Tema 16

Acceso a Bases de Datos Usando JPA

En el Tema 15: Acceso a base de Datos Usando JDBC vimos que Java nos permite crear aplicaciones que accedan a base de datos usando un protocolo en la forma de una API: la API JDBC. Una de las metas de JDBC era la de aislar el código de la aplicación de las diferentes implementaciones de manejadores de base de datos, permitiendo substituir un manejador de bases de datos por otro sin modificar una línea de código. Desafortunadamente JDBC no proporciona una completa independencia de los manejadores de bases de datos ya que hay ciertas diferencias entre esas implementaciones que la JDBC no puede manejar y que nos obliga a hacer modificaciones en la aplicación si se cambia de un manejador de base de datos a otro.

Desajuste de Impedancias

En el modelo del dominio, que es el que usamos en el diseño de nuestra aplicación, tenemos clases, las bases de datos tienen tablas. Otro de los problemas de JDBC es el trabajo que debe hacer el programador para convertir un objeto a un renglón de una tabla y visceversa. Esa dificultad en la traducción entre una clase y una tabla se debe a las diferencias entre ambos modelos. A esas diferencias se le conoce como Desajuste de impedancias y se ilustran en la tabla 16.1:

Tabla 16.1 Desajuste de Impedancias entre los mundos OO y Relacional

Objetos, clases	Tablas, renglones
Atributos, propiedades	Columnas
Identidad	Llave primaria
Relación/referencia a otra entidad	Llave foránea
Herencia/polimorfismo	No soportado
Métodos	Indirectamente: Lógica SQL, procedimientos almacenados, triggers
Código portable	No necesariamente portable, dependiente del vendedor.

La API de Persistencia de Java, JPA

La JPA es una capa de software por encima de JDBC que busca redicirle al programador de la tarea de conversión entre el mundo OO y el mundo relacional proveyendo de los siguientes servicios:

- Realiza el Mapeo Objeto - Relacional, esto es establece cómo persistir los objetos en una base de datos.
- Persiste los objetos a la base de datos. Realiza las operaciones de altas, bajas y cambios de objetos en la base de datos.
- Permite la búsqueda y recuperación de objetos de la base de datos mediante su propio Lenguaje de Consulta de Persistencia de Java o SQL.

Mapeo Objeto – Relacional

El Mapeo Objeto – Relacional, ORM, es un procedimiento para establecer en que registro de una tabla se almacena cada objeto y en que columna se almacenará cada atributo. Adicionalmente, ORM establece cómo se mapean las relaciones entre los objetos incluyendo la herencia.

Mapeo Uno a Uno

En una relación uno a uno unidireccional, un objeto de una clase tiene una referencia a cuando mucho otro objeto de otra clase. Por ejemplo, un cliente de una empresa puede tener una cuenta de usuario en esa empresa para acceder a su extranet. Aquí supondremos que un cliente sólo puede tener como máximo una cuenta de usuario en esa empresa. Las clases `Cliente` y `Usuario` son las siguientes:

```
public class Cliente {
   protected String clienteId;
   protected String nombre;
   protected String direccion
   protected Usuario usuario;
}

public class Usuario {
   protected String usuarioId;
   protected String contrasena;
   protected String rol;
}
```

Las clases anteriores se mapean a dos tablas relacionadas por una llave foranea de la siguiente manera:

clientes

```
clienteId not null, primary key varchar(10)
nombre    not null,             varchar(35)
direccion not null,             varchar(35)
usuarioId not null,             varchar(10)
          foreing key (usuarios(usuarioId))
```

usuarios

```
usuarioId  not null, primary key varchar(10)
```

```
contrasena not null,            varchar(10)
rol        not null,            varchar(20)
```

Si queremos que la relación sea bidireccional, dos objetos de diferentes clases tienen una referencia al otro objeto. Si para las clases `Cliente` y `Usuario` deseamos que la relación sea bidireccional tendremos que su código es el siguiente:

```
public class Cliente {
   protected String clienteId;
   protected String nombre;
   protected String direccion
   protected Usuario usuario;
}

public class Usuario {
   protected String usuarioId;
   protected String contrasena;
   protected String rol;
   protected Cliente cliente;
}
```

Mapeo Muchos a Uno

En una relación muchos a uno unidireccional, muchos objetos de una clase tiene una referencia a cuando mucho otro objeto de otra clase. Por ejemplo, varios medios (canciones o películas) pueden tener el mismo género. Las clases `Medio` y `Genero` son las siguientes:

```
public class Medio {
   protected String clave;
   protected String titulo;
   protected Genero genero;
   protected int duracion;
   protected Date fecha;
}

public class Genero {
   protected String cveGenero;
   protected String nombre;
   protected char tipoMedio;
}
```

Las clases anteriores se mapean a dos tablas relacionadas por una llave foranea de la siguiente manera:

medios

```
clave     not null, primary key varchar(10)
titulo    not null,             varchar(35)
cveGenero not null,             varchar(10)
          foreing key (generos(cveGenero))
duracion  not null,              integer
```

```
fecha                           Date
```

generos

```
cveGenero  not null, primary key varchar(10)
nombre not null,                 varchar(20)
tipoMedio  not null,             char
```

Mapeo Uno a Muchos

En una relación uno a muchos o muchos a uno bidireccional, el lado de uno de la relación tiene una referencia a una colección (`Collection`, `List` o `Set`) de objetos de la otra clase. Por ejemplo, las clases `Cancion` y `Genero` con una relación uno a muchos tendrían la forma siguiente:

```
public class Medio {
   protected String clave;
   protected String titulo;
   protected Genero genero;
   protected int duracion;
   protected Date fecha;
}

public class Genero {
   protected String cveGenero;
   protected String nombre;
   protected char tipoMedio;
   List<Medio> listamedios;
}
```

Las clases anteriores se mapean a las mismas tablas de la relación muchos a uno.

Mapeo Muchos a Muchos

En una relación muchos a muchos, dos objetos de diferente clase tienen una referencia a una colección de objetos de la otra clase. Por ejemplo, una canción puede estar en varios álbumes y un álbum puede tener varias canciones. Las clases `Cancion` y `Album` son las siguientes:

```
public class Cancion {
   protected String clave;
   protected String titulo;
   protected Genero genero;
   protected int duracion;
   protected Date fecha;
   protected List<Album> listaAlbumes;
}

public class Album {
   protected String cveAlbum;
```

```
   protected String nombre;
   protected List<Cancion> listaCanciones;
}
```

Las clases anteriores se mapean a dos tablas relacionadas por una tabla conjunta de la siguiente manera:

canciones

```
clave     not null, primary key varchar(10)
titulo    not null,             varchar(35)
cveGenero not null,             varchar(10)
          foreing key (generos(cveGenero))
duracion  not null,              integer
fecha                            Date
cveAlbum   not null,             varchar(10)
```

albumes

```
cveAlbum  not null, primary key varchar(10)
nombre not null,                 varchar(20)
clave  not null,                 varchar(10)
```

canciones_albumes

```
clave     not null, primary key varchar(10)
          foreing key (canciones(clave))
cveAlbum  not null, primary key varchar(10)
          foreing key (albumes(cveAlbum))
```

Herencia

El concepto de herencia no tiene un equivalente directo en el mundo relacional. Sin embargo hay varias estrategias para mapear la herencia entre clases a tablas:

- Una sola tabla
- Tablas conjuntas
- Una tabla por clase

Para ilustrar las tres estrategias, consideremos las siguientes tres clases:

```
public class Medio {
   protected String clave;
   protected String titulo;
   protected Genero genero;
   protected int duracion;
   protected Date fecha;
}

public class Cancion extends Medio {
   private String interprete;
   private String autor;
   private d String album;
}
```

```
public class Pelicula extends Medio {
   private String actor1;
   private String actor2;
   private String director;
}
```

Estrategia de una Sola Tabla

En esta estrategia, todas las clases de la jerarquía se mapean a una sola tabla. Los diferentes objetos de la jerrquía se identifican usando una columna especial llamada columna descriminadora que contiene un valor único para cada tipo de objeto en un registro dado:

medios

```
clave     not null, primary key varchar(10)
titulo    not null,             varchar(35)
tipoMedio not null,             varchar(1)
cveGenero not null,             varchar(10)
          foreing key (generos(cveGenero))
duracion  not null,              integer
fecha                            Date
interprete                       varchar(35)
autor                            varchar(35)
album                            varchar(20)
actor1                           varchar(35)
actor2                           varchar(35)
director                         varchar(35)
```

En este caso `tipoMedio` es la columna descriminadora y contendrá un valor diferente para cada tipo de objeto: `Cancion` o `Película`. Las columnas que no corresponden a un objeto en particular teinen valores `NULL`. Esta estrategia es simple de usar. Sin embargo tiene la desventaja de desperdiciar espacio debido a todos esos valores `NULL`. Adicionalmente todas las columnas correspondientes a los atributos exclusivas de las clases hijas no pueden ser obligatorias, `NOT NULL`.

Estrategia de Tablas Conjuntas

En esta estrategia, se utilizan relaciones uno a uno para modelar la herencia. En esta estrategia se crean tablas separadas para cada clase de la jerarquía y relacionar los descendientes directos en la jerarquía con relaciones uno a uno:

medios

```
clave     not null, primary key varchar(10)
titulo    not null,             varchar(35)
tipoMedio not null,             varchar(1)
cveGenero not null,             varchar(10)
```

```
          foreing key (generos(cveGenero))
duracion  not null,              integer
fecha                            Date
```

canciones

```
clave     not null, primary key varchar(10)
          foreing key (medios(clave))
interprete                       varchar(35)
autor                            varchar(35)
album                            varchar(20)
```

peliculas

```
clave     not null, primary key varchar(10)
          foreing key (medios(clave))
actor1                           varchar(35)
actor2                           varchar(35)
director                         varchar(35)
```

En esta estrategia se sigue utilizando la columna descriminadora para distinguir los diferentes tipos de las jerarquías Aunque esta estrategia es la mejor en términos de la perspectiva de diseño, en términos de desempeño es peor que la de una sola tabla por que requiere de tablas conjuntas para las consultas.

Estrategia de una Tabla por Clase

En esta estrategia, cada una de las clases concretas (no abstractas) de la jerarquía se almacena en su propia tablay no hay relaciones entre las tablas:

medios

```
clave     not null, primary key varchar(10)
titulo    not null,              varchar(35)
cveGenero not null,              varchar(10)
          foreing key (generos(cveGenero))
duracion  not null,               integer
fecha                             Date
```

canciones

```
clave     not null, primary key varchar(10)
titulo    not null,              varchar(35)
cveGenero not null,              varchar(10)
          foreing key (generos(cveGenero))
duracion  not null,               integer
fecha                             Date
interprete                        varchar(35)
autor                             varchar(35)
album                             varchar(20)
```

peliculas

```
clave     not null, primary key varchar(10)
titulo    not null,              varchar(35)
cveGenero not null,              varchar(10)
```

```
          foreing key (generos(cveGenero))
duracion  not null,              integer
fecha                            Date
actor1                           varchar(35)
actor2                           varchar(35)
director                         varchar(35)
```

Esta estrategia aunque es la más fácil de entender, es la peor desde el punto de vista de objetos y relacional. En la especificación de JPA esta estrategia es opcional y puede no estar soportada por una implementación.

