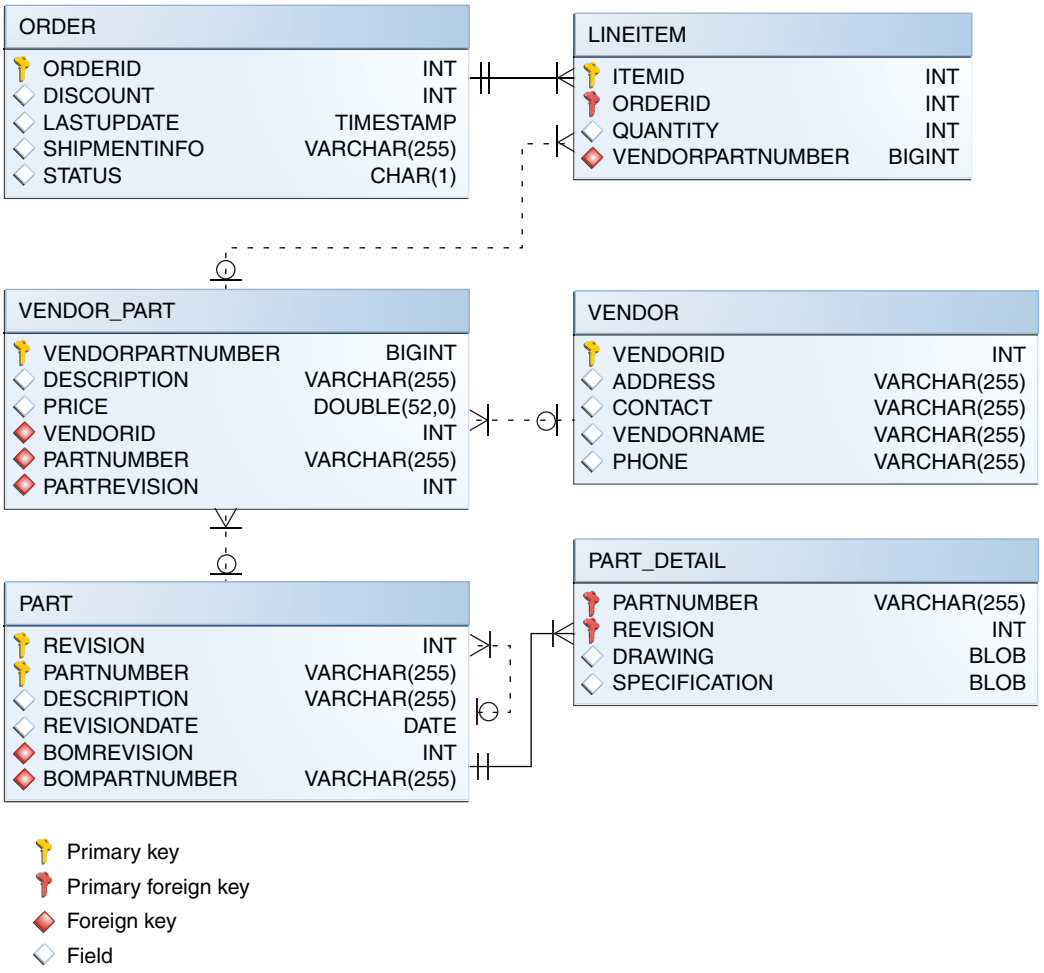
The `order` application is a simple inventory and ordering application for maintaining a catalog of parts and placing an itemized order of those parts. The application has entities that represent parts, vendors, orders, and line items. These entities are accessed using a stateful session bean that holds the business logic of the application. A simple singleton session bean creates the initial entities on application deployment. A Facelets web application manipulates the data and displays data from the catalog.

The information contained in an order can be divided into elements. What is the order number? What parts are included in the order? What parts make up that part? Who makes the part? What are the specifications for the part? Are there any schematics for the part? The `order` application is a simplified version of an ordering system that has all these elements.

The `order` application consists of a single WAR module that includes the enterprise bean classes, the entities, the support classes, and the Facelets XHTML and class files.

The database schema in the Java DB database for `order` is shown in Figure 33–1.

FIGURE 33–1 Database Schema for the order Application



Note – In this diagram, for simplicity, the `PERSISTENCE_ORDER_` prefix is omitted from the table names.

Entity Relationships in the order Application

The `order` application demonstrates several types of entity relationships: self-referential, one-to-one, one-to-many, many-to-one, and unidirectional relationships.

Self-Referential Relationships

A *self-referential* relationship occurs between relationship fields in the same entity. `Part` has a field, `bomPart`, which has a one-to-many relationship with the field `parts`, which is also in `Part`. That is, a part can be made up of many parts, and each of those parts has exactly one bill-of-material part.

The primary key for `Part` is a compound primary key, a combination of the `partNumber` and `revision` fields. This key is mapped to the `PARTNUMBER` and `REVISION` columns in the `EJB_ORDER_PART` table:

```
...
@ManyToOne
@JoinColumns({
    @JoinColumn(name="BOMPARTNUMBER",
        referencedColumnName="PARTNUMBER"),
    @JoinColumn(name="BOMREVISION",
        referencedColumnName="REVISION")
})
public Part getBomPart() {
    return bomPart;
}
...
@OneToMany(mappedBy="bomPart")
public Collection<Part> getParts() {
    return parts;
}
...
```

One-to-One Relationships

`Part` has a field, `vendorPart`, that has a one-to-one relationship with `VendorPart`'s `part` field. That is, each part has exactly one vendor part, and vice versa.

Here is the relationship mapping in `Part`:

```
@OneToOne(mappedBy="part")
public VendorPart getVendorPart() {
    return vendorPart;
}
```

Here is the relationship mapping in `VendorPart`:

```
@OneToOne
@JoinColumns({
    @JoinColumn(name="PARTNUMBER",
        referencedColumnName="PARTNUMBER"),
    @JoinColumn(name="PARTREVISION",
        referencedColumnName="REVISION")
})
public Part getPart() {
    return part;
}
```

Note that, because Part uses a compound primary key, the @JoinColumns annotation is used to map the columns in the PERSISTENCE_ORDER_VENDOR_PART table to the columns in PERSISTENCE_ORDER_PART. The PERSISTENCE_ORDER_VENDOR_PART table's PARTREVISION column refers to PERSISTENCE_ORDER_PART's REVISION column.

One-to-Many Relationship Mapped to Overlapping Primary and Foreign Keys

Order has a field, lineItems, that has a one-to-many relationship with LineItem's field order. That is, each order has one or more line item.

LineItem uses a compound primary key that is made up of the orderId and itemId fields. This compound primary key maps to the ORDERID and ITEMID columns in the PERSISTENCE_ORDER_LINEITEM table. ORDERID is a foreign key to the ORDERID column in the PERSISTENCE_ORDER_ORDER table. This means that the ORDERID column is mapped twice: once as a primary key field, orderId; and again as a relationship field, order.

Here is the relationship mapping in Order:

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

Here is the relationship mapping in LineItem:

```
@ManyToOne
    public Order getOrder() {
    return order;
}
```

Unidirectional Relationships

LineItem has a field, vendorPart, that has a unidirectional many-to-one relationship with VendorPart. That is, there is no field in the target entity in this relationship:

```
@ManyToOne
    public VendorPart getVendorPart() {
    return vendorPart;
}
```

Primary Keys in the order Application

The order application uses several types of primary keys: single-valued primary keys, compound primary keys, and generated primary keys.

Generated Primary Keys

`VendorPart` uses a generated primary key value. That is, the application does not assign primary key values for the entities but instead relies on the persistence provider to generate the primary key values. The `@GeneratedValue` annotation is used to specify that an entity will use a generated primary key.

In `VendorPart`, the following code specifies the settings for generating primary key values:

```
@TableGenerator(
    name="vendorPartGen",
    table="PERSISTENCE_ORDER_SEQUENCE_GENERATOR",
    pkColumnName="GEN_KEY",
    valueColumnName="GEN_VALUE",
    pkColumnValue="VENDOR_PART_ID",
    allocationSize=10)
@Id
@GeneratedValue(strategy=GenerationType.TABLE,
    generator="vendorPartGen")
public Long getVendorPartNumber() {
    return vendorPartNumber;
}
```

The `@TableGenerator` annotation is used in conjunction with `@GeneratedValue`'s `strategy=TABLE` element. That is, the strategy used to generate the primary keys is to use a table in the database. The `@TableGenerator` annotation is used to configure the settings for the generator table. The name element sets the name of the generator, which is `vendorPartGen` in `VendorPart`.

The `EJB_ORDER_SEQUENCE_GENERATOR` table, whose two columns are `GEN_KEY` and `GEN_VALUE`, will store the generated primary key values. This table could be used to generate other entity's primary keys, so the `pkColumnValue` element is set to `VENDOR_PART_ID` to distinguish this entity's generated primary keys from other entity's generated primary keys. The `allocationSize` element specifies the amount to increment when allocating primary key values. In this case, each `VendorPart`'s primary key will increment by 10.

The primary key field `vendorPartNumber` is of type `Long`, as the generated primary key's field must be an integral type.

Compound Primary Keys

A compound primary key is made up of multiple fields and follows the requirements described in "Primary Keys in Entities" on page 585. To use a compound primary key, you must create a wrapper class.

In order, two entities use compound primary keys: Part and LineItem.

- Part uses the PartKey wrapper class. Part's primary key is a combination of the part number and the revision number. PartKey encapsulates this primary key.
- LineItem uses the LineItemKey class. LineItem's primary key is a combination of the order number and the item number. LineItemKey encapsulates this primary key.

This is the LineItemKey compound primary key wrapper class:

```
package order.entity;

public final class LineItemKey implements
             java.io.Serializable {

    private Integer orderId;
    private int itemId;

    public int hashCode() {
        return ((this.getOrderId()==null
                        ?0:this.getOrderId().hashCode())
                 ^ ((int) this.getItemId()));
    }

    public boolean equals(Object otherOb) {
        if (this == otherOb) {
            return true;
        }
        if (!(otherOb instanceof LineItemKey)) {
            return false;
        }
        LineItemKey other = (LineItemKey) otherOb;
        return ((this.getOrderId()==null
                        ?other.orderId==null:this.getOrderId().equals
                (other.orderId)) && (this.getItemId ==
                    other.itemId));
    }

    public String toString() {
        return "" + orderId + "-" + itemId;
    }
}
```

The @IdClass annotation is used to specify the primary key class in the entity class. In LineItem, @IdClass is used as follows:

```
@IdClass(order.entity.LineItemKey.class)
@Entity
...
public class LineItem {
...
}
```

The two fields in LineItem are tagged with the @Id annotation to mark those fields as part of the compound primary key:

```
@Id
public int getItemId() {
    return itemId;
}
...
@Id
@Column(name="ORDERID", nullable=false,
    insertable=false, updatable=false)
public Integer getOrderId() {
    return orderId;
}
```

For `orderId`, you also use the `@Column` annotation to specify the column name in the table and that this column should not be inserted or updated, as it is an overlapping foreign key pointing at the `PERSISTENCE_ORDER_ORDER` table's `ORDERID` column (see "One-to-Many Relationship Mapped to Overlapping Primary and Foreign Keys" on page 606). That is, `orderId` will be set by the `Order` entity.

In `LineItem`'s constructor, the line item number (`LineItem.itemId`) is set using the `Order.getNextId` method:

```
public LineItem(Order order, int quantity, VendorPart
        vendorPart) {
    this.order = order;
    this.itemId = order.getNextId();
    this.orderId = order.getOrderId();
    this.quantity = quantity;
    this.vendorPart = vendorPart;
}
```

`Order.getNextId` counts the number of current line items, adds 1, and returns that number:

```
public int getNextId() {
    return this.lineItems.size() + 1;
}
```

`Part` doesn't require the `@Column` annotation on the two fields that comprise `Part`'s compound primary key, because `Part`'s compound primary key is not an overlapping primary key/foreign key:

```
@IdClass(order.entity.PartKey.class)
@Entity
...
public class Part {
...
    @Id
    public String getPartNumber() {
        return partNumber;
    }
...
    @Id
    public int getRevision() {
        return revision;
    }
```

```
...
}
```

Entity Mapped to More Than One Database Table

`Part`'s fields map to more than one database table: `PERSISTENCE_ORDER_PART` and `PERSISTENCE_ORDER_PART_DETAIL`. The `PERSISTENCE_ORDER_PART_DETAIL` table holds the specification and schematics for the part. The `@SecondaryTable` annotation is used to specify the secondary table.

```
...
@Entity
@Table(name="PERSISTENCE_ORDER_PART")
@SecondaryTable(name="PERSISTENCE_ORDER_PART_DETAIL", pkJoinColumns={
    @PrimaryKeyJoinColumn(name="PARTNUMBER",
        referencedColumnName="PARTNUMBER"),
    @PrimaryKeyJoinColumn(name="REVISION",
        referencedColumnName="REVISION")
})
public class Part {
...
}
```

`PERSISTENCE_ORDER_PART_DETAIL` and `PERSISTENCE_ORDER_PART` share the same primary key values. The `pkJoinColumns` element of `@SecondaryTable` is used to specify that `PERSISTENCE_ORDER_PART_DETAIL`'s primary key columns are foreign keys to `PERSISTENCE_ORDER_PART`. The `@PrimaryKeyJoinColumn` annotation sets the primary key column names and specifies which column in the primary table the column refers to. In this case, the primary key column names for both `PERSISTENCE_ORDER_PART_DETAIL` and `PERSISTENCE_ORDER_PART` are the same: `PARTNUMBER` and `REVISION`, respectively.

Cascade Operations in the order Application

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

In `order`, there are two cascade delete dependencies in the entity relationships. If the `Order` to which a `LineItem` is related is deleted, the `LineItem` also should be deleted. If the `Vendor` to which a `VendorPart` is related is deleted, the `VendorPart` also should be deleted.

