11ª Emisión

DIPLOMADO

# Desarrollo de Sistemas con Tecnología Java

## Módulo 11

## Persistencia con Jakarta

*Dr. Omar Mendoza González*

*omarmendoza564@aragon.unam.mx*

DGTIC

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Dirección General de Cómputo y de Tecnologías de información y Comunicación

Dirección de Docencia en TIC

50 AÑOS Educación Continua 1971 - 2021

# Primary Key

- Cada entidad debe tener una *primary key*.
- Una primary key corresponde a uno o más campos o propiedades de la clase de entidad.
- Una primary key simple o un campo o propiedad de una clave primaria compuesta debe ser uno de los siguientes tipos:
  - Cualquier tipo primitivo Java,
  - cualquier tipo contenedor primitivo
  - java.lang.String
  - java.util.Date
  - java.sql.Date
  - java.math.BigDecimal
  - java.math.BigInteger

# Primary Key

- Se supone que las primary key se pueden insertar, pero no se pueden anular ni actualizar.

- Al anular una columna de primary key, los elementos que aceptan valores NULL y actualizables no deben anularse.

- Solo en la circunstancia muy específica de mapear la misma columna a múltiples campos/relaciones el elemento insertable debe establecerse en falso.

# Generación de identificadores

- Los valores del identificador se generen automáticamente y se especifica mediante la anotación **@GeneratedValue.**
- Cuando la generación de ID está habilitada, el proveedor de persistencia generará un valor de identificador para cada instancia de ese tipo de entidad.
- Una vez que se obtiene el valor del identificador, el proveedor lo insertará en la nueva entidad persistente; sin embargo, dependiendo de la forma en que se genere, es posible que no esté presente en el objeto hasta que la entidad se haya insertado en la BD.
- Cuatro estrategias de generación de ID
  - AUTO
  - TABLE
  - SEQUENCE
  - IDENTITY

# Generación de identificadores

- Auto ID Generation

```
@Entity
public class Employee {
  @Id @GeneratedValue(strategy=GenerationType.AUTO)
  private long id;
  // ...
}
```

# Generación de identificadores

- Generación de ID usando una ***tabla***
- La forma más flexible y portátil de generar identificadores es utilizar una tabla de base de datos.
- Es portable entre diferentes SGBD y permite almacenar múltiples secuencias de identificadores diferentes para diferentes entidades dentro de la misma tabla.
- Una tabla de generación de ID debe tener dos columnas.
  - La primera columna es tipo cadena y se utiliza para identificar la secuencia del generador de ID.
  - La segunda columna es tipo integer y almacena la secuencia de ID real que se está generando.

# Generación de identificadores

- Generación de ID usando una ***tabla***
- La forma más fácil de usar una tabla para generar identificadores es simplemente especificar que la estrategia de generación sea TABLE en el elemento de estrategia

```
@Id
@GeneratedValue(strategy=GenerationType.T
ABLE)  private long id;
```

# Generación de identificadores

- Generación de ID usando una ***tabla***
- Se recomienda especificar la tabla que se utilizará para el almacenamiento de ID.
- Usar una anotación *@TableGenerator* y luego referirse a él por su nombre en la anotación *@GeneratedValue*

```
@TableGenerator(name="Emp_Gen",
      table="ID_GEN",
      pkColumnName="GEN_NAME",
      valueColumnName="GEN_VAL")
@Id
@GeneratedValue(strategy=GenerationType.TABLE,
generator="Emp_Gen")
```

# Generación de identificadores

- Generación de ID utilizando una ***Database Sequence***
- Muchas bases de datos admiten un mecanismo interno para la generación de ID llamado secuencias.
- Se puede utilizar una secuencia de base de datos para generar identificadores cuando la base de datos subyacente los admite.

```
CREATE SEQUENCE Emp_Seq MINVALUE 1 START WITH 100 INCREMENT
BY 50;

@SequenceGenerator(name="Emp_Gen",
sequenceName="Emp_Seq")  @Id
@GeneratedValue(generator="Emp_Gen")
private int id;
```

# Relaciones

- En toda relación hay dos entidades que están relacionadas entre sí, y se dice que cada entidad desempeña un papel en la relación
  - Un empleado tiene una relación con el departamento en el que trabaja.
  - La entidad Empleado desempeña la función de trabajar en el departamento, mientras que la entidad Departamento desempeña la función de tener un empleado trabajando en ella.

- Una entidad puede estar participando en muchas relaciones diferentes con muchas entidades diferentes

- Cualquier entidad podría estar desempeñando una serie de roles diferentes en cualquier modelo dado

# Direccionalidad

- Una entidad que tiene un atributo de relación se refiere a su entidad relacionada de una manera que la identifica como desempeñando el otro rol de la relación.