Entidades

En la sección anterior se estableció como pueden mapearse las clases y objetos a tablas y registros de tal manera que se puedan persistir los objetos en una base de datos. En JPA, una clase cuyo estado (valores de sus atributos) van a persistirse en una tabla, se llama **Entidad**. Para poder persistir un objeto, el provedor de la persistencia debe saber lo siguiente:

- La tabla en la que se va a persistir el objeto
- Cuáles atributos se van a persitir y en qué columnas de la tabla
- Cómo se establece la identidad del objeto a persistir
- Qué relaciones tiene este objeto con otros objetos de otras clases que también se van a persistir

Esta información se la agregamos a nuestra entidad mediante una serie de anotaciones que se verán a continuación:

@Entity

La anotación @Entity marca a una clase como una entidad cuyo estado será persistido en una base de datosy que será identificado univocamente. La sintaxis de esta anotación es:

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

Todas las entidades no abstractas deben tener un constructor vacío público o protegido.

Una entidad puede heredar de otra entidad o inclusive de una clase que no sea una entidad. Si la clase padre es una entidad, también su estado se estado se guarda al persistir a la clase hija. Si la clase padre no es una entidad su estado no se guarda.

@Table

Esta anotación permite mapear el nombre de la entidad al nombre de la tabla en la que se persistirán los objetos de esta entidad. Esta anotación es opcional. Si se omite se asume que el nombre de la entidad es el nombre de la tabla. Su sintaxis es:

```
@Entity
@Table(name = “nomTabla”)
public class nomEntidad {
   ...
}
```

Donde nomTabla es el nombre de la tabla en la que se persistirán los objetos de la entidad.

@Column

Esta anotación permite mapear el nombre de un atributo o propiedad de una entidad a la columna de la tabla en la que se persistirá. Esta anotación es opcional. Si se omite se asume que el nombre del atributo o propiedad es el nombre de la columna. Su sintaxis es:

```
@Entity
@Table(name = “nomTabla”)
public class nomEntidad {
   @Column(name = “nomColumna”)
   modificadorAcceso tipo nomAtributo
   ...
}
```

@Id

Cada instancia de una identidad debe ser identificada unívocamente. La anotación `@Id` establece el campo o propiedad de una entidad como la identidad de una entidad. Su correspondiente columna en la tabla será la llave primaria de la tabla.

El atributo o propiedad anotado como la identidad de la entidad debe ser un tipo primitivo, una clase envolvente, un tipo serializable como `java.lang.String`, `java.util.Date`, y `java.sql.Date`. Adicionalmente no es recommendable usar como identidad un tipo flotante: float o double ni sus clases envolventes por el problema de precisión.

La sintaxis de la anotación `@Id` es:

```
@Entity
@Table(name = “nomTabla”)
public class nomEntidad {
   @Id
   @Column(name = “nomLlavePrimaria”)
   modificadorAcceso tipo nomAtributoIdentidad
   ...
}
```

El siguiente código ilustra las diferentes anotaciones vistas:

```
@Entity
@Table(name = "generos")
public class Genero implements Serializable {
  @Id
  @Basic(optional = false)
  @Column(name = "cveGenero")
  private String cveGenero;

  @Basic(optional = false)
  @Column(name = "nombre")
  private String nombre;

  @Basic(optional = false)
  @Column(name = "tipoMedio")
  private char tipoMedio;
```

En este ejemplo la anotación `@Basic(optional = false)` establece que la columna asociada no es opcional.

Las tablas correspondientes a las entidades anteriores son:

generos

```
cveGenero  not null, primary key varchar(10)
nombre not null,                 varchar(20)
tipoMedio  not null,             char
```

Anotaciones para la Relación Uno a Uno

En una relación uno a uno unidireccional, la entidad que tiene la referencia a la otra entidad se dice que es la **dueña de la relación**. En una relación uno a uno bidireccional una de las entidades se dice que es la dueña de la relación. La tabla correspondiente a la entidad dueña de la relación es la que tiene la llave foránea relaciondola a la otra tabla.

@OneToOne

Esta anotación se usa para marcar las relaciones uno a uno uni- y bidireccionales. Esta anotación se emplea en el atributo o propiedad que es la referencia a la otra entidad. Su sintaxis es:

```
@Entity
@Table(name = “nomTabla”)
public class nomEntidad {
   ...
   @OneToOne
   modificadorAcceso tipo nomReferencia;
   ...
}
```

@JoinColumn

La anotación `@JoinColumn` se emplea en el atributo o propiedad con la anotación `@OneToOne` en la entidad que es dueña de la relación y permite establecer el nombre de la llave foránea en una tabla y el nombre de la columna a la que hace referencia en la otra tabla con la que está relacionada. Esa columna debe ser una llave primaria o única. Su sintaxis es:

```
@Entity
@Table(name = “nomTabla”)
public class nomEntidad {
   ...
   @OneToOne
   @JoinColumn(name = “nomLlaveForanea”,
               referencedColumnName=”nomColumnaReferenciada”)
   modificadorAcceso tipo nomReferencia;
   ...
}
```

Por ejemplo, en las entidades siguientes se tiene una relación uno a uno unidireccional entre un cliente y su cuenta de usuario. La entidad `Cliente` es la dueña de la relación.

```
@Entity
@Table(name = “clientes”)
public class Cliente {
   @Id
   protected String clienteId;
   protected String nombre;
   protected String direccion

   @OneToOne
   @JoinColumn(name = “usuarioId”, referencedColumnName=”usuarioId”)
   protected Usuario usuario;

   ...
}
```

```
@entity
@Table(name = “usuarios”)
public class Usuario {
   @Id
   protected String usuarioId;
   protected String contrasena;
   protected String rol;

   ...
}
```

Las tablas correspondientes a las entidades anteriores son:

clientes

```
clienteId not null, primary key varchar(10)
nombre    not null,             varchar(35)
direccion not null,             varchar(35)
usuarioId not null,             varchar(10)
          foreing key (usuarios(usuarioId))
```

usuarios

```
usuarioId  not null, primary key varchar(10)
contrasena not null,             varchar(10)
rol        not null,             varchar(20)
```

Si la relación es bidireccional, también debemos agregarle la anotación `@OneToOne` al atributo que contiene la relación inversa en la entidad relacionada, sólo que en este caso la anotación tiene la siguiente sintaxis:

```
@Entity
@Table(name = “nomTabla”)
public class nomEntidad {
   ...
   @OneToOne(mappedBy = “nomReferencia”)
   modificadorAcceso tipo nomReferenciaInversa;
   ...
}
```

El parámetro `mappedBy` es el nombre de la referencia a esta entidad en la entidad dueña de la relación.

El siguiente código muestra el ejemplo anterior modificado para que la relación sea bidireccional.

```
@Entity
@Table(name = “clientes”)
public class Cliente {
   @Id
   protected String clienteId;
   protected String nombre;
   protected String direccion
```

```
   @OneToOne
   @JoinColumn(name = “usuarioId”, referencedColumnName=”usuarioId”)
   protected Usuario usuario;
}

@entity
@Table(name = “usuarios”)
public class Usuario {
   @Id
   protected String usuarioId;
   protected String contrasena;
   protected String rol;

   @OneToOne(mappedBy = “usuario”)
   protected Cliente cliente;
}
```

Las tablas correspondientes a las entidades anteriores son las mismas que las de las entidades con una relación unidireccional:

Anotaciones para la Relación Muchos a Uno y Uno a Muchos

En una relación muchos a uno o uno a muchos, la entidad que representa el lado “muchos” de la relación, es la entidad **dueña de la relación**. La tabla correspondiente a la entidad dueña de la relación es la que tiene la llave foránea relaciondola a la otra tabla.

@ManyToOne

Esta anotación se usa para marcar las relaciones muchos a uno y uno a muchos en la entidad que es dueña de la relación. Esta anotación se emplea en el atributo o propiedad que es la referencia a la otra entidad. Su sintaxis es:

```
@Entity
@Table(name = “nomTabla”)
public class nomEntidad {
   ...
   @ManyToOne
   modificadorAcceso tipo nomReferencia;
   ...
}
```

@JoinColumn

La anotación `@JoinColumn` se emplea en el atributo o propiedad con la anotación `@OneToOne` en la entidad que es dueña de la relación y permite establecer el nombre

de la llave foránea en una tabla y el nombre de la columna a la que hace referencia en la otra tabla con la que está relacionada. Esa columna debe ser una llave primaria o única. Su sintaxis es la misma que el la relación uno a uno.

Por ejemplo, en las entidades siguientes se tiene una relación muchos a uno unidireccional entre un medio y un género. La entidad `Medio` es la dueña de la relación.

```
@Entity
@Table(name = “medios”)
public class Medio {
   @Id
   protected String clave;
   protected String titulo;

   @ManyToOne
   @JoinColumn(name = “cveGenero”,
               referencedColumnNane = “cveGenero”)
   protected Genero genero;

   protected int duracion;
   protected Date fecha;
   ...
}

@Entity
@Table(name = “generos”)
public class Genero {
   @Id
   protected String cveGenero;
   protected String nombre;
   protected char tipoMedio;
}
```

Las tablas correspondientes a las entidades anteriores son:

medios

```
clave     not null, primary key varchar(10)
titulo    not null,             varchar(35)
cveGenero not null,             varchar(10)
          foreing key (generos(cveGenero))
duracion  not null,              integer
fecha                            Date
```

generos

```
cveGenero  not null, primary key varchar(10)
nombre not null,                 varchar(20)
tipoMedio  not null,             char
```

@OneToMany

Esta anotación se usa para marcar la relación uno a muchos (o muchos a uno bidireccional) en la entidad que no es la dueña de la relación. Esta anotación se emplea en el atributo o propiedad que es una referencia a a una colección (`Collection`, `List` o `Set`) de objetos de la otra entidad. Su sintaxis es:

```
@Entity
@Table(name = “nomTabla”)
public class nomEntidad {
   ...
   @OneToMany(mappedBy = “nomReferencia”)
   modificadorAcceso tipo nomReferenciaColección;
   ...
}
```

Por ejemplo, si la relación entre un medio y un género se hace bidireccional se tiene lo siguiente.

```
@Entity
@Table(name = “medios”)
public class Medio {
   @Id
   protected String clave;
   protected String titulo;

   @ManyToOne
   @JoinColumn(name = “cveGenero”,
               referencedColumnNane = “cveGenero”)
   protected Genero genero;

   protected int duracion;
   protected Date fecha;
   ...
}

@Entity
@Table(name = “generos”)
public class Genero {
   @Id
   protected String cveGenero;
   protected String nombre;
   protected char tipoMedio;

   @OneToMany(mappedBy = “genero”)
   List<Medio> listamedios;
   ...
}
```

Las tablas correspondientes a las entidades anteriores son las mismas que las de las entidades con una relación muchos a uno unidireccional.

Anotaciones para la Relación Muchos a Muchos

En una relación muchos a muchos, la entidad que es **dueña de la relación** es arbitraria.