You specify the cascade operations for entity relationships by setting the `cascade` element in the inverse (nonowning) side of the relationship. The cascade element is set to `ALL` in the case of `Order.lineItems`. This means that all persistence operations (deletes, updates, and so on) are cascaded from orders to line items.

Here is the relationship mapping in `Order`:

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

Here is the relationship mapping in `LineItem`:

```
@ManyToOne
    public Order getOrder() {
    return order;
}
```

BLOB and CLOB Database Types in the order Application

The `PARTDETAIL` table in the database has a column, `DRAWING`, of type `BLOB`. `BLOB` stands for binary large objects, which are used for storing binary data, such as an image. The `DRAWING` column is mapped to the field `Part.drawing` of type `java.io.Serializable`. The `@Lob` annotation is used to denote that the field is large object.

```
@Column(table="PERSISTENCE_ORDER_PART_DETAIL")
@Lob
public Serializable getDrawing() {
    return drawing;
}
```

`PERSISTENCE_ORDER_PART_DETAIL` also has a column, `SPECIFICATION`, of type `CLOB`. `CLOB` stands for character large objects, which are used to store string data too large to be stored in a `VARCHAR` column. `SPECIFICATION` is mapped to the field `Part.specification` of type `java.lang.String`. The `@Lob` annotation is also used here to denote that the field is a large object.

```
@Column(table="PERSISTENCE_ORDER_PART_DETAIL")
@Lob
public String getSpecification() {
    return specification;
}
```

Both of these fields use the `@Column` annotation and set the `table` element to the secondary table.

Temporal Types in the order Application

The `Order.lastUpdate` persistent property, which is of type `java.util.Date`, is mapped to the `PERSISTENCE_ORDER_ORDER.LASTUPDATE` database field, which is of the SQL type `TIMESTAMP`. To ensure the proper mapping between these types, you must use the `@Temporal` annotation with the proper temporal type specified in `@Temporal`'s element. `@Temporal`'s elements are of type `javax.persistence.TemporalType`. The possible values are

- `DATE`, which maps to `java.sql.Date`
- `TIME`, which maps to `java.sql.Time`
- `TIMESTAMP`, which maps to `java.sql.Timestamp`

Here is the relevant section of `Order`:

```
@Temporal(TIMESTAMP)
public Date getLastUpdate() {
    return lastUpdate;
}
```

Managing the order Application's Entities

The `RequestBean` stateful session bean contains the business logic and manages the entities of `order`. `RequestBean` uses the `@PersistenceContext` annotation to retrieve an entity manager instance, which is used to manage `order`'s entities in `RequestBean`'s business methods:

```
@PersistenceContext
private EntityManager em;
```

This `EntityManager` instance is a container-managed entity manager, so the container takes care of all the transactions involved in the managing `order`'s entities.

Creating Entities

The `RequestBean.createPart` business method creates a new `Part` entity. The `EntityManager.persist` method is used to persist the newly created entity to the database.

```
Part part = new Part(partNumber,
    revision,
    description,
    revisionDate,
    specification,
    drawing);
em.persist(part);
```

The `ConfigBean` singleton session bean is used to initialize the data in `order`. `ConfigBean` is annotated with `@Startup`, which indicates that the EJB container should create `ConfigBean` when `order` is deployed. The `createData` method is annotated with `@PostConstruct` and creates the initial entities used by `order` by calling `RequestBean`'s business methods.

Finding Entities

The `RequestBean.getOrderPrice` business method returns the price of a given order, based on the `orderId`. The `EntityManager.find` method is used to retrieve the entity from the database.

```
Order order = em.find(Order.class, orderId);
```

The first argument of `EntityManager.find` is the entity class, and the second is the primary key.

Setting Entity Relationships

The `RequestBean.createVendorPart` business method creates a `VendorPart` associated with a particular `Vendor`. The `EntityManager.persist` method is used to persist the newly created `VendorPart` entity to the database, and the `VendorPart.setVendor` and `Vendor.setVendorPart` methods are used to associate the `VendorPart` with the `Vendor`.

```
PartKey pkey = new PartKey();
pkey.partNumber = partNumber;
pkey.revision = revision;

Part part = em.find(Part.class, pkey);
VendorPart vendorPart = new VendorPart(description, price,
    part);
em.persist(vendorPart);

Vendor vendor = em.find(Vendor.class, vendorId);
vendor.addVendorPart(vendorPart);
vendorPart.setVendor(vendor);
```

Using Queries

The `RequestBean.adjustOrderDiscount` business method updates the discount applied to all orders. This method uses the `findAllOrders` named query, defined in `Order`:

```
@NamedQuery(
    name="findAllOrders",
    query="SELECT o FROM Order o"
)
```

The `EntityManager.createNamedQuery` method is used to run the query. Because the query returns a `List` of all the orders, the `Query.getResultList` method is used.

```
List orders = em.createNamedQuery(
    "findAllOrders")
    .getResultList();
```

The `RequestBean.getTotalPricePerVendor` business method returns the total price of all the parts for a particular vendor. This method uses a named parameter, `id`, defined in the named query `findTotalVendorPartPricePerVendor` defined in `VendorPart`.

```
@NamedQuery(
    name="findTotalVendorPartPricePerVendor",
    query="SELECT SUM(vp.price) " +
    "FROM VendorPart vp " +
    "WHERE vp.vendor.vendorId = :id"
)
```

When running the query, the `Query.setParameter` method is used to set the named parameter `id` to the value of `vendorId`, the parameter to `RequestBean.getTotalPricePerVendor`:

```
return (Double) em.createNamedQuery(
    "findTotalVendorPartPricePerVendor")
    .setParameter("id", vendorId)
    .getSingleResult();
```

The `Query.getSingleResult` method is used for this query because the query returns a single value.

Removing Entities

The `RequestBean.removeOrder` business method deletes a given order from the database. This method uses the `EntityManager.remove` method to delete the entity from the database.

```
Order order = em.find(Order.class, orderId);
em.remove(order);
```

Running the order Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the `order` application. First, you will create the database tables in the Java DB server.

▼ To Run the order Example Using NetBeans IDE

1. **From the File menu, choose Open Project.**
2. **In the Open Project dialog, navigate to:**

   tut-install`/examples/persistence/`
3. **Select the `order` folder.**
4. **Select the Open as Main Project check box.**
5. **Click Open Project.**
6. **In the Projects tab, right-click the `order` project and select Run.**

   NetBeans IDE opens a web browser to `http://localhost:8080/order/`.

▼ To Run the order Example Using Ant

1. **In a terminal window, go to:**

   tut-install`/examples/persistence/order/`

2. **Type the following command:**

   ```
   ant
   ```

   This runs the `default` task, which compiles the source files and packages the application into a WAR file located at *tut-install*`/examples/persistence/order/dist/order.war`.

3. **To deploy the WAR, make sure that the GlassFish Server is started, then type the following command:**

   ```
   ant deploy
   ```

4. **Open a web browser to `http://localhost:8080/order/` to create and update the order data.**

The all Task

As a convenience, the `all` task will build, package, deploy, and run the application. To do this, type the following command:

```
ant all
```

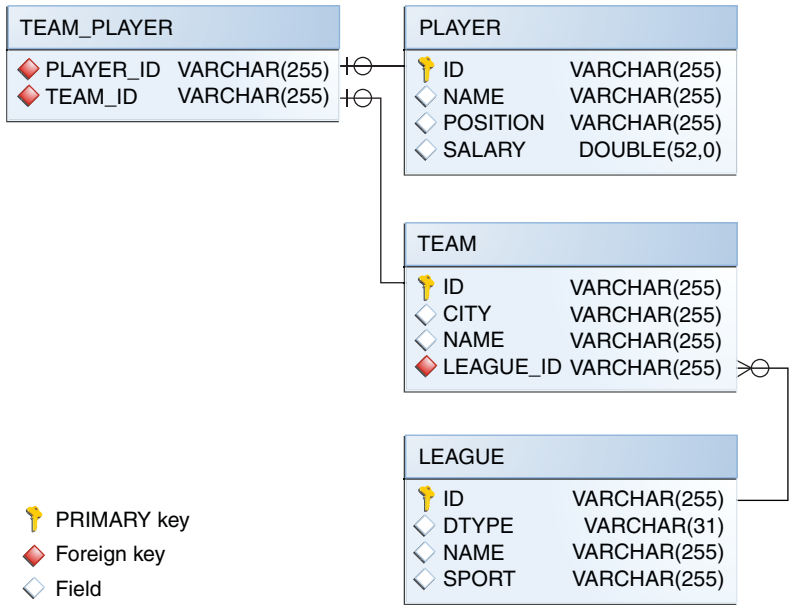
The roster Application

The `roster` application maintains the team rosters for players in recreational sports leagues. The application has four components: Java Persistence API entities (`Player`, `Team`, and `League`), a stateful session bean (`RequestBean`), an application client (`RosterClient`), and three helper classes (`PlayerDetails`, `TeamDetails`, and `LeagueDetails`).

Functionally, `roster` is similar to the `order` application, with three new features that `order` does not have: many-to-many relationships, entity inheritance, and automatic table creation at deployment time.

The database schema in the underlying Java DB database for `roster` is shown in Figure 33–2.

FIGURE 33–2 Database Schema for the roster Application



Note – In this diagram, for simplicity, the `PERSISTENCE_ROSTER_` prefix is omitted from the table names.

Relationships in the roster Application

A recreational sports system has the following relationships:

- A player can be on many teams.
- A team can have many players.
- A team is in exactly one league.
- A league has many teams.

In `roster` this system is reflected by the following relationships between the `Player`, `Team`, and `League` entities.

- There is a many-to-many relationship between `Player` and `Team`.
- There is a many-to-one relationship between `Team` and `League`.

The Many-To-Many Relationship in roster

The many-to-many relationship between `Player` and `Team` is specified by using the `@ManyToMany` annotation. In `Team.java`, the `@ManyToMany` annotation decorates the `getPlayers` method:

```
@ManyToMany
@JoinTable(
    name="EJB_ROSTER_TEAM_PLAYER",
    joinColumns=
        @JoinColumn(name="TEAM_ID", referencedColumnName="ID"),
    inverseJoinColumns=
        @JoinColumn(name="PLAYER_ID", referencedColumnName="ID")
)
public Collection<Player> getPlayers() {
    return players;
}
```

The `@JoinTable` annotation is used to specify a database table that will associate player IDs with team IDs. The entity that specifies the `@JoinTable` is the owner of the relationship, so the `Team` entity is the owner of the relationship with the `Player` entity. Because `roster` uses automatic table creation at deployment time, the container will create a join table named `EJB_ROSTER_TEAM_PLAYER`.

`Player` is the inverse, or nonowning, side of the relationship with `Team`. As one-to-one and many-to-one relationships, the nonowning side is marked by the `mappedBy` element in the relationship annotation. Because the relationship between `Player` and `Team` is bidirectional, the choice of which entity is the owner of the relationship is arbitrary.

In `Player.java`, the `@ManyToMany` annotation decorates the `getTeams` method:

```
@ManyToMany(mappedBy="players")
public Collection<Team> getTeams() {
    return teams;
}
```

Entity Inheritance in the roster Application

The `roster` application shows how to use entity inheritance, as described in "Entity Inheritance" on page 591.

The `League` entity in `roster` is an abstract entity with two concrete subclasses: `SummerLeague` and `WinterLeague`. Because `League` is an abstract class, it cannot be instantiated:

```
...
@Entity
@Table(name = "EJB_ROSTER_LEAGUE")
public abstract class League implements java.io.Serializable {
...
}
```

Instead, when creating a league, clients use `SummerLeague` or `WinterLeague`. `SummerLeague` and `WinterLeague` inherit the persistent properties defined in `League` and add only a constructor that verifies that the sport parameter matches the type of sport allowed in that seasonal league. For example, here is the `SummerLeague` entity:

```
...
@Entity
public class SummerLeague extends League
         implements java.io.Serializable {

    /** Creates a new instance of SummerLeague */
    public SummerLeague() {
    }

    public SummerLeague(String id, String name,
             String sport) throws IncorrectSportException {
        this.id = id;
        this.name = name;
        if (sport.equalsIgnoreCase("swimming") ||
                sport.equalsIgnoreCase("soccer") ||
                sport.equalsIgnoreCase("basketball") ||
                sport.equalsIgnoreCase("baseball")) {
            this.sport = sport;
        } else {
            throw new IncorrectSportException(
                "Sport is not a summer sport.");
        }
    }
}
```

The `roster` application uses the default mapping strategy of `InheritanceType.SINGLE_TABLE`, so the `@Inheritance` annotation is not required. If you want to use a different mapping strategy, decorate `League` with `@Inheritance` and specify the mapping strategy in the `strategy` element:

```
@Entity
@Inheritance(strategy=JOINED)
@Table(name="EJB_ROSTER_LEAGUE")
public abstract class League implements java.io.Serializable {
    ...
}
```

The `roster` application uses the default discriminator column name, so the `@DiscriminatorColumn` annotation is not required. Because you are using automatic table generation in `roster`, the Persistence provider will create a discriminator column called `DTYPE` in the `EJB_ROSTER_LEAGUE` table, which will store the name of the inherited entity used to create the league. If you want to use a different name for the discriminator column, decorate `League` with `@DiscriminatorColumn` and set the `name` element:

```
@Entity
@DiscriminatorColumn(name="DISCRIMINATOR")
@Table(name="EJB_ROSTER_LEAGUE")
public abstract class League implements java.io.Serializable {
    ...
}
```

Criteria Queries in the roster Application

The `roster` application uses Criteria API queries, as opposed to the JPQL queries used in `order`. Criteria queries are Java programming language, typesafe queries defined in the business tier of `roster`, in the `RequestBean` stateful session bean.

