Part VI
Persistence

Part VI explores the Java Persistence API.

This part contains the following chapters:

- Chapter 32, Introduction to the Java Persistence API
- Chapter 33, Running the Persistence Examples

- Chapter 34, The Java Persistence Query Language
- Chapter 35, Using the Criteria API to Create Queries
- Chapter 36, Creating and Using String-Based Criteria Queries
- Chapter 37, Controlling Concurrent Access to Entity Data with Locking
- Chapter 38, Improving the Performance of Java Persistence API Applications By Setting a Second-Level Cache

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
- Entity Inheritance
- Managing Entities
- Querying Entities
- Further Information about Persistence

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 customer entity when it is removed from the relationship:

```
@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.

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.

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