@ManyToMany

Esta anotación se usa para marcar las relaciones muchos a muchos. Esta anotación se emplea en el atributo o propiedad que es la referencia a la otra entidad. Su sintaxis es:

```
@Entity
@Table(name = “nomTabla”)
public class nomEntidad {
   ...
   @ManyToMany
   modificadorAcceso tipo nomReferencia;
   ...
}
```

@JoinTable

La anotación `@JoinTable` se emplea en el atributo o propiedad con la anotación `@ManyToMany` en la entidad que es dueña de la relación y permite establecer el nombre de la tabla conjunta y sus columnas. Esas columnas están relacionadas con las llaves primarias de las tablas correspondientes a las entidades de la relación. Su sintaxis es:

```
@Entity
@Table(name = “nomTabla”)
public class nomEntidad {
   ...
   @ManyToMany
   @JoinTable(name = “nomTablaConjunta”,
      joinColumns = {
         @JoinColumn(name = “nomLlaveForanea”,
                     referencedColumnName = ”nomColumnaReferenciada”),
         ...
      }
      inverseJoinColumns = {
         @JoinColumn(name = “nomLlaveForanea”,
                     referencedColumnName = ”nomColumnaReferenciada”),
         ...
      })
   modificadorAcceso tipo nomReferencia;
   ...
}
```

En la entidad que no es dueña de la relación también se agregarle la anotación `@ManyToMany` al atributo que contiene la relación inversa en la entidad relacionada, sólo que en este caso la anotación tiene la siguiente sintaxis:

```
@Entity
@Table(name = “nomTabla”)
public class nomEntidad {
   ...
   @ManyToMany(mappedBy = “nomReferencia”)
   modificadorAcceso tipo nomReferenciaInversa;
   ...
}
```

Por ejemplo, el código para las entidades `Cancion` y `Album` con la relación muchos a muchos es la siguiente:

```
@Entity
@Table(name = “canciones”)
public class Cancion {
   @Id
   protected String clave;
   protected String titulo;
   protected Genero genero;
   protected int duracion;
   protected Date fecha;

   @ManyToMany
   @JoinTable(name = “canciones_albumes”,
      joinColumns = @JoinColumn(name = “clave”,
                                referencedColumnName = ”clave”),
      inverseJoinColumns = @JoinColumn(name = “cveAlbum”,
                                    referencedColumnName = ”cveAlbum))
   protected List<Album> listaAlbumes;
}

@Entity
@Table(name = “albumes”)
public class Album {
   @Id
   protected String cveAlbum;
   protected String nombre;


   @ManyToMany(mappedBy = “listaAlbumes”)
   protected List<Cancion> listaCanciones;
}
```

Las tablas corresponientes a las entidades anterios son las siguientes:

canciones

```
clave     not null, primary key varchar(10)
titulo    not null,             varchar(35)
cveGenero not null,             varchar(10)
          foreing key (generos(cveGenero))
duracion  not null,              integer
fecha                            Date
cveAlbum   not null,             varchar(10)
```

albumes

```
cveAlbum  not null, primary key varchar(10)
nombre not null,                 varchar(20)
clave  not null,                 varchar(10)
```

canciones_albumes

```
clave     not null, primary key varchar(10)
          foreing key (canciones(clave))
cveAlbum  not null, primary key varchar(10)
          foreing key (albumes(cveAlbum))
```

Anotaciones para la Relación de Herencia

@Inheritance

Esta anotación se utiliza para especificar el tipo de estrategia que se empleará para mapear la herencia. Se emplea en la entidad padre y su sintaxis es la siguiente:

```
@Entity
@Table(name = “nomTabla”)
@Inheritance(strategy = InheritanceType.tipoHerencia)
public class nomEntidadPadre {
   ...
}
```

Donde *tipoHerencia* tiene los siguientes valores:

- **SINGLE_TABLE** para el caso de la estrategia de una sola tabla
- **JOINED** para el caso de la estrategia de tablas conjuntas
- **TABLE_PER_CLASS** para el caso de la estrategia de una tabla por clase

@DiscriminatorColumn

Esta anotación se utiliza para establecer la columna descriminadora y su tipo en las estrategias de una sola tabla y de tablas conjuntas. Su sintaxis es:

```
@Entity
@Table(name = “nomTabla”)
@Inheritance(strategy = InheritanceType.tipoHerencia)
@DiscriminatorColumn(name = “nomColumnadescriminadora”,
     discriminatorType = DiscriminatorType.tipodescriminador,
     length = longitud)
public class nomEntidadPadre {
   ...
}
```

Donde *tipoDescriminador* tiene los siguientes valores: STRING, CHAR o INTEGER. El valor por ausencia es STRING.

Longitud es la longitud máxima que puede tener el valor del discriminador. Por ausencia, su valor es 31.

@DiscriminatorValue

Esta anotación se utiliza para establecer el valor de la columna descriminadora. Esta anotación se usa en la clase padre si no es abstracta y en las clases hijas Su sintaxis si se usa en la clase padre, es:

```
@Entity
@Table(name = “nomTabla”)
@Inheritance(strategy = InheritanceType.tipoHerencia)
@DiscriminatorColumn(name = “nomColumnadescriminadora”,
     discriminatorType = DiscriminatorType.tipodescriminador,
     length = longitud)
@DiscriminatorValue(value = “valor”)
public class nomEntidadPadre {
   ...
}
```

O, en caso de las clases hijas:

```
@Entity
@Table(name = “nomTabla”)
@DiscriminatorValue(value = “valor”)
public class nomEntidad extends nomEntidadPadre {
   ...
}
```

Donde *valor* tiene que coincidir con el tipo del descriminador. El valor por ausencia es el nombre de la entidad.

@PrimaryKeyJoinColumn

Esta anotación se utiliza para establecer el nombre de las llaves foráneas en las clases hijas cuando se usa la estrategia de tablas conjuntas. Esas llaves foráneas están relacionadas con la llave primaria en la tabla asociada a la entidad padre. Su sintaxis es:

```
@Entity
@Table(name = “nomTabla”)
@DiscriminatorValue(value = “valor”)
@PrimaryKeyJoinColumn(name = “nomLlaveForanea”)
public class nomEntidadPadre {
   ...
}
```

Ejemplo de Herencia Usando la Estrategia de una Sola Tabla

```
@Entity
@Table(name = “medios”)
@Inheritance(strategy = InheritanceType.SIGLED_TABLE)
@DiscriminatorColumn(name = “tipoMedio”,
     discriminatorType = DiscriminatorType.CHAR)
public abstract class Medio {
   @Id
   protected String clave;
   protected String titulo;
   @ManyToOne
   @JoinColumn(name = “cveGenero”,
               referencedColumnNane = “cveGenero”)
   protected Genero genero;
   protected int duracion;
   protected Date fecha;
   ...
}

@Entity
@Table(name = “canciones”)
@DiscriminatorValue(value = ‘C’)
public class Cancion extends Medio {
   private String interprete;
   private String autor;
   private d String album;
   ...
}

@Entity
@Table(name = “peliculas”)
@DiscriminatorValue(value = ‘P’)
public class Pelicula extends Medio {
   private String actor1;
   private String actor2;
   private String director;
   ...
}
```

La tabla en la que se almacenan las entidades anteriores son:

medios

```
clave     not null, primary key varchar(10)
titulo    not null,             varchar(35)
tipoMedio not null,             varchar(1)
cveGenero not null,             varchar(10)
          foreing key (generos(cveGenero))
duracion  not null,              integer
fecha                            Date
interprete                       varchar(35)
autor                            varchar(35)
album                            varchar(20)
actor1                           varchar(35)
```

```
actor2                          varchar(35)
director                        varchar(35)
```

Ejemplo de herencia Usando la Estrategia de Tablas Conjuntas

```
@Entity
@Table(name = “medios”)
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = “tipoMedio”,
     discriminatorType = DiscriminatorType.CHAR)
public abstract class Medio {
   @Id
   protected String clave;
   protected String titulo;
   @ManyToOne
   @JoinColumn(name = “cveGenero”,
               referencedColumnNane = “cveGenero”)
   protected Genero genero;
   protected int duracion;
   protected Date fecha;
   ...
}

@Entity
@Table(name = “canciones”)
@DiscriminatorValue(value = ‘C’)
@PrimaryKeyJoinColumn(name = “clave”)
public class Cancion extends Medio {
   private String interprete;
   private String autor;
   private d String album;
   ...
}

@Entity
@Table(name = “peliculas”)
@DiscriminatorValue(value = ‘P’)
@PrimaryKeyJoinColumn(name = “clave”)
public class Pelicula extends Medio {
   private String actor1;
   private String actor2;
   private String director;
   ...
}
```

Las tablas en la que se almacenan las entidades anteriores son:

medios

```
clave     not null, primary key varchar(10)
titulo    not null,             varchar(35)
tipoMedio not null,             varchar(20)
cveGenero not null,             varchar(10)
          foreing key (generos(cveGenero))
duracion  not null,              integer
```

```
fecha                          Date
```

canciones

```
clave     not null, primary key varchar(10)
          foreing key (medios(clave))
interprete                     varchar(35)
autor                          varchar(35)
album                          varchar(20)
```

peliculas

```
clave     not null, primary key varchar(10)
          foreing key (medios(clave))
actor1                         varchar(35)
actor2                         varchar(35)
director                       varchar(35)
```

Ejemplo de Herencia Usando la Estrategia de una Tabla por Clase

```
@Entity
@Table(name = “medios”)
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Medio {
   @Id
   protected String clave;
   protected String titulo;
   @ManyToOne
   @JoinColumn(name = “cveGenero”,
               referencedColumnNane = “cveGenero”)
   protected Genero genero;
   protected int duracion;
   protected Date fecha;
   ...
}

@Entity
@Table(name = “canciones”)
public class Cancion extends Medio {
   private String interprete;
   private String autor;
   private d String album;
   ...
}

@Entity
@Table(name = “peliculas”)
public class Pelicula extends Medio {
   private String actor1;
   private String actor2;
   private String director;
   ...
}
```

La tabla en la que se almacenan las entidades anteriores son:

medios

```
clave     not null, primary key varchar(10)
titulo    not null,             varchar(35)
cveGenero not null,             varchar(10)
          foreing key (generos(cveGenero))
duracion  not null,              integer
fecha                            Date
```

canciones

```
clave     not null, primary key varchar(10)
titulo    not null,             varchar(35)
cveGenero not null,             varchar(10)
          foreing key (generos(cveGenero))
duracion  not null,              integer
fecha                            Date
interprete                       varchar(35)
autor                            varchar(35)
album                            varchar(20)
```

peliculas

```
clave     not null, primary key varchar(10)
titulo    not null,             varchar(35)
cveGenero not null,             varchar(10)
          foreing key (generos(cveGenero))
duracion  not null,              integer
fecha                            Date
actor1                           varchar(35)
actor2                           varchar(35)
director                         varchar(35)
```

Ejemplo sobre Mapeo Objeto – Relacional

Como un ejemplo sobre mapeo Objeto – Relacional considere el conjunto de entidades cuyo diagrama de clases se muestra en la figura 16.1 y que va a ser persistido en la base de datos musicaJPA cuyo diagrama Entidad – relación se muestra en la figura 16.2.