Metamodel Classes in the roster Application

Metamodel classes model an entity's attributes and are used by Criteria queries to navigate to an entity's attributes. Each entity class in `roster` has a corresponding metamodel class, generated at compile time, with the same package name as the entity and appended with an underscore character (_). For example, the `roster.entity.Player` entity has a corresponding metamodel class, `roster.entity.Player_`.

Each persistent field or property in the entity class has a corresponding attribute in the entity's metamodel class. For the `Player` entity, the corresponding metamodel class is:

```
@StaticMetamodel(Player.class)
public class Player_ {
  public static volatile SingularAttribute<Player, String> id;
  public static volatile SingularAttribute<Player, String> name;
  public static volatile SingularAttribute<Player, String> position;
  public static volatile SingularAttribute<Player, Double> salary;
  public static volatile CollectionAttribute<Player, Team> teams;
}
```

Obtaining a `CriteriaBuilder` Instance in `RequestBean`

The `CrtiteriaBuilder` interface defines methods to create criteria query objects and create expressions for modifying those query objects. `RequestBean` creates an instance of `CriteriaBuilder` by using a `@PostConstruct` method, `init`:

```
@PersistenceContext
private EntityManager em;
private CriteriaBuilder cb;

@PostConstruct
private void init() {
  cb = em.getCriteriaBuilder();
}
```

The `EntityManager` instance is injected at runtime, and then that `EntityManager` object is used to create the `CriteriaBuilder` instance by calling `getCriteriaBuilder`. The `CriteriaBuilder` instance is created in a `@PostConstruct` method to ensure that the `EntityManager` instance has been injected by the enterprise bean container.

Creating Criteria Queries in `RequestBean`'s Business Methods

Many of the business methods in `RequestBean` define Criteria queries. One business method, `getPlayersByPosition`, returns a list of players who play a particular position on a team:

```
public List<PlayerDetails> getPlayersByPosition(String position) {
    logger.info("getPlayersByPosition");
    List<Player> players = null;

    try {
        CriteriaQuery<Player> cq = cb.createQuery(Player.class);
        if (cq != null) {
            Root<Player> player = cq.from(Player.class);

            // set the where clause
            cq.where(cb.equal(player.get(Player_.position), position));
            cq.select(player);
            TypedQuery<Player> q = em.createQuery(cq);
            players = q.getResultList();
        }

        return copyPlayersToDetails(players);
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}
```

A query object is created by calling the `CriteriaBuilder` object's `createQuery` method, with the type set to `Player` because the query will return a list of players.

The *query root*, the base entity from which the query will navigate to find the entity's attributes and related entities, is created by calling the `from` method of the query object. This sets the FROM clause of the query.

The WHERE clause, set by calling the `where` method on the query object, restricts the results of the query according to the conditions of an expression. The `CriteriaBuilder.equal` method compares the two expressions. In `getPlayersByPosition`, the `position` attribute of the `Player_` metamodel class, accessed by calling the `get` method of the query root, is compared to the `position` parameter passed to `getPlayersByPosition`.

The SELECT clause of the query is set by calling the `select` method of the query object. The query will return `Player` entities, so the query root object is passed as a parameter to `select`.

The query object is prepared for execution by calling `EntityManager.createQuery`, which returns a `TypedQuery<T>` object with the type of the query, in this case `Player`. This typed query object is used to execute the query, which occurs when the `getResultList` method is called, and a `List<Player>` collection is returned.

Automatic Table Generation in the roster Application

At deployment time, the GlassFish Server will automatically drop and create the database tables used by `roster`. This is done by setting the `eclipselink.ddl-generation` property to `drop-and-create-tables` in `persistence.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="em" transaction-type="JTA">
    <jta-data-source>jdbc/__default</jta-data-source>
    <properties>
      <property name="eclipselink.ddl-generation"
                value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

This feature is specific to the Java Persistence API provider used by the GlassFish Server and is not portable across Java EE servers. Automatic table creation is useful for development purposes, however, and the `eclipselink.ddl-generation` property may be removed from `persistence.xml` when preparing the application for production use, when deploying to other Java EE servers, or when using other persistence providers.

Running the roster Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the `roster` application.

▼ To Run the roster Example Using NetBeans IDE

1. **From the File menu, choose Open Project.**
2. **In the Open Project dialog, navigate to:**

   tut-install`/examples/persistence/`
3. **Select the `roster` folder.**
4. **Select the Open as Main Project and Open Required Projects check boxes.**
5. **Click Open Project.**
6. **In the Projects tab, right-click the `roster` project and select Run.**

   You will see the following partial output from the application client in the Output tab:

```
List all players in team T2:
P6 Ian Carlyle goalkeeper 555.0
P7 Rebecca Struthers midfielder 777.0
P8 Anne Anderson forward 65.0
P9 Jan Wesley defender 100.0
P10 Terry Smithson midfielder 100.0

List all teams in league L1:
T1 Honey Bees Visalia
```

```
T2 Gophers Manteca
T5 Crows Orland

List all defenders:
P2 Alice Smith defender 505.0
P5 Barney Bold defender 100.0
P9 Jan Wesley defender 100.0
P22 Janice Walker defender 857.0
P25 Frank Fletcher defender 399.0
...
```

▼ To Run the roster Example Using Ant

1 In a terminal window, go to:

tut-install/examples/persistence/roster/

2 Type the following command:

```
ant
```

This runs the `default` task, which compiles the source files and packages the application into an EAR file located at *tut-install*/examples/persistence/roster/dist/roster.ear.

3 To deploy the EAR, make sure that the GlassFish Server is started; then type the following command:

```
ant deploy
```

The build system will check whether the Java DB database server is running and start it if it is not running, then deploy `roster.ear`. The GlassFish Server will then drop and create the database tables during deployment, as specified in `persistence.xml`.

After `roster.ear` is deployed, a client JAR, `rosterClient.jar`, is retrieved. This contains the application client.

4 To run the application client, type the following command:

```
ant run
```

You will see the output, which begins:

```
[echo] running application client container.
[exec] List all players in team T2:
[exec] P6 Ian Carlyle goalkeeper 555.0
[exec] P7 Rebecca Struthers midfielder 777.0
[exec] P8 Anne Anderson forward 65.0
[exec] P9 Jan Wesley defender 100.0
[exec] P10 Terry Smithson midfielder 100.0

[exec] List all teams in league L1:
[exec] T1 Honey Bees Visalia
[exec] T2 Gophers Manteca
[exec] T5 Crows Orland
```

```
[exec] List all defenders:
[exec] P2 Alice Smith defender 505.0
[exec] P5 Barney Bold defender 100.0
[exec] P9 Jan Wesley defender 100.0
[exec] P22 Janice Walker defender 857.0
[exec] P25 Frank Fletcher defender 399.0
...
```

The all Task

As a convenience, the `all` task will build, package, deploy, and run the application. To do this, type the following command:

```
ant all
```

The address-book Application

The `address-book` example application is a simple web application that stores contact data. It uses a single entity class, `Contact`, that uses the Java API for JavaBeans Validation (Bean Validation) to validate the data stored in the persistent attributes of the entity, as described in "Validating Persistent Fields and Properties" on page 583.

Bean Validation Constraints in address-book

The `Contact` entity uses the `@NotNull`, `@Pattern`, and `@Past` constraints on the persistent attributes.

The `@NotNull` constraint marks the attribute as a required field. The attribute must be set to a non-null value before the entity can be persisted or modified. Bean Validation will throw a validation error if the attribute is null when the entity is persisted or modified.

The `@Pattern` constraint defines a regular expression that the value of the attribute must match before the entity can be persisted or modified. This constraint has two different uses in `address-book`.

- The regular expression declared in the `@Pattern` annotation on the `email` field matches email addresses of the form *name*`@`*domain name*`.`*top level domain*, allowing only valid characters for email addresses. For example, `username@example.com` will pass validation, as will `firstname.lastname@mail.example.com`. However, `firstname,lastname@example.com`, which contains an illegal comma character in the local name, will fail validation.
- The `mobilePhone` and `homePhone` fields are annotated with a `@Pattern` constraint that defines a regular expression to match phone numbers of the form `(`*xxx*`)` *xxx*–*xxxx*.

The `@Past` constraint is applied to the birthday field, which must be a `java.util.Date` in the past.

Here are the relevant parts of the `Contact` entity class:

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

Specifying Error Messages for Constraints in address-book

Some of the constraints in the `Contact` entity specify an optional message:

```
@Pattern(regexp="^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
             message="{invalid.phonenumber}")
    protected String homePhone;
```

The optional message element in the `@Pattern` constraint overrides the default validation message. The message can be specified directly:

```
@Pattern(regexp="^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
             message="Invalid phone number!")
    protected String homePhone;
```

The constraints in `Contact`, however, are strings in the resource bundle *tut-install*`/examples/persistence/address-book/src/java/ValidationMessages.properties`. This allows the validation messages to be located in one single properties file and the messages to be easily localized. Overridden Bean Validation messages must be placed in a resource bundle properties file named `ValidationMessages.properties` in the default package, with localized resource bundles

taking the form `ValidationMessages_`*locale-prefix*`.properties`. For example, `ValidationMessages_es.properties` is the resource bundle used in Spanish speaking locales.

Validating `Contact` Input from a JavaServer Faces Application

The `address-book` application uses a JavaServer Faces web front end to allow users to enter contacts. While JavaServer Faces has a form input validation mechanism using tags in Facelets XHTML files, `address-book` doesn't use these validation tags. Bean Validation constraints in JavaServer Faces managed beans, in this case in the `Contact` entity, automatically trigger validation when the forms are submitted.

The following code snippet from the `Create.xhtml` Facelets file shows some of the input form for creating new `Contact` instances:

```
<h:form>
    <table columns="3" role="presentation">
        <tr>
            <td><h:outputLabel value="#{bundle.CreateContactLabel_firstName}"
                               for="firstName" /></td>
            <td><h:inputText id="firstName"
                             value="#{contactController.selected.firstName}"
                             title="#{bundle.CreateContactTitle_firstName}" /></td>
            <td><h:message for="firstName" /></td>
        </tr>
        <tr>
            <td><h:outputLabel value="#{bundle.CreateContactLabel_lastName}"
                               for="lastName" /></td>
            <td><h:inputText id="lastName"
                             value="#{contactController.selected.lastName}"
                             title="#{bundle.CreateContactTitle_lastName}" /></td>
            <td><h:message for="lastName" /></td>
        </tr>
        ...
    </table>
</h:form>
```

The `<h:inputText>` tags `firstName` and `lastName` are bound to the attributes in the `Contact` entity instance `selected` in the `ContactController` stateless session bean. Each `<h:inputText>` tag has an associated `<h:message>` tag that will display validation error messages. The form doesn't require any JavaServer Faces validation tags, however.

Running the address-book Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the `address-book` application.

▼ To Run the address-book Example Using NetBeans IDE

1. **From the File menu, choose Open Project.**
2. **In the Open Project dialog, navigate to:**
   tut-install`/examples/persistence/`
3. **Select the `address-book` folder.**
4. **Select the Open as Main Project and Open Required Projects check boxes.**
5. **Click Open Project.**
6. **In the Projects tab, right-click the `address-book` project and select Run.**
   After the application has been deployed, a web browser window appears at the following URL:
   `http://localhost:8080/address-book/`
7. **Click Show All Contact Items, then Create New Contact. Type values in the form fields; then click Save.**
   If any of the values entered violate the constraints in `Contact`, an error message will appear in red beside the form field with the incorrect values.

▼ To Run the address-book Example Using Ant

1. **In a terminal window, go to:**
   tut-install`/examples/persistence/address-book/`
2. **Type the following command:**
   ```
   ant
   ```
   This will compile and assemble the `address-book` application.
3. **Type the following command:**
   ```
   ant deploy
   ```
   This will deploy the application to GlassFish Server.
4. **Open a web browser window and type the following URL:**
   ```
   http://localhost:8080/address-book/
   ```

Tip – As a convenience, the `all` task will build, package, deploy, and run the application. To do this, type the following command:

```
ant all
```

5 **Click Show All Contact Items, then Create New Contact. Type values in the form fields; then click Save.**

If any of the values entered violate the constraints in `Contact`, an error message will appear in red beside the form field with the incorrect values.

◆ ◆ ◆ CHAPTER 34

The Java Persistence Query Language

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

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

This chapter relies on the material presented in earlier chapters. For conceptual information, see Chapter 32, "Introduction to the Java Persistence API." For code examples, see Chapter 33, "Running the Persistence Examples."

The following topics are addressed here:

Query Language Terminology

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

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

Creating Queries Using the Java Persistence Query Language

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

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

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

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

```
@NamedQuery(
    name="findAllCustomersWithName",
```

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

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

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

Named Parameters in Queries

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

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

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

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

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

Positional Parameters in Queries

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

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

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

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

Simplified Query Language Syntax

This section briefly describes the syntax of the query language so that you can quickly move on to "Example Queries" on page 633. When you are ready to learn about the syntax in more detail, see "Full Query Language Syntax" on page 637.

Select Statements

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

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

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

Update and Delete Statements

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

```
update_statement :: = update_clause [where_clause]
delete_statement :: = delete_clause [where_clause]
```

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

Example Queries

The following queries are from the `Player` entity of the `roster` application, which is documented in "The `roster` Application" on page 615.

Simple Queries

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

A Basic Select Query

```
SELECT p
FROM Player p
```

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

  ```
  FROM Player AS
   p
  ```

  The `Player` element is the abstract schema name of the `Player` entity.
- **See also**: "Identification Variables" on page 643.

Eliminating Duplicate Values

```
SELECT DISTINCT
 p
FROM Player p
WHERE p.position = ?1
```

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

  The `WHERE` clause restricts the players retrieved by checking their `position`, a persistent field of the `Player` entity. The `?1` element denotes the input parameter of the query.
- **See also**: "Input Parameters" on page 648 and "The `DISTINCT` Keyword" on page 658.