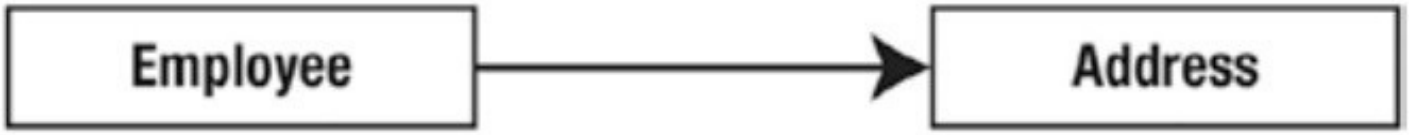
- A menudo sucede que la otra entidad, a su vez, tiene un atributo que apunta a la entidad original.

- Cuando cada entidad apunta a la otra, la relación es ***bidireccional***.

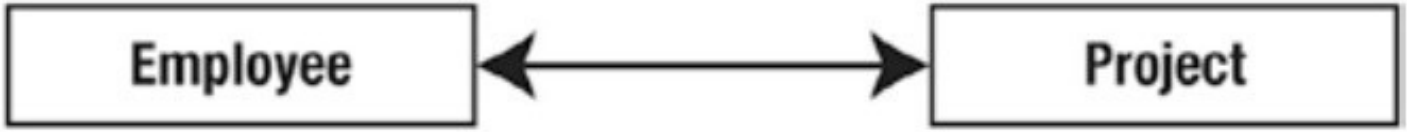
- Si solo una entidad tiene un puntero a la otra, se dice que la relación es
- ***unidireccional***.

# Direccionalidad

- Relaciones Unidireccionales




- Relaciones Bidireccionales



# Cardinalidad

- Indica si puede haber solo una instancia de la entidad o varias instancias en la relación.
  - One-to-one
  - One-to-many
  - Many-to-one
  - Many-to-many

# Direccionalidad y Cardinalidad

- *Relacion Unidireccional many-to-one*


- *Relacion Bidirectional many-to-many*

# Join Columns

- En la base de datos, un mapeo de relaciones significa que una tabla tiene una referencia a otra tabla.

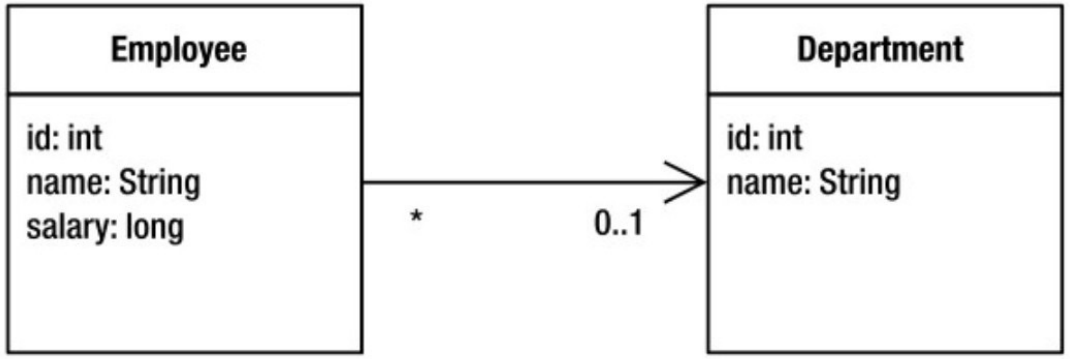
- El término de la base de datos para una columna que hace referencia a una *primary key* en otra tabla es una columna de *foreign key*.

- En Jakarta Persistence, se denominan join columns y la anotación
  - ***@JoinColumn*** se utiliza para configurar este tipo de columnas.

# Join Columns

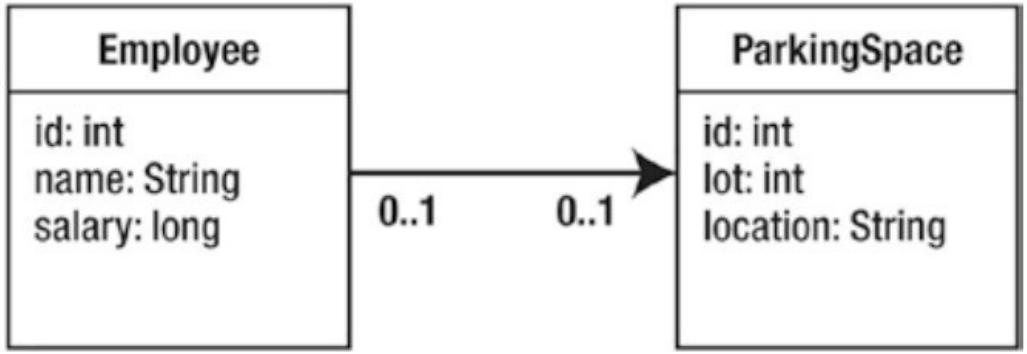
```
@Entity
public class Employee {
  @Id private long id;
  @ManyToOne
  @JoinColumn(name="DEPT_ID")
  private Department department;
  // ...
}
```