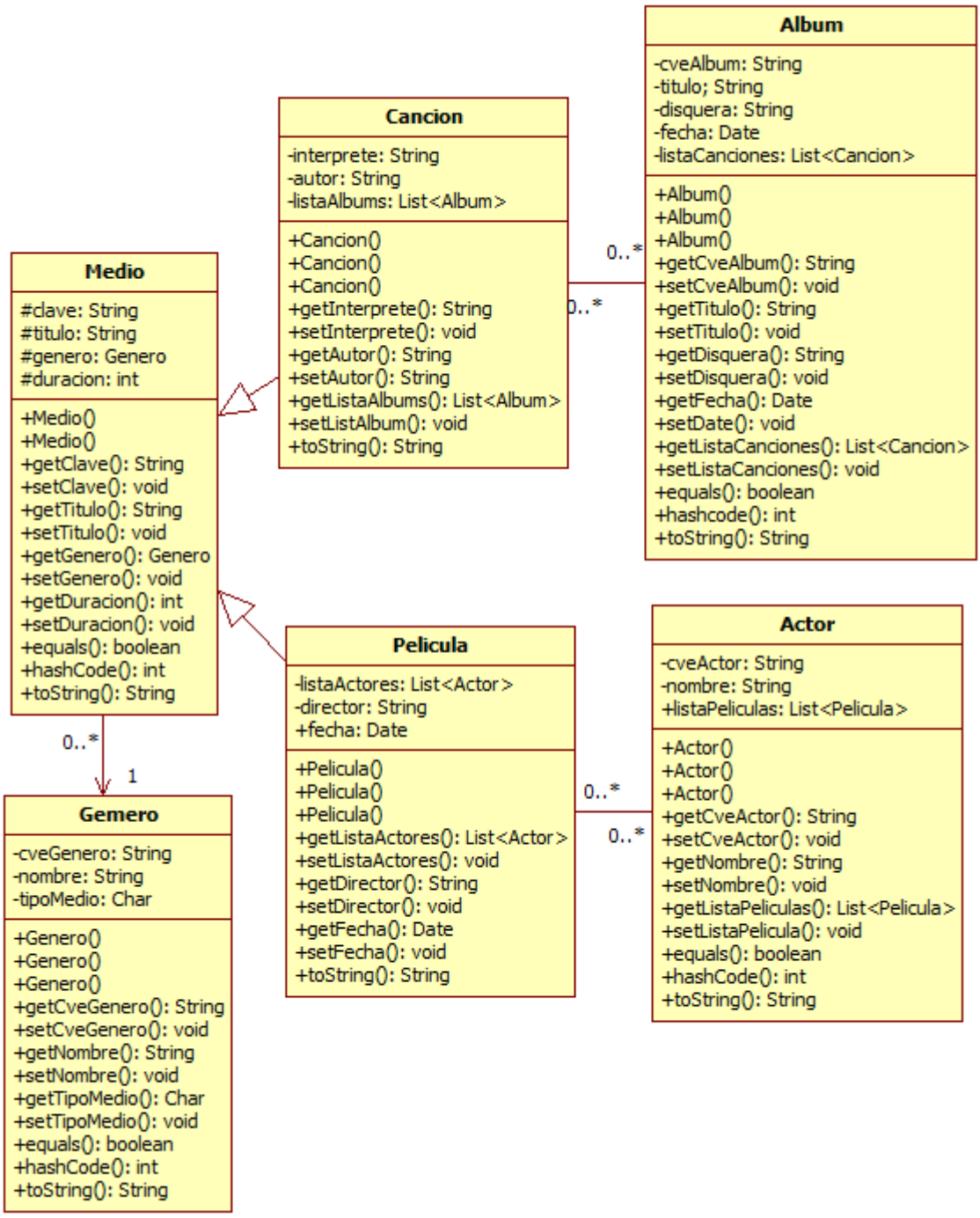
Album
-cveAlbum: String
-titulo; String
-disquera: String
-fecha: Date
-listaCanciones: List<Cancion>
+Album()
+Album()
+Album()
+getCveAlbum(): String
+setCveAlbum(): void
+getTitulo(): String
+setTitulo(): void
+getDisquera(): String
+setDisquera(): void
+getFecha(): Date
+setDate(): void
+getListaCanciones(): List<Cancion>
+setListaCanciones(): void
+equals(): boolean
+hashcode(): int
+toString(): String
Cancion
-interprete: String
-autor: String
-listaAlbums: List<Album>
+Cancion()
+Cancion()
+Cancion()
+getInterprete(): String
+setInterprete(): void
+getAutor(): String
+setAutor(): String
+getListaAlbums(): List<Album>
+setListAlbum(): void
+toString(): String
0..*
0..*
Medio
#clave: String
#titulo: String
#genero: Genero
#duracion: int
+Medio()
+Medio()
+getClave(): String
+setClave(): void
+getTitulo(): String
+setTitulo(): void
+getGenero(): Genero
+setGenero(): void
+getDuracion(): int
+setDuracion(): void
+equals(): boolean
+hashCode(): int
+toString(): String
0..*
1
Pelicula
-listaActores: List<Actor>
-director: String
+fecha: Date
+Pelicula()
+Pelicula()
+Pelicula()
+getListaActores(): List<Actor>
+setListaActores(): void
+getDirector(): String
+setDirector(): void
+getFecha(): Date
+setFecha(): void
+toString(): String
0..*
0..*
Actor
-cveActor: String
-nombre: String
+listaPeliculas: List<Pelicula>
+Actor()
+Actor()
+Actor()
+getCveActor(): String
+setCveActor(): void
+getNombre(): String
+setNombre(): void
+getListaPeliculas(): List<Pelicula>
+setListaPelicula(): void
+equals(): boolean
+hashCode(): int
+toString(): String
Gemero
-cveGenero: String
-nombre: String
-tipoMedio: Char
+Genero()
+Genero()
+Genero()
+getCveGenero(): String
+setCveGenero(): void
+getNombre(): String
+setNombre(): void
+getTipoMedio(): Char
+setTipoMedio(): void
+equals(): boolean
+hashCode(): int
+toString(): String

Figura 16.1

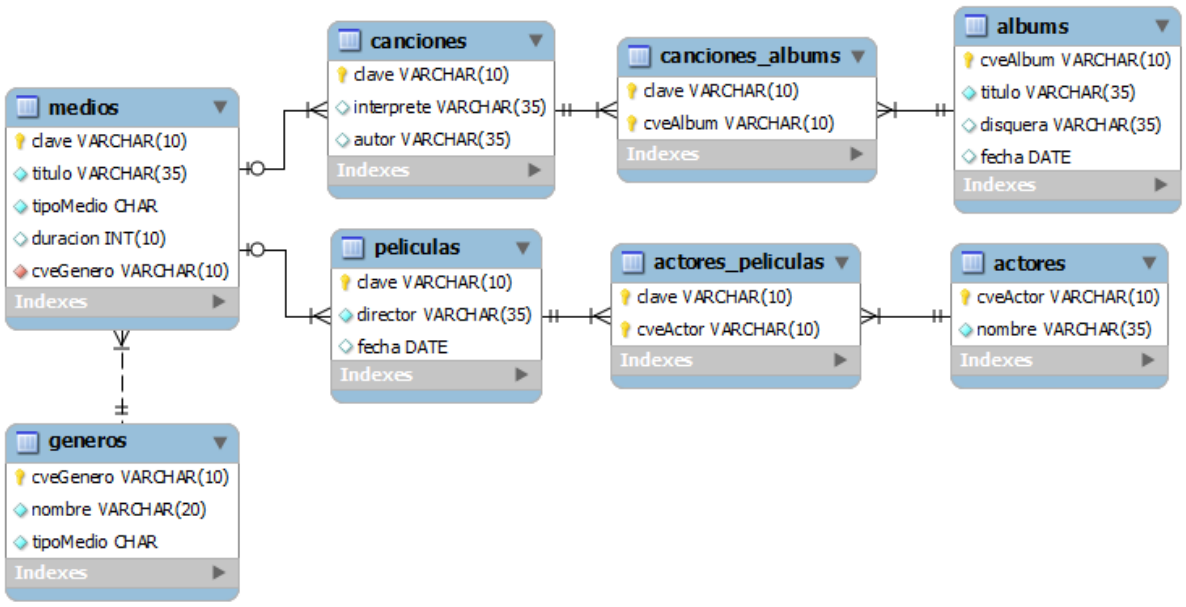

Figura 16.2

El siguiente código parcial de la entidad `Genero`, muestra las anotaciones de la entidad y sus atributos. No se muestran los mérodos de acceso `getXxx()`, `setXxx()`, `equals()` ni `hashCode()`:

Genero.java

```
/*
 * Genero.java.
 * @author mdomitsu
 */
package objetosNegocio;

import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

@Entity
@Table(name = "generos")

@NamedQueries({
    @NamedQuery(name = "Genero.findAll", query = "SELECT g FROM Genero g"),
    @NamedQuery(name = "Genero.findByCveGenero",
            query = "SELECT g FROM Genero g WHERE g.cveGenero = :cveGenero"),
    @NamedQuery(name = "Genero.findByNombre",
                query = "SELECT g FROM Genero g WHERE g.nombre = :nombre"),
    @NamedQuery(name = "Genero.findByTipoMedio",
           query = "SELECT g FROM Genero g WHERE g.tipoMedio = :tipoMedio")})
```

```
public class Genero implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @Basic(optional = false)
    @Column(name = "cveGenero")
    private String cveGenero;

    @Basic(optional = false)
    @Column(name = "nombre")
    private String nombre;

    @Basic(optional = false)
    @Column(name = "tipoMedio")
    private char tipoMedio;

    public Genero() {
    }

    public Genero(String cveGenero) {
        this.cveGenero = cveGenero;
    }

    public Genero(String cveGenero, String nombre, char tipoMedio) {
        this.cveGenero = cveGenero;
        this.nombre = nombre;
        this.tipoMedio = tipoMedio;
    }

    ...

    @Override
    public String toString() {
        return cveGenero + ", " + nombre + ", " + tipoMedio;
    }
}
```

El siguiente código parcial de la entidad `Medio`, muestra las anotaciones de la entidad y sus atributos. No se muestran los mérodos de acceso `getXxx()`, `setXxx()`, `equals()` ni `hashCode()`:

Medio.java

```
/*
 * Medio.java.
 * @author mdomitsu
 */
package objetosNegocio;

import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.DiscriminatorColumn;
import javax.persistence.DiscriminatorType;
import javax.persistence.Entity;
```

```
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

@Entity
@Table(name = "medios")
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "tipoMedio", discriminatorType =
DiscriminatorType.CHAR)

@NamedQueries({
    @NamedQuery(name = "Medio.findAll", query = "SELECT m FROM Medio m"),
    @NamedQuery(name = "Medio.findByClave",
                query = "SELECT m FROM Medio m WHERE m.clave = :clave"),
    @NamedQuery(name = "Medio.findByTitulo",
                   query = "SELECT m FROM Medio m WHERE m.titulo = :titulo"),
    @NamedQuery(name = "Medio.findByDuracion",
              query = "SELECT m FROM Medio m WHERE m.duracion = :duracion")})

public abstract class Medio implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @Basic(optional = false)
    @Column(name = "clave")
    protected String clave;

    @Basic(optional = false)
    @Column(name = "titulo")
    protected String titulo;

    @Column(name = "duracion")
    protected Integer duracion;

    @ManyToOne(optional = false)
    @JoinColumn(name = "cveGenero", referencedColumnName = "cveGenero")
    protected Genero genero;

    public Medio() {
    }

    public Medio(String clave) {
        this.clave = clave;
    }

    public Medio(String clave, String titulo, int duracion, Genero genero) {
        this.clave = clave;
        this.titulo = titulo;
        this.duracion = duracion;
        this.genero = genero;
    }
```

```
   ...

    @Override
    public String toString() {
        return clave + ", " + titulo + ", " + genero.getNombre();
    }
}
```

El siguiente código parcial de la entidad `Cancion`, muestra las anotaciones de la entidad y sus atributos. No se muestran los mérodos de acceso `getXxx()`ni `setXxx()`:

Cancion.java

```
/*
 * Cancion.
 * @author mdomitsu
 */

package objetosNegocio;

import java.io.Serializable;
import java.util.List;
import javax.persistence.Column;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;

@Entity
@Table(name = "canciones")
@DiscriminatorValue(value = "C")
@PrimaryKeyJoinColumn(name = "clave")

@NamedQueries({
    @NamedQuery(name = "Cancion.findAll", query = "SELECT c FROM Cancion c"),
    @NamedQuery(name = "Cancion.findByClave",
                query = "SELECT c FROM Cancion c WHERE c.clave = :clave"),
    @NamedQuery(name = "Cancion.findByInterprete",
         query = "SELECT c FROM Cancion c WHERE c.interprete = :interprete"),
    @NamedQuery(name = "Cancion.findByAutor",
                query = "SELECT c FROM Cancion c WHERE c.autor = :autor")})

public class Cancion extends Medio implements Serializable {
    private static final long serialVersionUID = 1L;

    @Column(name = "interprete")
    private String interprete;

    @Column(name = "autor")
    private String autor;
```

```
    @ManyToMany
    @JoinTable(name = "canciones_albums",
               joinColumns = {@JoinColumn(name = "clave",
                                          referencedColumnName = "clave")},
               inverseJoinColumns = {@JoinColumn(name = "cveAlbum",
                                         referencedColumnName = "cveAlbum")})
    private List<Album> listaAlbums;

    public Cancion() {
    }

    public Cancion(String clave) {
        super(clave);
    }

    public Cancion(String clave, String titulo, int duracion, Genero genero,
                   String interprete, String autor) {
        super(clave, titulo, duracion, genero);
        this.interprete = interprete;
        this.autor = autor;
    }

    ...

    public void agregaAlbum(Album album) {
        if(listaAlbums == null) {
            listaAlbums = new ArrayList<Album>();
        }
        listaAlbums.add(album);
    }

    public void eliminaAlbum(Album album) {
        listaAlbums.remove(album);
    }

    @Override
    public String toString() {
        return super.toString();
    }
}
```

El siguiente código parcial de la entidad `Pelicula`, muestra las anotaciones de la entidad y sus atributos. No se muestran los mérodos de acceso `getXxx()`ni `setXxx()`:

Pelicula.java

```
/*
 * Pelicula.java.
 * @author mdomitsu
 */
package objetosNegocio;

import java.io.Serializable;
import java.util.Date;
```

```
import java.util.List;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table(name = "peliculas")
@DiscriminatorValue(value = "P")
@PrimaryKeyJoinColumn(name = "clave")

@NamedQueries({
    @NamedQuery(name = "Pelicula.findAll",
                query = "SELECT p FROM Pelicula p"),
    @NamedQuery(name = "Pelicula.findByClave",
                  query = "SELECT p FROM Pelicula p WHERE p.clave = :clave"),
    @NamedQuery(name = "Pelicula.findByDirector",
            query = "SELECT p FROM Pelicula p WHERE p.director = :director"),
    @NamedQuery(name = "Pelicula.findByFecha",
                query = "SELECT p FROM Pelicula p WHERE p.fecha = :fecha")})

public class Pelicula extends Medio implements Serializable {
    private static final long serialVersionUID = 1L;

    @Basic(optional = false)
    @Column(name = "director")
    private String director;

    @Column(name = "fecha")
    @Temporal(TemporalType.DATE)
    private Date fecha;

    @ManyToMany
    @JoinTable(name = "actores_peliculas",
               joinColumns = {@JoinColumn(name = "clave",
                                          referencedColumnName = "clave")},
               inverseJoinColumns = {@JoinColumn(name = "cveActor",
                                                 referencedColumnName = "cveActor")})
    private List<Actor> listaActores;

    public Pelicula() {
    }

    public Pelicula(String clave) {
        super(clave);
    }

    public Pelicula(String clave, String titulo, int duracion, Genero genero,
```

```
                    String director, Date fecha) {
        super(clave, titulo, duracion, genero);
        this.director = director;
        this.fecha = fecha;
    }

   ...

    public void agregaActor(Actor actor) {
        if(listaActores == null) {
            listaActores = new ArrayList<Actor>();
        }
        listaActores.add(actor);
    }

    public void eliminaActor(Actor actor) {
        listaActores.remove(actor);
    }

    @Override
    public String toString() {
        return super.toString() + ", " + director;
    }
}
```

El siguiente código parcial de la entidad `Album`, muestra las anotaciones de la entidad y sus atributos. No se muestran los mérodos de acceso `getXxx()`, `setXxx()`, `equals()` ni `hashCode()`:

Album.java

```
/*
 * Album.java.
 * @author mdomitsu
 */
package objetosNegocio;

import java.io.Serializable;
import java.util.Date;
import java.util.List;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table(name = "albums")

@NamedQueries({
    @NamedQuery(name = "Album.findAll", query = "SELECT a FROM Album a"),
```

```
    @NamedQuery(name = "Album.findByCveAlbum",
                query = "SELECT a FROM Album a WHERE a.cveAlbum = :cveAlbum"),
    @NamedQuery(name = "Album.findByTitulo",
                 query = "SELECT a FROM Album a WHERE a.titulo = :titulo"),
    @NamedQuery(name = "Album.findByDisquera",
                query = "SELECT a FROM Album a WHERE a.disquera = :disquera"),
    @NamedQuery(name = "Album.findByFecha",
                 query = "SELECT a FROM Album a WHERE a.fecha = :fecha")})

public class Album implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @Basic(optional = false)
    @Column(name = "cveAlbum")
    private String cveAlbum;

    @Basic(optional = false)
    @Column(name = "titulo")
    private String titulo;

    @Column(name = "disquera")
    private String disquera;

    @Column(name = "fecha")
    @Temporal(TemporalType.DATE)
    private Date fecha;

    @ManyToMany(mappedBy = "listaAlbums")
    private List<Cancion> listaCanciones;

    public Album() {
    }

    public Album(String cveAlbum) {
        this.cveAlbum = cveAlbum;
    }

    public Album(String cveAlbum, String titulo, String disquera,
                 Date fecha) {
        this.cveAlbum = cveAlbum;
        this.titulo = titulo;
        this.disquera = disquera;
        this.fecha = fecha;
    }

    ...

    public void agregaCancion(Cancion cancion) {
        if(listaCanciones == null) {
            listaCanciones = new ArrayList<Cancion>();
        }
        listaCanciones.add(cancion);
    }

    public void eliminaCancion(Cancion cancion) {
        listaCanciones.remove(cancion);
```

```
    }

    @Override
    public String toString() {
        return cveAlbum + ", " + titulo;
    }
}
```

El siguiente código parcial de la entidad `Actor`, muestra las anotaciones de la entidad y sus atributos. No se muestran los mérodos de acceso `getXxx()`, `setXxx()`, `equals()` ni `hashCode()`:

Actor.java

```
/*
 * Actor.java
 * @author mdomitsu
 */
package objetosNegocio;

import java.io.Serializable;
import java.util.List;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

@Entity
@Table(name = "actores")

@NamedQueries({
    @NamedQuery(name = "Actor.findAll", query = "SELECT a FROM Actor a"),
    @NamedQuery(name = "Actor.findByCveActor",
                query = "SELECT a FROM Actor a WHERE a.cveActor = :cveActor"),
    @NamedQuery(name = "Actor.findByNombre",
                 query = "SELECT a FROM Actor a WHERE a.nombre = :nombre")})

public class Actor implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @Basic(optional = false)
    @Column(name = "cveActor")
    private String cveActor;

    @Basic(optional = false)
    @Column(name = "nombre")
    private String nombre;

    @ManyToMany(mappedBy = "listaActores")
    private List<Pelicula> listaPeliculas;
```

```
    public Actor() {
    }

    public Actor(String cveActor) {
        this.cveActor = cveActor;
    }

    public Actor(String cveActor, String nombre) {
        this.cveActor = cveActor;
        this.nombre = nombre;
    }

    ...

    public void agregaPelicula(Pelicula pelicula) {
        if(listaPeliculas == null) {
            listaPeliculas = new ArrayList<Pelicula>();
        }
        listaPeliculas.add(pelicula);
    }

    public void eliminaPelicula(Pelicula pelicula) {
        listaPeliculas.remove(pelicula);
    }

   @Override
    public String toString() {
        return cveActor + ", " + nombre;
    }
}
```

Interface used to interact with the persistence context.

An `EntityManager` instance is associated with a persistence context. A persistence context is a set of entity instances in which for any persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their lifecycle are managed. The `EntityManager` API is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities.

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

El Administrador de Entidades

El **Administrador de Entidades** de la JPA es la componente de software, la API de JPA, que persiste las entidades en una base de datos. Para ello el administrador de entidades utiliza la metainformación provista por las anotaciones que tiene una entidad.

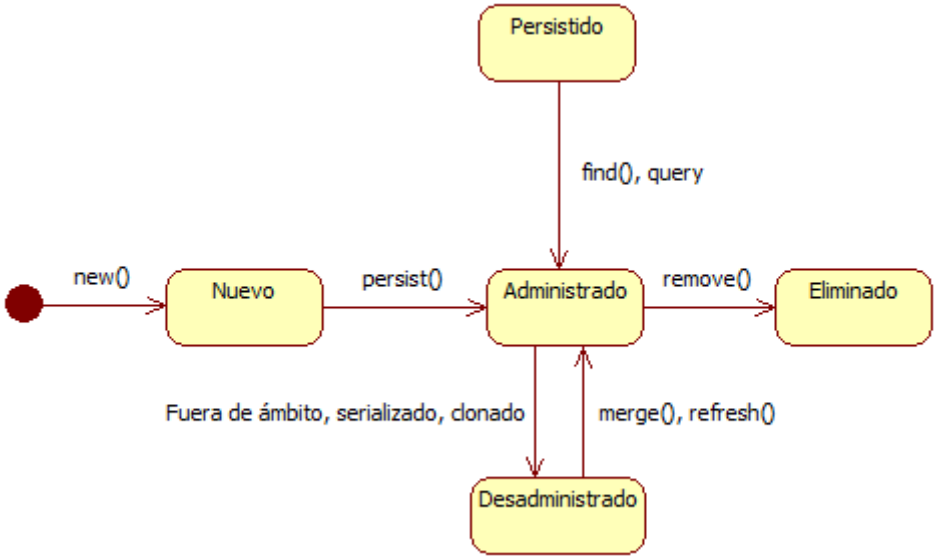

Figura 16.3

Las clases e interfaces principales de la API de JPA se muestran en el diagrama de clases de la figura 16.3. Esa API está implementada por el administrador de entidades y en su mayor parte está encapsulada por la interfaz `EntityManager`.

Un proveedor de persistencia (por ejemplo **Hibernate** o **Toplink**) implementa la máquina que repalda a toda la API de persistencia, desde el `EntityManager`, el `Query` y la generación de SQL.