Using Named Parameters

```
SELECT DISTINCT p
FROM Player p
WHERE p.position = :position AND p.name = :name
```

- **Data retrieved**: The players having the specified positions and names.

- **Description**: The `position` and `name` elements are persistent fields of the `Player` entity. The `WHERE` clause compares the values of these fields with the named parameters of the query, set using the `Query.setNamedParameter` method. The query language denotes a named input parameter using a colon (`:`) followed by an identifier. The first input parameter is `:position`, the second is `:name`.

Queries That Navigate to Related Entities

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

A Simple Query with Relationships

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

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

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

  ```
  SELECT DISTINCT p
  FROM Player p JOIN p.teams t
  ```

  This query could also be rewritten as:

  ```
  SELECT DISTINCT p
  FROM Player p
  WHERE p.team IS NOT EMPTY
  ```

Navigating to Single-Valued Relationship Fields

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

```
SELECT t
 FROM Team t JOIN t.league l
 WHERE l.sport = 'soccer' OR l.sport ='football'
```

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

Traversing Relationships with an Input Parameter

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) AS t
WHERE t.city = :city
```

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

  Expressions cannot navigate beyond (or further qualify) relationship fields that are collections. In the syntax of an expression, a collection-valued field is a terminal symbol. Because the `teams` field is a collection, the `WHERE` clause cannot specify `p.teams.city` (an illegal expression).
- **See also**: "Path Expressions" on page 646.

Traversing Multiple Relationships

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league = :league
```

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

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

Navigating According to Related Fields

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league.sport = :sport
```

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

Queries with Other Conditional Expressions

Every `WHERE` clause must specify a conditional expression, of which there are several kinds. In the previous examples, the conditional expressions are comparison expressions that test for equality. The following examples demonstrate some of the other kinds of conditional expressions. For descriptions of all conditional expressions, see "`WHERE` Clause" on page 647.

The LIKE Expression

```
SELECT p
 FROM Player p
 WHERE p.name LIKE 'Mich%'
```

- **Data retrieved**: All players whose names begin with "Mich."
- **Description**: The `LIKE` expression uses wildcard characters to search for strings that match the wildcard pattern. In this case, the query uses the `LIKE` expression and the `%` wildcard to find all players whose names begin with the string "Mich." For example, "Michael" and "Michelle" both match the wildcard pattern.
- **See also**: "`LIKE` Expressions" on page 650.

The IS NULL Expression

```
SELECT t
 FROM Team t
 WHERE t.league IS NULL
```

- **Data retrieved**: All teams not associated with a league.
- **Description**: The `IS NULL` expression can be used to check whether a relationship has been set between two entities. In this case, the query checks whether the teams are associated with any leagues and returns the teams that do not have a league.
- **See also**: "`NULL` Comparison Expressions" on page 650 and "NULL Values" on page 655.

The IS EMPTY Expression

```
SELECT p
FROM Player p
WHERE p.teams IS EMPTY
```

- **Data retrieved**: All players who do not belong to a team.
- **Description**: The `teams` relationship field of the `Player` entity is a collection. If a player does not belong to a team, the `teams` collection is empty, and the conditional expression is `TRUE`.
- **See also**: "Empty Collection Comparison Expressions" on page 651.

The BETWEEN Expression

```
SELECT DISTINCT p
FROM Player p
WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
```

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

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

- **See also**: "`BETWEEN` Expressions" on page 649.

Comparison Operators

```
SELECT DISTINCT p1
FROM Player p1, Player p2
WHERE p1.salary > p2.salary AND p2.name = :name
```

- **Data retrieved**: All players whose salaries are higher than the salary of the player with the specified name.
- **Description**: The `FROM` clause declares two identification variables (`p1` and `p2`) of the same type (`Player`). Two identification variables are needed because the `WHERE` clause compares the salary of one player (`p2`) with that of the other players (`p1`).
- **See also**: "Identification Variables" on page 643.

Bulk Updates and Deletes

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

Update Queries

```
UPDATE Player p
SET p.status = 'inactive'
WHERE p.lastPlayed < :inactiveThresholdDate
```

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

Delete Queries

```
DELETE
FROM Player p
WHERE p.status = 'inactive'
AND p.teams IS EMPTY
```

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

Full Query Language Syntax

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

BNF Symbols

Table 34–1 describes the BNF symbols used in this chapter.

TABLE 34–1 BNF Symbol Summary

Symbol	Description
`::=`	The element to the left of the symbol is defined by the constructs on the right.
*	The preceding construct may occur zero or more times.
`{...}`	The constructs within the braces are grouped together.
`[...]`	The constructs within the brackets are optional.
|	An exclusive `OR`.
`BOLDFACE`	A keyword; although capitalized in the BNF diagram, keywords are not case-sensitive.
White space	A whitespace character can be a space, a horizontal tab, or a line feed.

BNF Grammar of the Java Persistence Query Language

Here is the entire BNF diagram for the 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 ::= abstract_schema_name [AS]
        identification_variable
join ::= join_spec join_association_path_expression [AS]
        identification_variable
fetch_join ::= join_specFETCH join_association_path_expression
association_path_expression ::=
        collection_valued_path_expression |
        single_valued_association_path_expression
join_spec::= [LEFT [OUTER] |INNER] JOIN
join_association_path_expression ::=
        join_collection_valued_path_expression |
        join_single_valued_association_path_expression
join_collection_valued_path_expression::=
    identification_variable.collection_valued_association_field
join_single_valued_association_path_expression::=
        identification_variable.single_valued_association_field
collection_member_declaration ::=
        IN (collection_valued_path_expression) [AS]
        identification_variable
single_valued_path_expression ::=
```

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

```
    collection_member_declaration
simple_select_clause ::= SELECT [DISTINCT]
    simple_select_expression
simple_select_expression::=
    single_valued_path_expression |
    aggregate_expression |
    identification_variable
conditional_expression ::= conditional_term |
    conditional_expression OR conditional_term
conditional_term ::= conditional_factor | conditional_term AND
    conditional_factor
conditional_factor ::= [NOT] conditional_primary
conditional_primary ::= simple_cond_expression |(
    conditional_expression)
simple_cond_expression ::=
    comparison_expression |
    between_expression |
    like_expression |
    in_expression |
    null_comparison_expression |
    empty_collection_comparison_expression |
    collection_member_expression |
    exists_expression
between_expression ::=
    arithmetic_expression [NOT] BETWEEN
        arithmetic_expressionAND arithmetic_expression |
    string_expression [NOT] BETWEEN string_expression AND
        string_expression |
    datetime_expression [NOT] BETWEEN
        datetime_expression AND datetime_expression
in_expression ::=
    state_field_path_expression [NOT] IN (in_item {, in_item}*
    | subquery)
in_item ::= literal | input_parameter
like_expression ::=
    string_expression [NOT] LIKE pattern_value [ESCAPE
        escape_character]
null_comparison_expression ::=
    {single_valued_path_expression | input_parameter} IS [NOT]
        NULL
empty_collection_comparison_expression ::=
    collection_valued_path_expression IS [NOT] EMPTY
collection_member_expression ::= entity_expression
    [NOT] MEMBER [OF] collection_valued_path_expression
exists_expression::= [NOT] EXISTS (subquery)
all_or_any_expression ::= {ALL |ANY |SOME} (subquery)
comparison_expression ::=
    string_expression comparison_operator {string_expression |
    all_or_any_expression} |
    boolean_expression {= |<> } {boolean_expression |
    all_or_any_expression} |
    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}
```

```
comparison_operator ::= = |> |>= |< |<= |<>
arithmetic_expression ::= simple_arithmetic_expression |
    (subquery)
simple_arithmetic_expression ::=
    arithmetic_term | simple_arithmetic_expression {+ |- }
        arithmetic_term
arithmetic_term ::= arithmetic_factor | arithmetic_term {* |/ }
    arithmetic_factor
arithmetic_factor ::= [{+ |- }] arithmetic_primary
arithmetic_primary ::=
    state_field_path_expression |
    numeric_literal |
    (simple_arithmetic_expression) |
    input_parameter |
    functions_returning_numerics |
    aggregate_expression
string_expression ::= string_primary | (subquery)
string_primary ::=
    state_field_path_expression |
    string_literal |
    input_parameter |
    functions_returning_strings |
    aggregate_expression
datetime_expression ::= datetime_primary | (subquery)
datetime_primary ::=
    state_field_path_expression |
    input_parameter |
    functions_returning_datetime |
    aggregate_expression
boolean_expression ::= boolean_primary | (subquery)
boolean_primary ::=
    state_field_path_expression |
    boolean_literal |
    input_parameter
 enum_expression ::= enum_primary | (subquery)
enum_primary ::=
    state_field_path_expression |
    enum_literal |
    input_parameter
entity_expression ::=
    single_valued_association_path_expression |
        simple_entity_expression
simple_entity_expression ::=
    identification_variable |
    input_parameter
functions_returning_numerics::=
    LENGTH(string_primary) |
    LOCATE(string_primary, string_primary[,
        simple_arithmetic_expression]) |
    ABS(simple_arithmetic_expression) |
    SQRT(simple_arithmetic_expression) |
    MOD(simple_arithmetic_expression,
        simple_arithmetic_expression) |
    SIZE(collection_valued_path_expression)
functions_returning_datetime ::=
    CURRENT_DATE |
    CURRENT_TIME |
    CURRENT_TIMESTAMP
functions_returning_strings ::=
```

```
    CONCAT(string_primary, string_primary) |
    SUBSTRING(string_primary,
        simple_arithmetic_expression,
        simple_arithmetic_expression)|
    TRIM([[trim_specification] [trim_character] FROM]
        string_primary) |
    LOWER(string_primary) |
    UPPER(string_primary)
trim_specification ::= LEADING | TRAILING | BOTH
```

FROM Clause

The `FROM` clause defines the domain of the query by declaring identification variables.

Identifiers

An identifier is a sequence of one or more characters. The first character must be a valid first character (letter, $, _) in an identifier of the Java programming language, hereafter in this chapter called simply "Java". Each subsequent character in the sequence must be a valid nonfirst character (letter, digit, $, _) in a Java identifier. (For details, see the Java SE API documentation of the `isJavaIdentifierStart` and `isJavaIdentifierPart` methods of the `Character` class.) The question mark (?) is a reserved character in the query language and cannot be used in an identifier.

A query language identifier is case-sensitive, with two exceptions:

- Keywords
- Identification variables

An identifier cannot be the same as a query language keyword. Here is a list of query language keywords:

ABS	ALL	AND	ANY
AS	ASC	AVG	BETWEEN
BIT_LENGTH	BOTH	BY	CASE
CHAR_LENGTH	CHARACTER_LENGTH	CLASS	COALESCE
CONCAT	COUNT	CURRENT_DATE	CURRENT_TIMESTAMP
DELETE	DESC	DISTINCT	ELSE
EMPTY	END	ENTRY	ESCAPE
EXISTS	FALSE	FETCH	FROM
GROUP	HAVING	IN	INDEX
INNER	IS	JOIN	KEY

LEADING	LEFT	LENGTH	LIKE
LOCATE	LOWER	MAX	MEMBER
MIN	MOD	NEW	NOT
NULL	NULLIF	OBJECT	OF
OR	ORDER	OUTER	POSITION
SELECT	SET	SIZE	SOME
SQRT	SUBSTRING	SUM	THEN
TRAILING	TRIM	TRUE	TYPE
UNKNOWN	UPDATE	UPPER	VALUE
WHEN	WHERE		

It is not recommended that you use an SQL keyword as an identifier, because the list of keywords may expand to include other reserved SQL words in the future.

Identification Variables

An *identification variable* is an identifier declared in the `FROM` clause. Although they can reference identification variables, the `SELECT` and `WHERE` clauses cannot declare them. All identification variables must be declared in the `FROM` clause.

Because it is an identifier, an identification variable has the same naming conventions and restrictions as an identifier, with the exception that an identification variables is case-insensitive. For example, an identification variable cannot be the same as a query language keyword. (See the preceding section for more naming rules.) Also, within a given persistence unit, an identification variable name must not match the name of any entity or abstract schema.

The `FROM` clause can contain multiple declarations, separated by commas. A declaration can reference another identification variable that has been previously declared (to the left). In the following `FROM` clause, the variable `t` references the previously declared variable `p`:

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

Even if it is not used in the `WHERE` clause, an identification variable's declaration can affect the results of the query. For example, compare the next two queries. The following query returns all players, whether or not they belong to a team:

```
SELECT p
FROM Player p
```

In contrast, because it declares the `t` identification variable, the next query fetches all players who belong to a team:

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

The following query returns the same results as the preceding query, but the `WHERE` clause makes it easier to read:

```
SELECT p
FROM Player p
WHERE p.teams IS NOT EMPTY
```

An identification variable always designates a reference to a single value whose type is that of the expression used in the declaration. There are two kinds of declarations: range variable and collection member.

Range Variable Declarations

To declare an identification variable as an abstract schema type, you specify a range variable declaration. In other words, an identification variable can range over the abstract schema type of an entity. In the following example, an identification variable named `p` represents the abstract schema named `Player`:

```
FROM Player p
```

A range variable declaration can include the optional `AS` operator:

```
FROM Player AS p
```

To obtain objects, a query usually uses path expressions to navigate through the relationships. But for those objects that cannot be obtained by navigation, you can use a range variable declaration to designate a starting point, or *root*.

If the query compares multiple values of the same abstract schema type, the `FROM` clause must declare multiple identification variables for the abstract schema:

```
FROM Player p1, Player p2
```

For an example of such a query, see "Comparison Operators" on page 637.

Collection Member Declarations

In a one-to-many relationship, the multiple side consists of a collection of entities. An identification variable can represent a member of this collection. To access a collection member, the path expression in the variable's declaration navigates through the relationships in the abstract schema. (For more information on path expressions, see "Path Expressions" on page 646.) Because a path expression can be based on another path expression, the navigation can traverse several relationships. See "Traversing Multiple Relationships" on page 635.

A collection member declaration must include the `IN` operator but can omit the optional `AS` operator.

In the following example, the entity represented by the abstract schema named `Player` has a relationship field called `teams`. The identification variable called `t` represents a single member of the `teams` collection.

```
FROM Player p, IN (p.tea
ms) t
```

Joins

The `JOIN` operator is used to traverse over relationships between entities and is functionally similar to the `IN` operator.

In the following example, the query joins over the relationship between customers and orders:

```
SELECT c
 FROM Customer c JOIN c.orders o
 WHERE c.status = 1 AND o.totalPrice > 10000
```

The `INNER` keyword is optional:

```
SELECT c
 FROM Customer c INNER JOIN c.orders o
 WHERE c.status = 1 AND o.totalPrice > 10000
```

These examples are equivalent to the following query, which uses the `IN` operator:

```
SELECT c
 FROM Customer c, IN(c.orders) o
 WHERE c.status = 1 AND o.totalPrice > 10000
```

You can also join a single-valued relationship:

```
SELECT t
 FROM Team t JOIN t.league l
 WHERE l.sport = :sport
```

A `LEFT JOIN` or `LEFT OUTER JOIN` retrieves a set of entities where matching values in the join condition may be absent. The `OUTER` keyword is optional.

```
SELECT c.name, o.totalPrice
FROM Order o LEFT JOIN o.customer c
```

A `FETCH JOIN` is a join operation that returns associated entities as a side effect of running the query. In the following example, the query returns a set of departments and, as a side effect, the associated employees of the departments, even though the employees were not explicitly retrieved by the `SELECT` clause.

```
SELECT d
FROM Department d LEFT JOIN FETCH d.employees
WHERE d.deptno = 1
```

Path Expressions

Path expressions are important constructs in the syntax of the query language, for several reasons. First, path expressions define navigation paths through the relationships in the abstract schema. These path definitions affect both the scope and the results of a query. Second, path expressions can appear in any of the main clauses of a query (`SELECT`, `DELETE`, `HAVING`, `UPDATE`, `WHERE`, `FROM`, `GROUP BY`, `ORDER BY`). Finally, although much of the query language is a subset of SQL, path expressions are extensions not found in SQL.

Examples of Path Expressions

Here, the `WHERE` clause contains a `single_valued_path_expression`; the `p` is an identification variable, and `salary` is a persistent field of `Player`:

```
SELECT DISTINCT p
FROM Player p
 WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
```

Here, the `WHERE` clause also contains a `single_valued_path_expression`; `t` is an identification variable, `league` is a single-valued relationship field, and `sport` is a persistent field of `league`:

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
 WHERE t.league.sport = :sport
```

Here, the `WHERE` clause contains a `collection_valued_path_expression`; `p` is an identification variable, and `teams` designates a collection-valued relationship field:

```
SELECT DISTINCT p
FROM Player p
 WHERE p.teams IS EMPTY
```

Expression Types

The type of a path expression is the type of the object represented by the ending element, which can be one of the following:

- Persistent field
- Single-valued relationship field
- Collection-valued relationship field

For example, the type of the expression `p.salary` is `double` because the terminating persistent field (`salary`) is a `double`.

In the expression `p.teams`, the terminating element is a collection-valued relationship field (`teams`). This expression's type is a collection of the abstract schema type named `Team`. Because `Team` is the abstract schema name for the `Team` entity, this type maps to the entity. For more information on the type mapping of abstract schemas, see "Return Types" on page 657.

Navigation

A path expression enables the query to navigate to related entities. The terminating elements of an expression determine whether navigation is allowed. If an expression contains a single-valued relationship field, the navigation can continue to an object that is related to the field. However, an expression cannot navigate beyond a persistent field or a collection-valued relationship field. For example, the expression `p.teams.league.sport` is illegal because `teams` is a collection-valued relationship field. To reach the `sport` field, the `FROM` clause could define an identification variable named `t` for the `teams` field:

```
FROM Player AS p, IN (p.teams) t
 WHERE t.league.sport = 'soccer'
```

WHERE Clause

The `WHERE` clause specifies a conditional expression that limits the values returned by the query. The query returns all corresponding values in the data store for which the conditional expression is `TRUE`. Although usually specified, the `WHERE` clause is optional. If the `WHERE` clause is omitted, the query returns all values. The high-level syntax for the `WHERE` clause follows:

```
where_clause ::= WHERE conditional_expression
```

Literals

There are four kinds of literals: string, numeric, Boolean, and enum.

- **String literals**: A string literal is enclosed in single quotes:

  ```
  'Duke'
  ```

  If a string literal contains a single quote, you indicate the quote by using two single quotes:

  ```
  'Duke''s'
  ```

  Like a Java `String`, a string literal in the query language uses the Unicode character encoding.

- **Numeric literals**: There are two types of numeric literals: exact and approximate.

  An exact numeric literal is a numeric value without a decimal point, such as 65, –233, and +12. Using the Java integer syntax, exact numeric literals support numbers in the range of a Java `long`.

  An approximate numeric literal is a numeric value in scientific notation, such as 57., –85.7, and +2.1. Using the syntax of the Java floating-point literal, approximate numeric literals support numbers in the range of a Java `double`.

- **Boolean literals**: A Boolean literal is either `TRUE` or `FALSE`. These keywords are not case-sensitive.

- **Enum literals**: The Java Persistence query language supports the use of enum literals using the Java enum literal syntax. The enum class name must be specified as a fully qualified class name:

```
SELECT e
 FROM Employee e
 WHERE e.status = com.xyz.EmployeeStatus.FULL_TIME
```

Input Parameters

An input parameter can be either a named parameter or a positional parameter.

- A named input parameter is designated by a colon (`:`) followed by a string; for example, `:name`.
- A positional input parameter is designated by a question mark (`?`) followed by an integer. For example, the first input parameter is `?1`, the second is `?2`, and so forth.

The following rules apply to input parameters.

- They can be used only in a `WHERE` or `HAVING` clause.
- Positional parameters must be numbered, starting with the integer 1.
- Named parameters and positional parameters may not be mixed in a single query.
- Named parameters are case-sensitive.

Conditional Expressions

A `WHERE` clause consists of a conditional expression, which is evaluated from left to right within a precedence level. You can change the order of evaluation by using parentheses.

Operators and Their Precedence

Table 34–2 lists the query language operators in order of decreasing precedence.

TABLE 34–2 Query Language Order Precedence

Type	Precedence Order
Navigation	`.` (a period)
Arithmetic	`+` `–` (unary)
	`*` `/` (multiplication and division)
	`+` `–` (addition and subtraction)

TABLE 34–2 Query Language Order Precedence *(Continued)*

Type	Precedence Order
Comparison	=
	>
	>=
	<
	<=
	<> (not equal)
	[NOT] BETWEEN
	[NOT] LIKE
	[NOT] IN
	IS [NOT] NULL
	IS [NOT] EMPTY
	[NOT] MEMBER OF
Logical	NOT
	AND
	OR

BETWEEN Expressions

A `BETWEEN` expression determines whether an arithmetic expression falls within a range of values.

These two expressions are equivalent:

```
p.age BETWEEN 15 AND 19
 p.age >= 15 AND p.age <= 19
```

The following two expressions also are equivalent:

```
p.age NOT BETWEEN 15 AND 19
 p.age < 15 OR p.age > 19
```

If an arithmetic expression has a `NULL` value, the value of the `BETWEEN` expression is unknown.

IN Expressions

An `IN` expression determines whether a string belongs to a set of string literals or whether a number belongs to a set of number values.

The path expression must have a string or numeric value. If the path expression has a NULL value, the value of the IN expression is unknown.

In the following example, the expression is TRUE if the country is UK , but FALSE if the country is Peru.

```
o.country IN ('UK', 'US', 'France')
```

You may also use input parameters:

```
o.country IN ('UK', 'US', 'France', :country)
```

LIKE Expressions

A LIKE expression determines whether a wildcard pattern matches a string.

The path expression must have a string or numeric value. If this value is NULL, the value of the LIKE expression is unknown. The pattern value is a string literal that can contain wildcard characters. The underscore (_) wildcard character represents any single character. The percent (%) wildcard character represents zero or more characters. The ESCAPE clause specifies an escape character for the wildcard characters in the pattern value. Table 34–3 shows some sample LIKE expressions.

TABLE 34–3 LIKE Expression Examples

Expression	TRUE	FALSE
address.phone LIKE '12%3'	'123'	'1234'
	'12993'	
asentence.word LIKE 'l_se'	'lose'	'loose'
aword.underscored LIKE '_%' ESCAPE '\'	'_foo'	'bar'
address.phone NOT LIKE '12%3'	'1234'	'123'
		'12993'

NULL Comparison Expressions

A NULL comparison expression tests whether a single-valued path expression or an input parameter has a NULL value. Usually, the NULL comparison expression is used to test whether a single-valued relationship has been set:

```
SELECT t
 FROM Team t
 WHERE t.league IS NULL
```

This query selects all teams where the league relationship is not set. Note that the following query is *not* equivalent:

```
SELECT t
 FROM Team t
 WHERE t.league = NULL
```

The comparison with `NULL` using the equals operator (=) always returns an unknown value, even if the relationship is not set. The second query will always return an empty result.

Empty Collection Comparison Expressions

The `IS [NOT] EMPTY` comparison expression tests whether a collection-valued path expression has no elements. In other words, it tests whether a collection-valued relationship has been set.

If the collection-valued path expression is `NULL`, the empty collection comparison expression has a `NULL` value.

Here is an example that finds all orders that do not have any line items:

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

Collection Member Expressions

The `[NOT] MEMBER [OF]` collection member expression determines whether a value is a member of a collection. The value and the collection members must have the same type.

If either the collection-valued or single-valued path expression is unknown, the collection member expression is unknown. If the collection-valued path expression designates an empty collection, the collection member expression is `FALSE`.

The `OF` keyword is optional.

The following example tests whether a line item is part of an order:

```
SELECT o
 FROM Order o
 WHERE :lineItem MEMBER OF o.lineItems
```

Subqueries

Subqueries may be used in the `WHERE` or `HAVING` clause of a query. Subqueries must be surrounded by parentheses.

The following example finds all customers who have placed more than ten orders:

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

Subqueries may contain `EXISTS`, `ALL`, and `ANY` expressions.

- **`EXISTS` expressions**: The `[NOT] EXISTS` expression is used with a subquery and is true only if the result of the subquery consists of one or more values and is false otherwise.

  The following example finds all employees whose spouses are also employees:

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

- **`ALL` and `ANY` expressions**: The `ALL` expression is used with a subquery and is true if all the values returned by the subquery are true or if the subquery is empty.

  The `ANY` expression is used with a subquery and is true if some of the values returned by the subquery are true. An `ANY` expression is false if the subquery result is empty or if all the values returned are false. The `SOME` keyword is synonymous with `ANY`.

  The `ALL` and `ANY` expressions are used with the =, <, <=, >, >=, and <> comparison operators.

  The following example finds all employees whose salaries are higher than the salaries of the managers in the employee's department:

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

Functional Expressions

The query language includes several string, arithmetic, and date/time functions that may be used in the `SELECT`, `WHERE`, or `HAVING` clause of a query. The functions are listed in Table 34–4, Table 34–5, and Table 34–6.

In Table 34–4, the `start` and `length` arguments are of type `int` and designate positions in the `String` argument. The first position in a string is designated by 1.

TABLE 34–4 String Expressions

Function Syntax	Return Type
`CONCAT(String, String)`	`String`
`LENGTH(String)`	`int`
`LOCATE(String, String [, start])`	`int`
`SUBSTRING(String, start, length)`	`String`
`TRIM([[LEADING|TRAILING|BOTH] char) FROM] (String)`	`String`
`LOWER(String)`	`String`
`UPPER(String)`	`String`

The `CONCAT` function concatenates two strings into one string.

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

The `LOCATE` function returns the position of a given string within a string. This function 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 starting string position. By default, `LOCATE` starts at the beginning of the string. The starting position of a string is `1`. If the string cannot be located, `LOCATE` returns `0`.

The `SUBSTRING` function returns a string that is a substring of the first argument based on the starting position and length.

The `TRIM` function trims the specified character from the beginning and/or end of a string. If no character is specified, `TRIM` removes spaces or blanks from the string. If the optional `LEADING` specification is used, `TRIM` removes only the leading characters from the string. If the optional `TRAILING` specification is used, `TRIM` removes only the trailing characters from the string. The default is `BOTH`, which removes the leading and trailing characters from the string.

The `LOWER` and `UPPER` functions convert a string to lowercase or uppercase, respectively.

In Table 34–5, the `number` argument can be an `int`, a `float`, or a `double`.

TABLE 34–5 Arithmetic Expressions

Function Syntax	Return Type
`ABS(number)`	`int`, `float`, or `double`
`MOD(int, int)`	`int`
`SQRT(double)`	`double`
`SIZE(Collection)`	`int`

The `ABS` function takes a numeric expression and returns a number of the same type as the argument.

The `MOD` function returns the remainder of the first argument divided by the second.

The `SQRT` function returns the square root of a number.

The `SIZE` function returns an integer of the number of elements in the given collection.

In Table 34–6, the date/time functions return the date, time, or timestamp on the database server.

TABLE 34–6 Date/Time Expressions

Function Syntax	Return Type
`CURRENT_DATE`	`java.sql.Date`
`CURRENT_TIME`	`java.sql.Time`
`CURRENT_TIMESTAMP`	`java.sql.Timestamp`

Case Expressions

Case expressions change based on a condition, similar to the `case` keyword of the Java programming language. The `CASE` keyword indicates the start of a case expression, and the expression is terminated by the `END` keyword. The `WHEN` and `THEN` keywords define individual conditions, and the `ELSE` keyword defines the default condition should none of the other conditions be satisfied.

The following query selects the name of a person and a conditional string, depending on the subtype of the `Person` entity. If the subtype is `Student`, the string `kid` is returned . If the subtype is `Guardian` or `Staff`, the string `adult` is returned. If the entity is some other subtype of `Person`, the string `unknown` is returned.