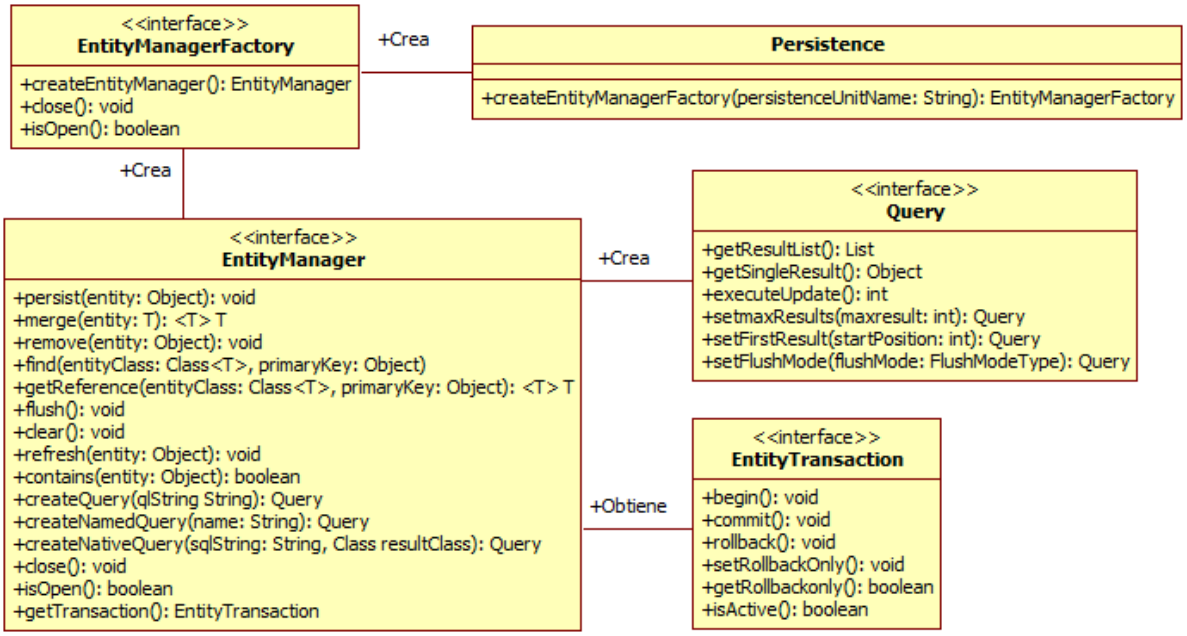

Figura 16.4

Para que una aplicación pueda persistir una o más entidades, se requiere crear uno o más administradores de entidad. Todos los administradores de entidades son creados invocando al método `createEntityManager()` de una instancia de `EntityManagerFactory`, que es una fábrica de entidades. La configuración administrador de entidades

Clase `Persistencia`

Esta clase nos permite crear una fábrica de administradores de entidad en ambientes de ejecución en los que no se tiene un contenedor de java EE, como es el caso de una aplicación de escritorio.

Tabla 16.1 Clase Persistance

```
public static EntityManagerFactory createEntityManagerFactory(
                                                    String persistenceUnitName)
```

Crea y regresa una fábrica de administradores de entidad para la unidad de persistencia del parámetro.

Parámetros:

`persistenceUnitName`: Nombre de la unidad de persistencia.

Regresa:

La fábrica que crea administradores de entidad configurados de acuerdo a la unidad de persistencia dada.

Interfaz `EntityManagerFactory`

Esta interfaz nos permite interactuar con la fábrica de administradores de entidad para la unidad de persistencia.

Una vez que la aplicación termina de usar la fábrica de administradores de entidad, la aplicación debe cerrarla. Cuando se cierra la fábrica, todos sus administradores de entidades están en el estado cerrado. Los métodos de esta interfaz están en la tabla 16.2.

Tabla 16.2 Interfaz EntityManagerFactory

`EntityManager` **`createEntityManager`**`()` Este método crea un administrador de entidades administrado por la aplicación. **Regresa:** Una instancia de un administrador de persistencia. **Lanza**: `IllegalStateException` - Si la fábrica de administradores de entidad se ha cerrado.
`void` **`close`**`()` Cierra la fábrica de administradores de entidad liberando los recursos ocupados. Cuando se cierra la fábrica, todos sus administradores de entidades están en el estado cerrado. **Lanza**: `IllegalStateException` - Si la fábrica de administradores de entidad se ha cerrado.
`boolean` **`isOpen`**`()` Regresa verdadera hasta que la fábrica de administradores de entidad se cierre. **Regresa:** Verdadero si la fábrica está abierta, falso en caso contrario.

Interfaz EntityManager

Esta interfaz nos permite interactuar con el contexto de persistencia. Los métodos de esta interfaz, a excepción de los que nos permiten crear consultas, están en la tabla 16.3.

Tabla 16.3 Interfaz EntityManager

`void` **`persist`**`(Object` *`entity`*`)` Hace que una instancia de una entidad se vuelva administrada y persistente. **Parámetros:** *`entity`*: Instancia de una entidad a volverse administrada y persistente.. **Lanza**: `EntityExistsException` – Si la entidad ya existe.

`IllegalArgumentException` – Si el parámetro no es una entidad.

`<T> T` **`merge`**`(T` *`entity`*`)`

Combina el estado de la entidad del parámetro con el contexto de persistencia actual.

Parámetros:

`entity`: Instancia de una entidad a combinar con el contexto de persistencia actual.

Regresa:

La instancia administrada a la que se le combinó el estado de la entidad del parámeto.

Lanza:

`IllegalArgumentException` – Si el parámetro no es una entidad o ya ha sido eliminada.

`void` **`remove`**`(Object entity)`

Elimina la entidad del parámetro.

Parámetros:

`entity`: Instancia de una entidad a eliminarse.

Lanza:

`IllegalArgumentException` – Si el parámetro no es una entidad o ya no se encuentra administrada.

`<T> T` **`find`**`(Class<T> entityClass, Object primaryKey)`

Encuentra y regresa una entidad de la clase y llave primaria dados por sus parámetros. Si la instancia de la entidad está en el contexto de persistencia, se regresa de allí.

Parámetros:

`entityClass`: Clase a la que pertenece la Instancia.
`primaryKey`: Valor de la llave primaria.

Regresa:

La instancia de la entidad hallada o null si la entidad no existe.

Lanza:

`IllegalArgumentException` – Si el primer argumento no denota un tipo de entidad o si el segundo argumento no es un tipo válido para la llave primaria de la entidad o es null.

`<T> T` **`getReference`**`(Class<T> entityClass, Object primaryKey)`

Obtiene una instancia cuyo estado puede obtenerse de modo “lazy”. La aplicación no puede suponer que el estado de la instancia se encuentre no administrada a menos que haya sido accesado por la aplicación mientras el administrador de entidades estuvo abierto.

Parámetros:

`entityClass`: Clase a la que pertenece la Instancia.
`primaryKey`: Valor de la llave primaria.

Regresa:

La instancia de la entidad hallada.

Lanza:

`IllegalArgumentException` – Si el primer argumento no denota un tipo de entidad o si el segundo argumento no es un tipo válido para la llave primaria de la entidad o es null.

`EntityNotFoundException` – Si el estado de la entidad no puede accederse.
`void` **`flush`**`()` Sincroniza el contexto de persistencia con su base de datos relacionado. **Lanza**: `TransactionRequiredException` – Si no hay una transacción. `PersistenceException` – Si el sincronizado falla.
`void` **`refresh`**`(Object entity)` Refresca el estado de una instancia de una base de datos, sobreescribiendo los cambios hechos a la entidad, si los hay. **Parámetros:** *`entity`*: Instancia de la entidad. **Lanza**: `IllegalArgumentException` – Si la instancia no es una entidad o la entidad no está administrada. `EntityNotFoundException` – Si la entidad no existe en la base de datos.
`void` **`clear`**`()` Limpia todo el context de la persistencia, hacienda que las entidades administradas se conviertan a no aministradas. Los cambios hechos a las entidades y que no se hayan vaciado a la base de datos no serán persistidos.
`void` **`detach`**`(Object` *`entity`*`)` Elimina la entidad dada por el parámetro del contexto de persistencia, haciendo que la entidad administrada se convierta a no administrada. Los cambios hechos a las entidades y que no se hayan vaciado a la base de datos no serán persistidos. **Parámetros:** *`entity`*: Instancia de la entidad. **Lanza**: `IllegalArgumentException` – Si la instancia no es una entidad.
`boolean` **`contains`**`(Object entity)` Verifica si la instancia es una instancia de una entidad administrada que pertenece al contexto de persistencia actual. **Parámetros:** *`entity`*: Instancia de la entidad. **Regresa:** Verdadero si la instancia de la entidad está en el contexto de la persistencia. **Lanza**: `IllegalArgumentException` – Si la instancia no es una entidad.
`void` **`close`**`()` Cierra el administrador de persistencia, Después de invocar al método close() todos los métodos del administrador de las entidades lanzarán una excepción del tipo `IllegalStateException`.`
`boolean` **`isOpen`**`()`

Determina si el administrador de entidades está abierto. **Regresa:** Verdadero hasta que el administrador de entidades se cierre.
`EntityTransaction` **`getTransaction`**`()` Regresa un objeto del tipo `EntityTransaction`. La instancia de `EntityTransaction` se puede usar para iniciar y comprometer transacciones múltiples. **Regresa:** Una instancia de `EntityTransaction`.

Interfaz `EntityTransaction`

Esta interfaz nos permite controlar las transacciones en administradores de entidad con recursos locales. Los métodos de esta interfaz están en la tabla 16.4.

Tabla 16.4 Interfaz `EntityTransaction`

`void` **`begin`**`()` Inicia la transacción de un recurso. **Lanza**: `IllegalStateException` - Si `isActive()` regresa verdadero.
`void` **`commit`**`()` Compromete (termina) la transacción del recurso actual, escribiendo los cambios que no se hayan hecho a la base de datos. **Lanza**: `IllegalStateException` - Si `isActive()` regresa falso. `RollBackException` – Si la operación no puede comprometerse.
`void` **`rollback`**`()` Regresa al estado original la transacción del recurso actual. **Lanza**: `IllegalStateException` - Si `isActive()` regresa falso. `PersistenceException` – Si ocurre una condición de error inesperada.
`isActive()` Indica si una transacción de un recurso se encuentra en proceso. **Regresa:** Verdadero si una transacción de un recurso se encuentra en proceso, falso en caso contrario. **Lanza**: `PersistenceException` – Si ocurre una condición de error inesperada.

La API de Consultas de JPA

La API de Consultas de JPA nos permite escribir consultas para obtener una entidad simple o una colección de entidades. Esas consultas pueden escribirse usando el Lenguaje de Consultas de la Persistencia de Java, JPQL o el lenguaje SQL. Al usar JPQL obtenemos como resultado un objeto o una colección de objetos, mientras que si usamos el lenguaje SQL los resutados de la consulta son registros de la BD que deben ser convertidos a objetos.

La API de Consultas de JPA está formada por:

- Los métodos de la interfaz `EntityManager` que nos permiten crear consultas.
- Los métodos de la interfaz `Query` que nos permiten definir y ejecutar las consultas.
- El lenguaje JPQL

La API de Consultas nos permite crear dos tipos de consultas: **Consultas con Nombre** o **Estáticas** y **Consultas Dinámicas**. Las consultas con nombre son para almacenarse y reutilizarse. Son consultas que se van a realizarse desde varios módulos de la aplicación y no queremos que se construya la consulta cada vez que se haga. Sus ventajas son:

- Mejoran la reutilización de las consultas.
- Mejoran la mantenibilidad del código, las consultas se pueden concentrar en ciertas partes del código en lugar de estar dispersas en todo el código de la lógica del negocio.
- Aumentan el desempeño ya que se preparan una sóla vez y pueden reusarse eficientemente.

Por otro lado, las consultas dinámicas se emplean cuando la consulta depende de las entradas del usuario o alguna condición en la lógica de la aplicación. Tanto las consultas con nombre como las consultas dinámicas pueden escribirse usando el lenguaje JPQL o el lenguaje SQL. A las consultas escritas en el lenguaje SQL se les llama consultas nativas.