```
SELECT p.name
CASE TYPE(p)
  WHEN Student THEN 'kid'
```

```
  WHEN Guardian THEN 'adult'
  WHEN Staff THEN 'adult'
  ELSE 'unknown'
END
FROM Person p
```

The following query sets a discount for various types of customers. Gold-level customers get a 20% discount, silver-level customers get a 15% discount, bronze-level customers get a 10% discount, and everyone else gets a 5% discount.

```
UPDATE Customer c
SET c.discount =
  CASE c.level
    WHEN 'Gold' THEN 20
    WHEN 'SILVER' THEN 15
    WHEN 'Bronze' THEN 10
    ELSE 5
  END
```

NULL Values

If the target of a reference is not in the persistent store, the target is `NULL`. For conditional expressions containing `NULL`, the query language uses the semantics defined by SQL92. Briefly, these semantics are as follows.

- If a comparison or arithmetic operation has an unknown value, it yields a `NULL` value.
- Two `NULL` values are not equal. Comparing two `NULL` values yields an unknown value.
- The `IS NULL` test converts a `NULL` persistent field or a single-valued relationship field to `TRUE`. The `IS NOT NULL` test converts them to `FALSE`.
- Boolean operators and conditional tests use the three-valued logic defined by Table 34–7 and Table 34–8. (In these tables, T stands for `TRUE`, F for `FALSE`, and U for unknown.)

TABLE 34–7 AND Operator Logic

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

TABLE 34–8 OR Operator Logic

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

Equality Semantics

In the query language, only values of the same type can be compared. However, this rule has one exception: Exact and approximate numeric values can be compared. In such a comparison, the required type conversion adheres to the rules of Java numeric promotion.

The query language treats compared values as if they were Java types and not as if they represented types in the underlying data store. For example, a persistent field that could be either an integer or a `NULL` must be designated as an `Integer` object and not as an `int` primitive. This designation is required because a Java object can be `NULL`, but a primitive cannot.

Two strings are equal only if they contain the same sequence of characters. Trailing blanks are significant; for example, the strings `'abc'` and `'abc '` are not equal.

Two entities of the same abstract schema type are equal only if their primary keys have the same value. Table 34–9 shows the operator logic of a negation, and Table 34–10 shows the truth values of conditional tests.

TABLE 34–9 NOT Operator Logic

NOT Value	Value
T	F
F	T
U	U

TABLE 34–10 Conditional Test

Conditional Test	T	F	U
Expression `IS TRUE`	T	F	F
Expression `IS FALSE`	F	T	F
Expression is unknown	F	F	T

SELECT Clause

The SELECT clause defines the types of the objects or values returned by the query.

Return Types

The return type of the SELECT clause is defined by the result types of the select expressions contained within it. If multiple expressions are used, the result of the query is an Object[], and the elements in the array correspond to the order of the expressions in the SELECT clause and in type to the result types of each expression.

A SELECT clause cannot specify a collection-valued expression. For example, the SELECT clause p.teams is invalid because teams is a collection. However, the clause in the following query is valid because the t is a single element of the teams collection:

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

The following query is an example of a query with multiple expressions in the SELECT clause:

```
SELECT c.name, c.country.name
 FROM customer c
 WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

This query returns a list of Object[] elements; the first array element is a string denoting the customer name, and the second array element is a string denoting the name of the customer's country.

The result of a query may be the result of an aggregate function, listed in Table 34–11.

TABLE 34–11 Aggregate Functions in Select Statements

Name	Return Type	Description
AVG	Double	Returns the mean average of the fields
COUNT	Long	Returns the total number of results
MAX	The type of the field	Returns the highest value in the result set
MIN	The type of the field	Returns the lowest value in the result set
SUM	Long (for integral fields) Double (for floating-point fields) BigInteger (for BigInteger fields) BigDecimal (for BigDecimal fields)	Returns the sum of all the values in the result set

For select method queries with an aggregate function (`AVG`, `COUNT`, `MAX`, `MIN`, or `SUM`) in the `SELECT` clause, the following rules apply:

- The `AVG`, `MAX`, `MIN`, and `SUM` functions return `null` if there are no values to which the function can be applied.
- The `COUNT` function returns 0 if there are no values to which the function can be applied.

The following example returns the average order quantity:

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

The following example returns the total cost of the items ordered by Roxane Coss:

```
SELECT SUM(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

The following example returns the total number of orders:

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

The following example returns the total number of items that have prices in Hal Incandenza's order:

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

The DISTINCT Keyword

The `DISTINCT` keyword eliminates duplicate return values. If a query returns a `java.util.Collection`, which allows duplicates, you must specify the `DISTINCT` keyword to eliminate duplicates.

Constructor Expressions

Constructor expressions allow you to return Java instances that store a query result element instead of an `Object[]`.

The following query creates a `CustomerDetail` instance per `Customer` matching the `WHERE` clause. A `CustomerDetail` stores the customer name and customer's country name. So the query returns a `List` of `CustomerDetail` instances:

```
SELECT NEW com.xyz.CustomerDetail(c.name, c.country.name)
 FROM customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

ORDER BY Clause

As its name suggests, the ORDER BY clause orders the values or objects returned by the query.

If the ORDER BY clause contains multiple elements, the left-to-right sequence of the elements determines the high-to-low precedence.

The ASC keyword specifies ascending order, the default, and the DESC keyword indicates descending order.

When using the ORDER BY clause, the SELECT clause must return an orderable set of objects or values. You cannot order the values or objects for values or objects not returned by the SELECT clause. For example, the following query is valid because the ORDER BY clause uses the objects returned by the SELECT clause:

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

The following example is *not* valid, because the ORDER BY clause uses a value not returned by the SELECT clause:

```
SELECT p.product_name
FROM Order o, IN(o.lineItems) l JOIN o.customer c
WHERE c.lastname = 'Faehmel' AND c.firstname = 'Robert'
ORDER BY o.quantity
```

GROUP BY and HAVING Clauses

The GROUP BY clause allows you to group values according to a set of properties.

The following query groups the customers by their country and returns the number of customers per country:

```
SELECT c.country, COUNT(c)
 FROM Customer c GROUP BY c.country
```

The HAVING clause is used with the GROUP BY clause to further restrict the returned result of a query.

The following query groups orders by the status of their customer and returns the customer status plus the average totalPrice for all orders where the corresponding customers has the same status. In addition, it considers only customers with status 1, 2, or 3, so orders of other customers are not taken into account:

```
SELECT c.status, AVG(o.totalPrice)
 FROM Order o JOIN o.customer c
GROUP BY c.status HAVING c.status IN (1, 2, 3)
```

◆ ◆ ◆ CHAPTER 35

Using the Criteria API to Create Queries

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

The following topics are addressed here:

Overview of the Criteria and Metamodel APIs

Similar to JPQL, the Criteria API is based on the abstract schema of persistent entities, their relationships, and embedded objects. The Criteria API operates on this abstract schema to allow developers to find, modify, and delete persistent entities by invoking Java Persistence API entity operations. The Metamodel API works in concert with the Criteria API to model persistent entity classes for Criteria queries.

The Criteria API and JPQL are closely related and are designed to allow similar operations in their queries. Developers familiar with JPQL syntax will find equivalent object-level operations in the Criteria API.

The following simple Criteria query returns all instances of the `Pet` entity in the data source:

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> allPets = q.getResultList();
```

The equivalent JPQL query is:

```
SELECT p
FROM Pet p
```

This query demonstrates the basic steps to create a Criteria query:

1. Use an `EntityManager` instance to create a `CriteriaBuilder` object.
2. Create a query object by creating an instance of the `CriteriaQuery` interface. This query object's attributes will be modified with the details of the query.
3. Set the query root by calling the `from` method on the `CriteriaQuery` object.
4. Specify what the type of the query result will be by calling the `select` method of the `CriteriaQuery` object.
5. Prepare the query for execution by creating a `TypedQuery<T>` instance, specifying the type of the query result.
6. Execute the query by calling the `getResultList` method on the `TypedQuery<T>` object. Because this query returns a collection of entities, the result is stored in a `List`.

The tasks associated with each step are discussed in detail in this chapter.

To create a `CriteriaBuilder` instance, call the `getCriteriaBuilder` method on the `EntityManager` instance:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
```

The query object is created by using the `CriteriaBuilder` instance:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
```

The query will return instances of the `Pet` entity, so the type of the query is specified when the `CriteriaQuery` object is created to create a typesafe query.

The `FROM` clause of the query is set, and the root of the query specified, by calling the `from` method of the query object:

```
Root<Pet> pet = cq.from(Pet.class);
```

The `SELECT` clause of the query is set by calling the `select` method of the query object and passing in the query root:

```
cq.select(pet);
```

The query object is now used to create a `TypedQuery<T>` object that can be executed against the data source. The modifications to the query object are captured to create a ready-to-execute query:

```
TypedQuery<Pet> q = em.createQuery(cq);
```

This typed query object is executed by calling its `getResultList` method, because this query will return multiple entity instances. The results are stored in a `List<Pet>` collection-valued object.

```
List<Pet> allPets = q.getResultList();
```

Using the Metamodel API to Model Entity Classes

The Metamodel API is used to create a metamodel of the managed entities in a particular persistence unit. For each entity class in a particular package, a metamodel class is created with a trailing underscore and with attributes that correspond to the persistent fields or properties of the entity class.

The following entity class, `com.example.Pet`, has four persistent fields: `id`, `name`, `color`, and `owners`:

```
package com.example;

...

@Entity
public class Pet {
  @Id
  protected Long id;
  protected String name;
  protected String color;
  @ManyToOne
  protected Set<Owner> owners;
  ...
}
```

The corresponding Metamodel class is:

```
package com.example;

...

@Static Metamodel(Pet.class)
public class Pet_ {

  public static volatile SingularAttribute<Pet, Long> id;
  public static volatile SingularAttribute<Pet, String> name;
  public static volatile SingularAttribute<Pet, String> color;
  public static volatile SetAttribute<Pet, Owner> owners;
}
```

The metamodel class and its attributes are used in Criteria queries to refer to the managed entity classes and their persistent state and relationships.

Using Metamodel Classes

Metamodel classes that correspond to entity classes are of the following type:

```
javax.persistence.metamodel.EntityType<T>
```

Metamodel classes are typically generated by annotation processors either at development time or at runtime. Developers of applications that use Criteria queries may generate static metamodel classes by using the persistence provider's annotation processor or may obtain the metamodel class by either calling the `getModel` method on the query root object or first obtaining an instance of the `Metamodel` interface and then passing the entity type to the instance's `entity` method.

The following code snippet shows how to obtain the `Pet` entity's metamodel class by calling `Root<T>.getModel`:

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
EntityType<Pet> Pet_ = pet.getModel();
```

The following code snippet shows how to obtain the `Pet` entity's metamodel class by first obtaining a metamodel instance by using `EntityManager.getMetamodel` and then calling `entity` on the metamodel instance:

```
EntityManager em = ...;
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
```

Note – The most common use case is to generate type-safe static metamodel classes at development time. Obtaining the metamodel classes dynamically, by calling `Root<T>.getModel` or `EntityManager.getMetamodel` and then the `entity` method, doesn't allow for type-safety and doesn't allow the application to call persistent field or property names on the metamodel class.

Using the Criteria API and Metamodel API to Create Basic Typesafe Queries

The basic semantics of a Criteria query consists of a `SELECT` clause, a `FROM` clause, and an optional `WHERE` clause, similar to a JPQL query. Criteria queries set these clauses by using Java programming language objects, so the query can be created in a typesafe manner.

Creating a Criteria Query

The `javax.persistence.criteria.CriteriaBuilder` interface is used to construct

- Criteria queries
- Selections
- Expressions
- Predicates
- Ordering

To obtain an instance of the `CriteriaBuilder` interface, call the `getCriteriaBuilder` method on either an `EntityManager` or an `EntityManagerFactory` instance.

The following code shows how to obtain a `CriteriaBuilder` instance by using the `EntityManager.getCriteriaBuilder` method.

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
```

Criteria queries are constructed by obtaining an instance of the following interface:

```
javax.persistence.criteria.CriteriaQuery
```

`CriteriaQuery` objects define a particular query that will navigate over one or more entities. Obtain `CriteriaQuery` instances by calling one of the `CriteriaBuilder.createQuery` methods. For creating typesafe queries, call the `CriteriaBuilder.createQuery` method as follows:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
```

The `CriteriaQuery` object's type should be set to the expected result type of the query. In the preceding code, the object's type is set to `CriteriaQuery<Pet>` for a query that will find instances of the `Pet` entity.

In the following code snippet, a `CriteriaQuery` object is created for a query that returns a `String`:

```
CriteriaQuery<String> cq = cb.createQuery(String.class);
```

Query Roots

For a particular `CriteriaQuery` object, the root entity of the query, from which all navigation originates, is called the *query root*. It is similar to the `FROM` clause in a JPQL query.

Create the query root by calling the `from` method on the `CriteriaQuery` instance. The argument to the `from` method is either the entity class or an `EntityType<T>` instance for the entity.

The following code sets the query root to the `Pet` entity:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
```

The following code sets the query root to the `Pet` class by using an `EntityType<T>` instance:

```
EntityManager em = ...;
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet_);
```

Criteria queries may have more than one query root. This usually occurs when the query navigates from several entities.

The following code has two `Root` instances:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet1 = cq.from(Pet.class);
Root<Pet> pet2 = cq.from(Pet.class);
```

Querying Relationships Using Joins

For queries that navigate to related entity classes, the query must define a join to the related entity by calling one of the `From.join` methods on the query root object or another join object. The `join` methods are similar to the `JOIN` keyword in JPQL.

The target of the join uses the Metamodel class of type `EntityType<T>` to specify the persistent field or property of the joined entity.

The `join` methods return an object of type `Join<X, Y>`, where `X` is the source entity and `Y` is the target of the join. In the following code snippet, `Pet` is the source entity, `Owner` is the target, and `Pet_` is a statically generated metamodel class:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);

Root<Pet> pet = cq.from(Pet.class);
Join<Pet, Owner> owner = pet.join(Pet_.owners);
```

Joins can be chained together to navigate to related entities of the target entity without having to create a `Join<X, Y>` instance for each join:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);

Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = cq.join(Pet_.owners).join(Owner_.addresses);
```

Path Navigation in Criteria Queries

`Path` objects are used in the `SELECT` and `WHERE` clauses of a Criteria query and can be query root entities, join entities, or other `Path` objects. The `Path.get` method is used to navigate to attributes of the entities of a query.

The argument to the `get` method is the corresponding attribute of the entity's Metamodel class. The attribute can either be a single-valued attribute, specified by `@SingularAttribute` in the Metamodel class, or a collection-valued attribute, specified by one of `@CollectionAttribute`, `@SetAttribute`, `@ListAttribute`, or `@MapAttribute`.

The following query returns the names of all the pets in the data store. The `get` method is called on the query root, `pet`, with the `name` attribute of the `Pet` entity's Metamodel class, `Pet_`, as the argument:

```
CriteriaQuery<String> cq = cb.createQuery(String.class);

Root<Pet> pet = cq.from(Pet.class);
cq.select(pet.get(Pet_.name));
```

Restricting Criteria Query Results

The results of a query can be restricted on the `CriteriaQuery` object according to conditions set by calling the `CriteriaQuery.where` method. Calling the `where` method is analogous to setting the `WHERE` clause in a JPQL query.

The `where` method evaluates instances of the `Expression` interface to restrict the results according to the conditions of the expressions. `Expression` instances are created by using methods defined in the `Expression` and `CriteriaBuilder` interfaces.

The `Expression` Interface Methods

An `Expression` object is used in a query's `SELECT`, `WHERE`, or `HAVING` clause. Table 35–1 shows conditional methods you can use with `Expression` objects.

TABLE 35–1 Conditional Methods in the `Expression` Interface

Method	Description
`isNull`	Tests whether an expression is null
`isNotNull`	Tests whether an expression is not null
`in`	Tests whether an expression is within a list of values

The following query uses the `Expression.isNull` method to find all pets where the `color` attribute is null:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get(Pet_.color).isNull());
```

The following query uses the `Expression.in` method to find all brown and black pets:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get(Pet_.color).in("brown", "black"));
```

The `in` method also can check whether an attribute is a member of a collection.

Expression Methods in the `CriteriaBuilder` Interface

The `CriteriaBuilder` interface defines additional methods for creating expressions. These methods correspond to the arithmetic, string, date, time, and case operators and functions of JPQL. Table 35–2 shows conditional methods you can use with `CriteriaBuilder` objects.

TABLE 35–2 Conditional Methods in the `CriteriaBuilder` Interface

Conditional Method	Description
`equal`	Tests whether two expressions are equal
`notEqual`	Tests whether two expressions are not equal
`gt`	Tests whether the first numeric expression is greater than the second numeric expression
`ge`	Tests whether the first numeric expression is greater than or equal to the second numeric expression
`lt`	Tests whether the first numeric expression is less than the second numeric expression
`le`	Tests whether the first numeric expression is less than or equal to the second numeric expression
`between`	Tests whether the first expression is between the second and third expression in value
`like`	Tests whether the expression matches a given pattern

The following code uses the `CriteriaBuilder.equal` method:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get(Pet_.name), "Fido"));
...
```

The following code uses the `CriteriaBuilder.gt` method:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Date someDate = new Date(...);
cq.where(cb.gt(pet.get(Pet_.birthday), date));
```

The following code uses the `CriteriaBuilder.between` method:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Date firstDate = new Date(...);
Date secondDate = new Date(...);
cq.where(cb.between(pet.get(Pet_.birthday), firstDate, secondDate));
```

The following code uses the `CriteriaBuilder.like` method:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.like(pet.get(Pet_.name), "*do"));
```

Multiple conditional predicates can be specified by using the compound predicate methods of the `CriteriaBuilder` interface, as shown in Table 35–3.

TABLE 35–3 Compound Predicate Methods in the `CriteriaBuilder` Interface

Method	Description
`and`	A logical conjunction of two Boolean expressions
`or`	A logical disjunction of two Boolean expressions
`not`	A logical negation of the given Boolean expression

The following code shows the use of compound predicates in queries:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get(Pet_.name), "Fido")
    .and(cb.equal(pet.get(Pet_.color), "brown")));
```

Managing Criteria Query Results

For queries that return more than one result, it's often helpful to organize those results. The `CriteriaQuery` interface defines the `orderBy` method to order query results according to attributes of an entity. The `CriteriaQuery` interface also defines the `groupBy` method to group the results of a query together according to attributes of an entity, and the `having` method to restrict those groups according to a condition.

Ordering Results

The order of the results of a query can be set by calling the `CriteriaQuery.orderBy` method and passing in an `Order` object. `Order` objects are created by calling either the `CriteriaBuilder.asc` or the `CriteriaBuilder.desc` method. The `asc` method is used to

order the results by ascending value of the passed expression parameter. The `desc` method is used to order the results by descending value of the passed expression parameter. The following query shows the use of the `desc` method:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
cq.orderBy(cb.desc(pet.get(Pet_.birthday)));
```

In this query, the results will be ordered by the pet's birthday from highest to lowest. That is, pets born in December will appear before pets born in May.

The following query shows the use of the `asc` method:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = cq.join(Pet_.owners).join(Owner_.address);
cq.select(pet);
cq.orderBy(cb.asc(address.get(Address_.postalCode)));
```

In this query, the results will be ordered by the pet owner's postal code from lowest to highest. That is, pets whose owner lives in the 10001 zip code will appear before pets whose owner lives in the 91000 zip code.

If more than one `Order` object is passed to `orderBy`, the precedence is determined by the order in which they appear in the argument list of `orderBy`. The first `Order` object has precedence.

The following code orders results by multiple criteria:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Pet, Owner> owner = cq.join(Pet_.owners);
cq.select(pet);
cq.orderBy(cb.asc(owner.get(Owner_.lastName), owner.get(Owner_.firstName)));
```

The results of this query will be ordered alphabetically by the pet owner's last name, then first name.

Grouping Results

The `CriteriaQuery.groupBy` method partitions the query results into groups. These groups are set by passing an expression to `groupBy`:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.groupBy(pet.get(Pet_.color));
```

This query returns all `Pet` entities and groups the results by the pet's color.

The `CriteriaQuery.having` method is used in conjunction with `groupBy` to filter over the groups. The `having` method takes a conditional expression as a parameter. By calling the `having` method, the query result is restricted according to the conditional expression:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.groupBy(pet.get(Pet_.color));
cq.having(cb.in(pet.get(Pet_.color)).value("brown").value("blonde"));
```

In this example, the query groups the returned `Pet` entities by color, as in the preceding example. However, the only returned groups will be `Pet` entities where the `color` attribute is set to `brown` or `blonde`. That is, no gray-colored pets will be returned in this query.

Executing Queries

To prepare a query for execution, create a `TypedQuery<T>` object with the type of the query result by passing the `CriteriaQuery` object to `EntityManager.createQuery`.

Queries are executed by calling either `getSingleResult` or `getResultList` on the `TypedQuery<T>` object.

Single-Valued Query Results

The `TypedQuery<T>.getSingleResult` method is used for executing queries that return a single result:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
...
TypedQuery<Pet> q = em.createQuery(cq);
Pet result = q.getSingleResult();
```

Collection-Valued Query Results

The `TypedQuery<T>.getResultList` method is used for executing queries that return a collection of objects:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
...
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> results = q.getResultList();
```

◆ ◆ ◆ CHAPTER 36

Creating and Using String-Based Criteria Queries

This chapter describes how to create weakly-typed string-based Criteria API queries.

The following topics are addressed here:

Overview of String-Based Criteria API Queries

String-based Criteria API queries ("string-based queries") are Java programming language queries that use strings rather than strongly-typed metamodel objects to specify entity attributes when traversing a data hierarchy. String-based queries are constructed similarly to metamodel queries, can be static or dynamic, and can express the same kind of queries and operations as strongly-typed metamodel queries.

Strongly-typed metamodel queries are the preferred method of constructing Criteria API queries. The main advantage of string-based queries over metamodel queries is the ability to construct Criteria queries at development time without the need to generate static metamodel classes or otherwise access dynamically generated metamodel classes. The main disadvantage to string-based queries is their lack of type safety, which may lead to runtime errors due to type mismatches that would be caught at development time when using strongly-typed metamodel queries.

For information on constructing criteria queries, see Chapter 35, "Using the Criteria API to Create Queries."

Creating String-Based Queries

To create a string-based query, specify the attribute names of entity classes directly as strings, rather than the attributes of the metamodel class. For example, this query finds all `Pet` entities where the value of the `name` attribute is `Fido`:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get("name"), "Fido"));
...
```

The name of the attribute is specified as a string. This query is the equivalent of the following metamodel query:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get(Pet_.name), "Fido"));
```

Note – Type mismatch errors in string-based queries won't appear until the code is executed at runtime, unlike in the above metamodel query, where type mismatches will be caught at compile time.

Joins are specified in the same way:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = pet.join("owners").join("addresses");
...
```

All the conditional expressions, method expressions, path navigation methods, and result restriction methods used in metamodel queries can be used in string-based queries. In each case, the attributes are specified using strings. For example, here is a string-based query that uses the `in` expression:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get("color").in("brown", "black"));
```

Here is a string-based query that orders the results in descending order by date:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
cq.orderBy(cb.desc(pet.get("birthday")));
```

Executing String-Based Queries

String-based queries are executed similarly to strongly-typed Criteria queries. First create a `javax.persistence.TypedQuery` object by passing the criteria query object to the `EntityManager.createQuery` method and then call either `getSingleResult` or `getResultList` on the query object to execute the query.

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get("name"), "Fido"));
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> results = q.getResultList();
```

◆ ◆ ◆ CHAPTER 37

Controlling Concurrent Access to Entity Data with Locking

This chapter details how to handle concurrent access to entity data, and the locking strategies available to Java Persistence API application developers.

The following topics are addressed here:

Overview of Entity Locking and Concurrency

Entity data is *concurrently accessed* if the data in a data source is accessed at the same time by multiple applications. Special care must be taken to ensure that the underlying data's integrity is preserved when accessed concurrently.

When data is updated in the database tables in a transaction, the persistence provider assumes the database management system will hold short-term read locks and long-term write locks to maintain data integrity. Most persistence providers will delay database writes until the end of the transaction, except when the application explicitly calls for a flush (that is, the application calls the `EntityManager.flush` method or executes queries with the flush mode set to `AUTO`).

By default, persistence providers use *optimistic locking*, where, before committing changes to the data, the persistence provider checks that no other transaction has modified or deleted the data since the data was read. This is accomplished by a version column in the database table, with a corresponding version attribute in the entity class. When a row is modified, the version value is incremented. The original transaction checks the version attribute, and if the data has been modified by another transaction, a `javax.persistence.OptimisticLockException` will be thrown, and the original transaction will be rolled back. When the application specifies optimistic lock modes, the persistence provider verifies that a particular entity has not changed since it was read from the database even if the entity data was not modified.

Pessimistic locking goes further than optimistic locking. With pessimistic locking, the persistence provider creates a transaction that obtains a long-term lock on the data until the

transaction is completed, which prevents other transactions from modifying or deleting the data until the lock has ended. Pessimistic locking is a better strategy than optimistic locking when the underlying data is frequently accessed and modified by many transactions.

Note – Using pessimistic locks on entities that are not subject to frequent modification may result in decreased application performance.

Using Optimistic Locking

The `javax.persistence.Version` annotation is used to mark a persistent field or property as a version attribute of an entity. By adding a version attribute, the entity is enabled for optimistic concurrency control. The version attribute is read and updated by the persistence provider when an entity instance is modified during a transaction. The application may read the version attribute, but *must not* modify the value.

Note – Although some persistence providers may support optimistic locking for entities that do not have version attributes, portable applications should always use entities with version attributes when using optimistic locking. If the application attempts to lock an entity without a version attribute, and the persistence provider doesn't support optimistic locking for non-versioned entities, a `PersistenceException` will be thrown.

The `@Version` annotation has the following requirements:

- Only a single `@Version` attribute may be defined per entity.
- The `@Version` attribute must be in the primary table for an entity mapped to multiple tables.
- The type of the `@Version` attribute must be one of the following: `int`, `Integer`, `long`, `Long`, `short`, `Short`, or `java.sql.Timestamp`.

The following code snippet shows how to define a version attribute in an entity with persistent fields:

```
@Version
protected int version;
```

The following code snippet shows how to define a version attribute in an entity with persistent properties:

```
@Version
protected Short getVersion() { ... }
```

Lock Modes

The application may increase the level of locking for an entity by specifying the use of lock modes. Lock modes may be specified to increase the level of optimistic locking or to request the use of pessimistic locks.

The use of optimistic lock modes causes the persistence provider to check the version attributes for entities that were read (but not modified) during a transaction as well as for entities that were updated.

The use of pessimistic lock modes specifies that the persistence provider is to immediately acquire long-term read or write locks for the database data corresponding to entity state.

The lock mode for an entity operation may be set by specifying one of the lock modes defined in the `javax.persistence.LockModeType` enumerated type, listed in Table 37–1.