Antes de crear y ejecutar una consulta con nombre debemos definirla. En el caso de una consulta dinámica la definición se hace en el momento de crearla.

Definición de Consultas con Nombre

Para definir una consulta con nombre se usa la anotación `@NamedQuery` en una entidad. Su sintaxis es:

```
@Entity
@Table(name = “nombreTabla”)
@NamedQuery(name = “nombreConsulta”, query = “consulta”)
```

```
Public class NomEntidad implements Serializable {
   ...
}
```

Por ejemplo, para la entidad `Genero` podemos definir la siguiente consulta con nombre:

```
@Entity
@Table(name = "generos")
@NamedQuery(name = "Genero.findAll", query = "SELECT g FROM Genero g")
public class Genero implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @Basic(optional = false)
    @Column(name = "cveGenero")
    private String cveGenero;

    @Basic(optional = false)
    @Column(name = "nombre")
    private String nombre;

    @Basic(optional = false)
    @Column(name = "tipoMedio")
    private char tipoMedio;

    ...
}
```

Podemos agrupar varias definiciones usando la anotación `@NamedQueries` en una entidad. Su sintaxis es:

```
@Entity
@NamedQueries({
      @NamedQuery(name = “nombreConsulta1”, query = “consulta1”)[,
      @NamedQuery(name = “nombreConsulta2”, query = “consulta2”)]...
})
@Table(name = “nombreTabla”)

Public class NomEntidad implements Serializable {
   ...
}
```

Por ejemplo, para la entidad `Genero` podemos definir las siguientes consultas con nombre:

```
@Entity
@Table(name = "generos")
@NamedQueries({
    @NamedQuery(name = "Genero.findAll",
                query = "SELECT g FROM Genero g"),
    @NamedQuery(name = "Genero.findByCveGenero",
                query = "SELECT g FROM Genero g
                WHERE g.cveGenero = :cveGenero"),
    @NamedQuery(name = "Genero.findByNombre",
```

```
                    query = "SELECT g FROM Genero g
                    WHERE g.nombre = :nombre"),
        @NamedQuery(name = "Genero.findByTipoMedio",
                    query = "SELECT g FROM Genero g
                    WHERE g.tipoMedio = :tipoMedio")})
    public class Genero implements Serializable {
        private static final long serialVersionUID = 1L;

        @Id
        @Basic(optional = false)
        @Column(name = "cveGenero")
        private String cveGenero;

        @Basic(optional = false)
        @Column(name = "nombre")
        private String nombre;

        @Basic(optional = false)
        @Column(name = "tipoMedio")
        private char tipoMedio;

        ...
}
```

Las consultas con nombre tienen ámbito de unidad de persistencia y por lo tanto tienen nombres únicos.

Creación de Consultas

La interfaz `EntityManager` contiene varios métodos que nos permiten crear consultas tanto con nombre o dinámicas. Estas últimas pueden usar el lenguaje JPQL o el lenguaje SQL. Los métodos de la interfaz `EntityManager` que nos permiten crear consultas se muestran en la tabla 16.5:

Tabla 16.5 Interfaz EntityManager

`Query` **`createNamedQuery`**`(String name)` Crea una consulta con nombre en el lenguaje de consultas de la persistencia de Java, JPQL o en el SQL nativo. **Parámetros:** *`name`*: El nombre de la consulta definido en metadata. **Regresa:** La consulta con nombre creada. Una instancia del tipo `Query`. **Lanza**: `IllegalArgumentException` – Si la cadena de consulta en JPQL mo es válida. O no hay una consulta con ese nombre.
`Query` **`createQuery`**`(String` *`qlString`*`)` Crea una consulta dinámica en el lenguaje de consultas de la persistencia de Java, JPQL.

Parámetros:
qlString: Una cadena de consulta en JPQL.

Regresa:
La consulta dinámica creada. Una instancia del tipo `Query`.

Lanza:
`IllegalArgumentException` – Si la cadena de consulta en JPQL no es válida.

`Query` **`createNativeQuery`**`(String` *`sqlString`*`)`

Crea una consulta dinámica en el lenguaje SQL para actualizar o borrar.

Parámetros:
sqlString: Una cadena de consulta en SQL nativo.

Regresa:
La consulta creada. Una instancia del tipo `Query`.

`Query` **`createNativeQuery`**`(String sqlString, Class resultClass)`

Crea una consulta dinámica en el lenguaje SQLque obtiene una entidad simple.

Parámetros:
sqlString: Una cadena de consulta en SQL nativo.
`resultClass`: La clase de la(s) instancia(s) resultante(s).

Regresa:
La consulta creada. Una instancia del tipo `Query`.

`Query` **`createNativeQuery`**`(String sqlString, String result-setMapping)`

Crea una instancia de `Query` para ejecutar una consulta SQL nativo.

Parámetros:
sqlString: Una cadena de consulta en SQL nativo.
`result-setMapping`: El nombre del mapeo del resultado.

Regresa:
La consulta creada. Una instancia del tipo `Query`.

Para crear una consulta debemos tener un manejador de entidad abierto. Por ejemplo, el siguiente código nos crea una consulta con nombre para obtener todos los géneros de la BD.

```
    public List<Genero> findAllGeneroEntities() {
        // Crea un administrador de entidades
        EntityManager em = getEntityManager();

        try {
            // Genera la consulta con nombre
            Query q = em.createNamedQuery("Genero.findAll");

            return q.getResultList();
        } finally {
            em.close();
```

```
        }
```

El siguiente código nos crea una consulta dinámica para obtener todos los géneros de la BD.

```
    public List<Genero> findAllGeneroEntities2() {
        // Crea un administrador de entidades
        EntityManager em = getEntityManager();

        try {
            // Genera la consulta dinamica
            Query q = em.createQuery("select g from Genero g");

            return q.getResultList();
        } finally {
            em.close();
        }
```

Por último, el siguiente código nos crea una consulta dinámica nativa para obtener todos los géneros de la BD.

```
    public List<Genero> findAllGeneroEntities2() {
        // Crea un administrador de entidades
        EntityManager em = getEntityManager();

        try {
            // Genera la consulta dinamica
            Query q = createNativeQuery("select * from generos",
                                        Genero.class);

            return q.getResultList();
        } finally {
            em.close();
        }
```

Ejecución de Consultas

La interfaz `Query` nos permite controlar la ejecución de las consultas. Los métodos de esta interfaz pueden usarse tanto con las consultas con nombre, las consultas dinámicas y las consultas dinámicas nativas. Los métodos de esta interfaz están en la tabla 16.6.

Tabla 16.6 Interfaz Query

`List` **`getResultList`**`()` Este método ejecuta una consulta SELECT y regresa los resultados en una lista sin tipo. **Regresa:** Una lista de los resultados. **Lanza**: `IllegalStateException` - Si se le llama con una sentencia UPDATE o DELETE en el

lenguaje de consultas de la persistencia de Java, JPQL.

`QueryTimeOutException` – Si la ejecución de la consulta excede el tiempo establecido (timeout) y sólo la sentencia se regresa a su estado original.

`PersistenceException` – Si la ejecución de la consulta excede el tiempo establecido (timeout) y la transacción se regresa a su estado original.

`Object` **`getSingleResult`**`()`

Este método ejecuta una consulta SELECT que regresa un solo resultado sin tipo.

Regresa:

El resultado.

Lanza:

`NoResultException` – Si no hay un resultado.

`NoUniqueResultException` – Si hay más de un resultado.

`IllegalStateException` - Si se le llama con una sentencia UPDATE o DELETE en el lenguaje de consultas de la persistencia de Java, JPQL.

`QueryTimeOutException` – Si la ejecución de la consulta excede el tiempo establecido (timeout) y sólo la sentencia se regresa a su estado original.

`PersistenceException` – Si la ejecución de la consulta excede el tiempo establecido (timeout) y la transacción se regresa a su estado original.

`int` **`executeUpdate`**`()`

Este método ejecuta una sentencia UPDATE o SELECT .

Regresa:

El número de entidades actualizadas o borradas.

Lanza:

`IllegalStateException` - Si se le llama con una sentencia SELECT en el lenguaje de consultas de la persistencia de Java, JPQL.

`TransactionRequiredException` – Si no hay una transacción.

`QueryTimeOutException` – Si la ejecución de la consulta excede el tiempo establecido (timeout) y sólo la sentencia se regresa a su estado original.

`PersistenceException` – Si la ejecución de la consulta excede el tiempo establecido (timeout) y la transacción se regresa a su estado original.

`Query` **`setMaxResults`**`(int` *`maxResult`*`)`

Establece el número máximo de resultados a obtener.

Parámetros:

`maxResult`: El número máximo de resultados a obtener.

Regresa:

La misma instancia de la consulta.

Lanza:

`IllegalArgumentException` – Si el parámetro es negativo.

`Query` **`setFirstResult`**`(int` *`startPosition`*`)`

Establece la posición del primer resultado a obtener.

Parámetros:

`startPosition`: Posición del primer resultado a obtenerm numerados desde 0.

Regresa:
La misma instancia de la consulta.

Lanza:
`IllegalArgumentException` – Si el parámetro es negativo.

`Query` **`setParameter`**`(String` *`name`*`, Object` *`value`*`)`

Liga un argumento a un parámetro nombrado

Parámetros:
`name`: Nombre del parámetro.
`value`: Valor del parámetro.

Regresa:
La misma instancia de la consulta.

Lanza:
`IllegalArgumentException` – Si el parámetro name no corresponde a un parámetro de la consulta o si el argumento es del tipo incorrecto,

Obtención de un resultado simple

Para obtener un solo resultado de una consulta, podemos utilizar el método `getSingleResult()` como se ilustra en el siguiente código. Debemos estar seguros que la consulta sólo generará un resultado:

```
    public Genero findGeneroByNombre(String nombre) {
        // Crea un administrador de entidades
        EntityManager em = getEntityManager();

        try {
            // Genera la consulta con nombre
            Query q = em.createNamedQuery("Genero.findByNombre");
            // Establece el nombre del genero a buscar
            q.setParameter("nombre", nombre);

            // Obten y regresa el genero de nombre nombre
            return (Genero)q.getSingleResult();
        } finally {
            em.close();
        }
    }
```

Obtención de resultados múltiples

Para obtener resultados múltiples de una consulta, podemos utilizar el método `getResultList()` como se ilustra en el siguiente código:

```
public List<Genero> findByTipoMedioGeneroEntities(char tipoMedio) {
    // Crea un administrador de entidades
    EntityManager em = getEntityManager();

    try {
        // Genera la consulta con nombre
        Query q = em.createNamedQuery("Genero.findByTipoMedio");
        // Establece el tipo de medio de los generos a buscar
        q.setParameter("tipoMedio", tipoMedio);

        // Obten y regresa la lista de generos del mismo tipo de medio
        return q.getResultList();
    } finally {
        em.close();
    }
}
```

Parámetros de una Consulta

Si una consulta contiene la clausula WHERE, podemos hacer que la consulta tenga parámetros en los que se reciban los valores a emplearse en la clausula WHERE. Por ejemplo en la siguiente consulta con nombre:

```
@NamedQuery(name = "Genero.findByNombre",
                query = "SELECT g FROM Genero g
                WHERE g.nombre = :nombre"),
```

La expresión **:nombre** es un parámetro en la que se recibirá el nombre del género del que se hará la búsqueda.