TABLE 37–1 Lock Modes for Concurrent Entity Access

Lock Mode	Description
`OPTIMISTIC`	Obtain an optimistic read lock for all entities with version attributes.
`OPTIMISTIC_FORCE_INCREMENT`	Obtain an optimistic read lock for all entities with version attributes, and increment the version attribute value.
`PESSIMISTIC_READ`	Immediately obtain a long-term read lock on the data to prevent the data from being modified or deleted. Other transactions may read the data while the lock is maintained, but may not modify or delete the data. The persistence provider is permitted to obtain a database write lock when a read lock was requested, but not vice versa.
`PESSIMISTIC_WRITE`	Immediately obtain a long-term write lock on the data to prevent the data from being read, modified, or deleted.
`PESSIMISTIC_FORCE_INCREMENT`	Immediately obtain a long-term lock on the data to prevent the data from being modified or deleted, and increment the version attribute of versioned entities.
`READ`	A synonym for `OPTIMISTIC`. Use of `LockModeType.OPTIMISTIC` is to be preferred for new applications.

TABLE 37–1 Lock Modes for Concurrent Entity Access *(Continued)*

Lock Mode	Description
`WRITE`	A synonym for `OPTIMISTIC_FORCE_INCREMENT`. Use of `LockModeType.OPTIMISTIC_FORCE_INCREMENT` is to be preferred for new applications.
`NONE`	No additional locking will occur on the data in the database.

Setting the Lock Mode

The lock mode may be specified by one of the following techniques:

- Calling the `EntityManager.lock` and passing in one of the lock modes:

```
EntityManager em = ...;
Person person = ...;
em.lock(person, LockModeType.OPTIMISTIC);
```

- Calling one of the `EntityManager.find` methods that takes the lock mode as a parameter:

```
EntityManager em = ...;
String personPK = ...;
Person person = em.find(Person.class, personPK,
        LockModeType.PESSIMISTIC_WRITE);
```

- Calling one of the `EntityManager.refresh` methods that takes the lock mode as a parameter:

```
EntityManager em = ...;
String personPK = ...;
Person person = em.find(Person.class, personPK);
...
em.refresh(person, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
```

- Calling the `Query.setLockMode` or `TypedQuery.setLockMode` method, passing the lock mode as the parameter:

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

- Adding a `lockMode` element to the `@NamedQuery` annotation:

```
@NamedQuery(name="lockPersonQuery",
  query="SELECT p FROM Person p WHERE p.name LIKE :name",
  lockMode=PESSIMISTIC_READ)
```

Using Pessimistic Locking

Versioned entities as well as entities that do not have version attributes can be locked pessimistically.

To lock entities pessimistically, set the lock mode to `PESSIMISTIC_READ`, `PESSIMISTIC_WRITE`, or `PESSIMISTIC_FORCE_INCREMENT`.

If a pessimistic lock cannot be obtained on the database rows, and the failure to lock the data results in a transaction rollback, a `PessimisticLockException` is thrown. If a pessimistic lock cannot be obtained, but the locking failure doesn't result in a transaction rollback, a `LockTimeoutException` is thrown.

Pessimistically locking a version entity with `PESSIMISTIC_FORCE_INCREMENT` results in the version attribute being incremented even if the entity data is unmodified. When pessimistically locking a versioned entity, the persistence provider will perform the version checks that occur during optimistic locking, and if the version check fails, an `OptimisticLockException` will be thrown. Attempting to lock a non-versioned entity with `PESSIMISTIC_FORCE_INCREMENT` is not portable and may result in a `PersistenceException` if the persistence provider doesn't support optimistic locks for non-versioned entities. Locking a versioned entity with `PESSIMISTIC_WRITE` results in the version attribute being incremented if the transaction was successfully committed.

Pessimistic Locking Timeouts

The length of time in milliseconds the persistence provider should wait to obtain a lock on the database tables may be specified using the `javax.persistence.lock.timeout` property. If the time it takes to obtain a lock exceeds the value of this property, a `LockTimeoutException` will be thrown, but the current transaction will not be marked for rollback. If this property is set to `0`, the persistence provider should throw a `LockTimeoutException` if it cannot immediately obtain a lock.

Note – Portable applications should not rely on the setting of `javax.persistence.lock.timeout`, as the locking strategy and underlying database may mean that the timeout value cannot be used. The value of `javax.persistence.lock.timeout` is a hint, not a contract.

This property may be set programmatically by passing it to the `EntityManager` methods that allow lock modes to be specified, the `Query.setLockMode` and `TypedQuery.setLockMode` methods, the `@NamedQuery` annotation, and as a property to the `Persistence.createEntityManagerFactory` method. It may also be set as a property in the `persistence.xml` deployment descriptor.

If `javax.persistence.lock.timeout` is set in multiple places, the value will be determined in the following order:

1. The argument to one of the `EntityManager` or `Query` methods.
2. The setting in the `@NamedQuery` annotation.
3. The argument to the `Persistence.createEntityManagerFactory` method.
4. The value in the `persistence.xml` deployment descriptor.

◆ ◆ ◆ CHAPTER 38

Using a Second-Level Cache with Java Persistence API Applications

This chapter explains how to modify the second-level cache mode settings to improve the performance of applications that use the Java Persistence API.

The following topics are addressed here:

Overview of the Second-Level Cache

A *second-level cache* is a local store of entity data managed by the persistence provider to improve application performance. A second-level cache helps improve performance by avoiding expensive database calls, keeping the entity data local to the application. A second-level cache is typically transparent to the application, as it is managed by the persistence provider and underlies the persistence context of an application. That is, the application reads and commits data through the normal entity manager operations without knowing about the cache.

Note – Persistence providers are not required to support a second-level cache. Portable applications should not rely on support by persistence providers for a second-level cache.

The second-level cache for a persistence unit may be configured to one of several second-level cache modes. The following cache mode settings are defined by the Java Persistence API.

TABLE 38–1 Cache Mode Settings for the Second-Level Cache

Cache Mode Setting	Description
ALL	All entity data is stored in the second-level cache for this persistence unit.

TABLE 38–1 Cache Mode Settings for the Second-Level Cache *(Continued)*

Cache Mode Setting	Description
`NONE`	No data is cached in the persistence unit. The persistence provider must not cache any data.
`ENABLE_SELECTIVE`	Enable caching for entities that have been explicitly set with the `@Cacheable` annotation.
`DISABLE_SELECTIVE`	Enable caching for all entities except those that have been explicitly set with the `@Cacheable(false)` annotation.
`UNSPECIFIED`	The caching behavior for the persistence unit is undefined. The persistence provider's default caching behavior will be used.

One consequence of using a second-level cache in an application is that the underlying data may have changed in the database tables, while the value in the cache has not, a circumstance called a *stale read*. Stale reads may be avoided by changing the second-level cache to one of the cache mode settings, controlling which entities may be cached (described in "Controlling Whether Entities May Be Cached" on page 684), or changing the cache's retrieval or store modes (described in "Setting the Cache Retrieval and Store Modes" on page 686). Which strategies best avoid stale reads are application dependent.

Controlling Whether Entities May Be Cached

The `javax.persistence.Cacheable` annotation is used to specify that an entity class, and any subclasses, may be cached when using the `ENABLE_SELECTIVE` or `DISABLE_SELECTIVE` cache modes. Subclasses may override the `@Cacheable` setting by adding a `@Cacheable` annotation and changing the value.

To specify that an entity may be cached, add a `@Cacheable` annotation at the class level:

```
@Cacheable
@Entity
public class Person { ... }
```

By default, the `@Cacheable` annotation is `true`. The following example is equivalent:

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

To specify that an entity must not be cached, add a `@Cacheable` annotation and set it to `false`:

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

When the `ENABLE_SELECTIVE` cache mode is set, the persistence provider will cache any entities that have the `@Cacheable(true)` annotation and any subclasses of that entity that have not been overridden. The persistence provider will not cache entities that have `@Cacheable(false)` or have no `@Cacheable` annotation. That is, the `ENABLE_SELECTIVE` mode will cache only entities that have been explicitly marked for the cache using the `@Cacheable` annotation.

When the `DISABLE_SELECTIVE` cache mode is set, the persistence provider will cache any entities that *do not* have the `@Cacheable(false)` annotation. Entities that do not have `@Cacheable` annotations, and entities with the `@Cacheable(true)` annotation will be cached. That is, the `DISABLE_SELECTIVE` mode will cache all entities that have not been explicitly prevented from being cached.

If the cache mode is set to `UNDEFINED`, or is left unset, the behavior of entities annotated with `@Cacheable` is undefined. If the cache mode is set to `ALL` or `NONE`, the value of the `@Cacheable` annotation is ignored by the persistence provider.

Specifying the Cache Mode Settings to Improve Performance

To adjust the cache mode settings for a persistence unit, specify one of the cache modes as the value of the `shared-cache-mode` element in the `persistence.xml` deployment descriptor (shown in **bold**):

```
<persistence-unit name="examplePU" transaction-type="JTA">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <jta-data-source>jdbc/__default</jta-data-source>
  <shared-cache-mode>DISABLE_SELECTIVE</shared-cache-mode>
</persistence-unit>
```

Note – Because support for a second-level cache is not required by the Java Persistence API specification, setting the second-level cache mode in `persistence.xml` will have no effect when using a persistence provider that does not implement a second-level cache.

Alternatively, the shared cache mode may be specified by setting the `javax.persistence.sharedCache.mode` property to one of the shared cache mode settings:

```
EntityManagerFactor emf =
    Persistence.createEntityManagerFactory(
        "myExamplePU", new Properties().add(
            "javax.persistence.sharedCache.mode", "ENABLE_SELECTIVE"));
```

Setting the Cache Retrieval and Store Modes

If the second-level cache has been enabled for a persistence unit by setting the shared cache mode, the behavior of the second-level cache can be further modified by setting the `javax.persistence.cache.retrieveMode` and `javax.persistence.cache.storeMode` properties. These properties may be set at the persistence context level by passing the property name and value to the `EntityManager.setProperty` method, or may be set on a per-`EntityManager` operation (`EntityManager.find` or `EntityManager.refresh`) or per-query level.

Cache Retrieval Mode

The cache retrieval mode, set by the `javax.persistence.retrieveMode` property, controls how data is read from the cache for calls to the `EntityManager.find` method and from queries.

The `retrieveMode` property can be set to one of the constants defined by the `javax.persistence.CacheRetrieveMode` enumerated type, either `USE` (the default) or `BYPASS`. When it is set to `USE`, data is retrieved from the second-level cache, if available. If the data is not in the cache, the persistence provider will read it from the database. When it is set to `BYPASS`, the second-level cache is bypassed and a call to the database is made to retrieve the data.

Cache Store Mode

The cache store mode, set by the `javax.persistence.storeMode` property, controls how data is stored in the cache.

The `storeMode` property can be set to one of the constants defined by the `javax.persistence.CacheStoreMode` enumerated type, either `USE` (the default), `BYPASS`, or `REFRESH`. When set to `USE` the cache data is created or updated when data is read from or committed to the database. If data is already in the cache, setting the store mode to `USE` will not force a refresh when data is read from the database.

When the store mode is set to `BYPASS`, data read from or committed to the database is *not* inserted or updated in the cache. That is, the cache is unchanged.

When the store mode is set to `REFRESH`, the cache data is created or updated when data is read from or committed to the database, and a refresh is forced on data in the cache upon database reads.

Setting the Cache Retrieval or Store Mode

To set the cache retrieval or store mode for the persistence context, call the `EntityManager.setProperty` method with the property name and value pair:

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

To set the cache retrieval or store mode when calling the `EntityManger.find` or `EntityManager.refresh` methods, first create a `Map<String, Object>` instance and add a name/value pair as follows:

```
EntityManager em = ...;
Map<String, Object> props = new HashMap<String, Object>();
props.put("javax.persistence.cache.retrieveMode", "BYPASS");
String personPK = ...;
Person person = em.find(Person.class, personPK, props);
```

Note – The cache retrieve mode is ignored when calling the `EntityManager.refresh` method, as calls to `refresh` always result in data being read from the database, not the cache.

To set the retrieval or store mode when using queries, call the `Query.setHint` or `TypedQuery.setHint` methods, depending on the type of query:

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

Setting the store or retrieve mode in a query or when calling the `EntityManager.find` or `EntityManager.refresh` method overrides the setting of the entity manager.

Controlling the Second-Level Cache Programmatically

The `javax.persistence.Cache` interface defines methods for interacting with the second-level cache programmatically. The `Cache` interface defines methods to check whether a particular entity has cached data, to remove a particular entity from the cache, to remove all instances (and instances of subclasses) of an entity class from the cache, and to clear the cache of all entity data.

Note – If the second-level cache has been disabled, calls to the `Cache` interface's methods have no effect, except for `contains`, which will always return `false`.

Checking Whether an Entity's Data Is Cached

Call the `Cache.contains` method to find out whether a given entity is currently in the second-level cache. The `contains` method returns `true` if the entity's data is cached, and `false` if the data is not in the cache.

```
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
String personPK = ...;
if (cache.contains(Person.class, personPK)) {
  // the data is cached
} else {
  // the data is NOT cached
}
```

Removing an Entity from the Cache

Call one of the `Cache.evict` methods to remove a particular entity or all entities of a given type from the second-level cache. To remove a particular entity from the cache, call the `evict` method and pass in the entity class and the primary key of the entity:

```
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
String personPK = ...;
cache.evict(Person.class, personPK);
```

To remove all instances of a particular entity class, including subclasses, call the `evict` method and specify the entity class:

```
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
cache.evict(Person.class);
```

All instances of the `Person` entity class will be removed from the cache. If the `Person` entity has a subclass, `Student`, calls to the above method will remove all instances of `Student` from the cache as well.

Removing All Data from the Cache

Call the `Cache.evictAll` method to completely clear the second-level cache:

```
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
cache.evictAll();
```