Al crear la consulta podemos establecer el valor que tomará el parámetro **:nombre** usando el método **setParameter()** como se muestra en el siguiente código:

```
public Genero findGeneroByNombre(String nombre) {
    // Crea un administrador de entidades
    EntityManager em = getEntityManager();

    try {
        // Genera la consulta con nombre
        Query q = em.createNamedQuery("Genero.findByNombre");
        // Establece el nombre del genero a buscar
        q.setParameter("nombre", nombre);

        // Obten y regresa el genero de nombre nombre
        return (Genero)q.getSingleResult();
    } finally {
        em.close();
    }
}
```

La consulta puede tener más de un parámetro. Sus valores se establecerán con tantas invocaciones al método **setParameter()** como parámetros haya.

Paginación de resultados

El resultado de una consulta puede generar cientos, miles o aún millones de entidades. Puede no ser práctico o posible almacenarlos todos en una lista. La interfaz `Query` nos permite limitar el número de resultados a obtener y establecer la posición del primer resultado de la lista. Para ello podemos utilizar los métodos `setMaxresults()` y `setFirstResult()`, respectivamente. El siguiente código es un ejemplo del uso de esos métodos:

```
public List<Genero> findGeneroEntities(int maxResults, int firstResult) {
    // Crea un administrador de entidades
    EntityManager em = getEntityManager();

    try {
        // Genera el query
        Query q = em.createQuery("select object(o) from Genero as o");
        // Establece el numero de generos a recuperar
        q.setMaxResults(maxResults);

        // Establece la posicion del primer genero a regresar
        q.setFirstResult(firstResult);

        // Obtiene la lista y la regresa
        return q.getResultList();
    } finally {
        em.close();
    }
}
```

El Lenguaje JPQL

El lenguaje de Consultas de la Persistencia de Java, JPQL, es un lenguaje de consultas similar a SQL sólo que en lugar de operar sobre tablas, renglones y columnas como lo hace SQL, opera sobre entidades y sus atributos o propiedades.

Las consultas de JPQL son traducidas a SQL por el parser de consultas de JPQL.

Sentencias de JPQL

JPQL soporta las sentencias mostradas en la tabla 16.7

Tabla 16.7 Tipos de Sentencias de JPQL

Tipo de Sentencia	Descripción
SELECT	Recupera entidades o datos relacionados a entidades
UPDATE	Actualiza una o más entidades
DELETE	Elimina una o más entidades

La Sentencia Select

La sintaxis de la sentencia select es:

```
clausula Select
   clausula From
   [clausula Where]
   [clausula Group by [clausula Having]]
   [clausula Order by]
```

Clausula Select

La clausula Select define los resultados de la consulta. Su sintaxis es:

```
SELECT [DISTINT] expresion1[, expresion2]...
```

Donde **`DISTINT`** elimina los resultados repetidos.

`expresion1` es una expresión que establece el resultado de la consulta. Y puede ser:

- Una variable de identificación que identifica a una. Por ejemplo la sentencia:

  ```
  SELECT g FROM Genero g
  ```

  Regresaría un objeto de tipo Genero (o una colección de tipo Genero, dependiendo de la clausula Where).

- Una expresión de trayectoria que identifica un atributo o propiedad de una entidad. Por ejemplo la sentencia:

  ```
  SELECT g.nombre FROM Genero g
  ```

  Regresaría un String con el nombre de un genero (o una colección de tipo String, dependiendo de la clausula Where).

- Un constructor de la clase. En este caso los resultados de la consulta se usarían para formar un objeto de la clase del constructor. Por ejemplo la sentencia:

  ```
  SELECT NEW objetosNegocio.Contacto(c.telefono, c.email)
     FROM cliente c
  ```

  Regresaría un objeto de tipo `objetosnegocio.Contacto` (o una colección de tipo Contacto, dependiendo de la clausula Where). Aquí suponemos que tenemos la siguiente clase:

  ```
  public class Contacto {
     private String telefono;
     private String email;
  ```

```
    public Contacto(String telefono, String email) {
       this.telefono = telefono;
       this.email = email:
    }

    ...
}
```

- Una agregación. Para ello JPQL soporta las funciones de la tabla 16.8

Tabla 16.8 Funciones de Agregación de JPQL

Función de Agregación	Descripción	Tipo Regresado
AVG	Regresa el valor promedio de los valores del campo al que se aplica	Double
COUNT	Regresa el número de resultados regresados por la consulta	Long
MAX	Regresa el valor máximo de los valores del campo al que se aplica	Depende del tipo del campo al que se aplica
MIN	Regresa el valor mínimo de los valores del campo al que se aplica	Depende del tipo del campo al que se aplica
SUM	Regresa la suma de los valores del campo al que se aplica	Double o Long

Por ejemplo la sentencia:

```
SELECT AVG(c.duracion) FROM Cancion c
```

Regresaría un Double con la duración promedio de las canciones.

```
SELECT MAX(p.duracion) FROM Pelicula p
```

Regresaría un Double con la duración de la película más larga.

Por último:

```
SELECT COUNT(c) FROM Cancion c
```

Regresaría el número de canciones.

- En el caso de expresiones múltiples, se regresará un arreglo de tipo Object. Cada elemento del arreglo contiene el resultado de cada expresión. Por ejemplo la sentencia:

```
SELECT g.clave, g.nombre FROM Genero g
```

Regresaría un arreglo de tamaño 2 del tipo String con la clave en el elemento 0 y el nombre wn wl wlwmwnto 1 (o una colección de arreglos de tamaño 2 del tipo String, dependiendo de la clausula Where).

Clausula From

La clausula FROM establece los nombres de las entidades que serán usadas en la consulta, asociándoles variables de identificación. Una clausula from puede tener varias variables de identificación separadas por comas. Las variables de identificación deben ser identificadores válidos de java y no pueden ser ninguno de los nombres reservados de JPQL. La tabla 16.9 lista todos los nombres reservados de JPQL-

Tabla 16.9 Nombres Reservados de JPQL

SELECT	OBJECT	OF	POSITION
FROM	NULL	IS AVG	CHARACTER_LENGTH
WHERE	TRUE	MAX	CHAR_LENGTH
UPDATE	FALSE	MIN	BIT_LENGTH
DELETE	NOT	SUM	CURRENT_TIME
JOIN	AND	COUNT	CURRENT_DATE
OUTER	OR	ORDER	CURRENT_TIMESTAMP
INNER	BETWEEN	BY	NEW
LEFT	LIKE	ASC	EXISTS
GROUP	IN	DESC	ALL
BY	AS	MOD	ANY
HAVING	UNKNOWN	UPPER	SOME
FETCH	EMPTY	LOWER	
DISTINCT	MEMBER	TRIM	

La sintaxis de una clausula FROM es la siguiente:

```
FROM nombreEntidad [AS] variableIdentificacion
```

Clausula Where

La clausula Where permite filtrar los resultados de la consulta. Sólo las entidades que cumplen con la condición establecida por la clausula Where serán recuperadas. La sintaxis de una clausula Where es la siguiente:

```
WHERE expresión
```

Donde expresión puede contener variables de identificación, expresiones de trayectoria, y los operadores soportados por el lenguaje JPQL. La tabla 16.10 muestra los operadores soportados por JPQL en orden de precedencia.

Tabla 16.10 Operadores de JPQL

Tipo de Operador	Operador
Navegacional	.
Signo	+, -
Aritméticos	*, / +, -
Relacional	=, >, >=, <, <=, <> [NOT] BETWEEN, [NOT]LIKE [NOT] IN, IS [NOT] NULL, IS [NOT]EMPTY, [NOT] MEMBER[OF]
Lógico	NOT AND OR

- Por ejemplo la sentencia:

```
SELECT g FROM Genero g WHERE g.tipoMedio = :tipoMedio
```

  Regresaría los géneros del tipo de medio específicado por su parámetro.

- La sentencia:

```
SELECT c FROM Cancion c WHERE c.duracion > :duracion
```

  Regresaría las canciones cuya duración sea mayor que su parámetro.

- La sentencia:

```
SELECT c FROM Cancion c WHERE c.autor IS NULL
```

  Regresaría las canciones que no tienen autor.

- Por último, la sentencia:

```
SELECT p FROM Pelicula p WHERE p.fecha BETWEEN :fecha1 AND :fecha2
```

  Regresaría las películas producidas en el periodo comprendido entre las fechas dadas por los parámetros `fecha1` y `fecha2`.

Clausula Group By

La clausula Group By define la expresión de agrupamiento sobre la que se agregarán los resultados. La expresión de agrupación debe ser una expresión de trayectoria simple o una variable de identificación.

- La sentencia:

```
SELECT NEW utils.CuentaCancionesInterprete(c.interprete, COUNT(c))
FROM Cancion c GROUP BY c.interprete");
```

Regresaría el número de canciones para cada intérprete en una lista de objetos de objetos de tipo `utils.CuentaCancionesInterprete`, cuyo código parcial es:

```
package utils;
import objetosNegocio.Cancion;

public class CuentaCancionesInterprete {
    private String interprete;
    private Long cuenta;

    public CuentaCancionesInterprete() {
    }

    public CuentaCancionesInterprete(String interprete,
                                     Long cuenta) {
        this.interprete = interprete;
        this.cuenta = cuenta;
    }

    @Override
    public String toString() {
        return interprete + ", " + cuenta;
    }
}
```

Clausula Having

La clausula Having define un filtro que se aplicará después de que los resultados hayan sido agrupados. Equivale a una clausula Where secundaria. Su sintaxis es similar a la de la clausula Where.

```
HAVING expresión
```

La expresión sólo puede tener laexpresión de trayectoria simple o la variable de identificación usada en la clausula Group By. La expresión condicional de la clausula Having puede contener funciones de agregación.

- La sentencia:

```
SELECT NEW utils.CuentaCancionesGenero(c.genero.nombre, count(c))
FROM Cancion c GROUP BY c.genero.nombre HAVING c.genero.nombre =
:nomGenero
```

Regresaría el número de canciones que son baladas en una lista de objetos de objetos de tipo `utils.CuentaCancionesGenero`, cuyo código parcial es:

```
package utils;

/**
 *
```

```
 * @author mdomitsu
 */
public class CuentaCancionesGenero {
    private String nomGenero;
    private Long cuenta;

    public CuentaCancionesGenero() {
    }

    public CuentaCancionesGenero(String nomGenero, Long cuenta) {
        this.nomGenero = nomGenero;
        this.cuenta = cuenta;
    }

    @Override
    public String toString() {
        return nomGenero + ", " + cuenta;
    }
}
```

Clausula Order By

La clausula Order By permite ordenar los resultados de la consulta usando una o más expresiones que contengan variables de identificación o expresiones de trayectoria que den como resultado una entidad o atributo. Las palabras reservadas ASC y DESC permiten establecer que el ordenamiento sea ascendente o descendente. El valor por omición es ASC. La sintaxis de una clausula Order By es la siguiente:

```
Order By expresión[, expresion2]
```

- La sentencia:

  ```
  SELECT c FROM Cancion c WHERE c.duracion > :duracion ORDER BY
  c.titulo
  ```

  Regresaría las canciones cuya duración sea mayor que su parámetro, ordenadas por título.

- La sentencia:

  ```
  SELECT p FROM Pelicula p ORDER BY p.autor, p.titulo
  ```

  Regresaría las películas ordenadas por autor y luego por título.