# ManyToOne - Unidireccional



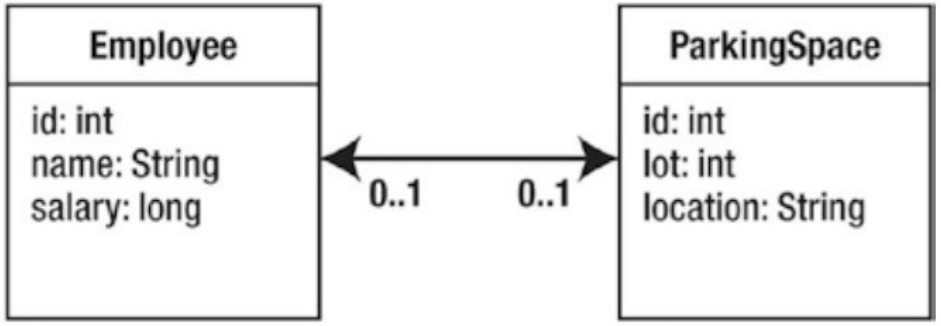

```
@Entity
public class Employee {
    // ...
    @ManyToOne
    private Department department;
    // ...
}
```

# OneToOne Unidireccional




```
@Entity
public class Employee {  @Id  private
      long id;  private String name;
      @OneToOne
      @JoinColumn(name="PSPACE_ID")  private
      ParkingSpace parkingSpace;
      // ...
}
```

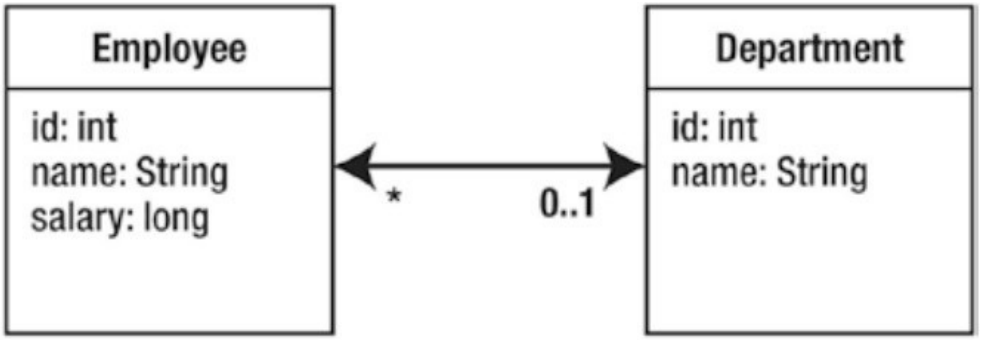
# OneToOne Bidireccional




```
@Entity
public class Employee {
    @Id private long
    id; private String
    name;
    @OneToOne
    @JoinColumn(name="PSPACE_ID
    ") private ParkingSpace
    parkingSpace;
    // ...
}
```

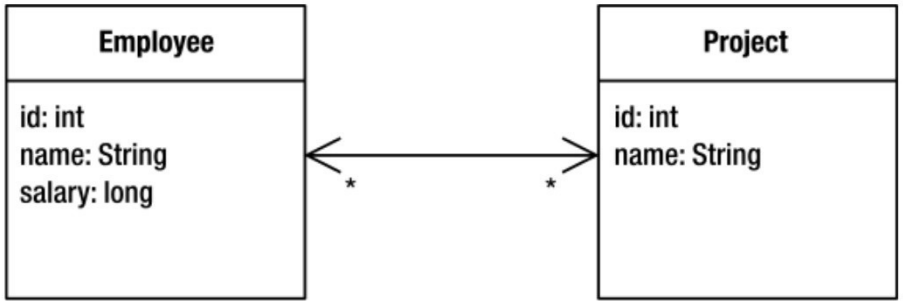
```
@Entity
public class ParkingSpace
    { @Id private long
    id; private int lot;
    private String location;
    @OneToOne(mappedBy="parkingSpac
    e")
    private Employee employee;
    // ...
}
```

# OneToMany Bidireccional




```
@Entity
public class Department{
   @Id private long id;
   private String name;
   @OneToMany(mappedBy="department")
   private Collection<Employee> employees;
   // ...
}
```

# ManyToMany - Bidireccional




```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @ManyToMany
    private Collection<Project> projects;
    // ...
}
```

```
@Entity
public class Project {
    @Id private int id;
    private String name;
    @ManyToMany(mappedBy="projects")
    private Collection<Employee> employees;
    // ...
}
```

# Join Tables

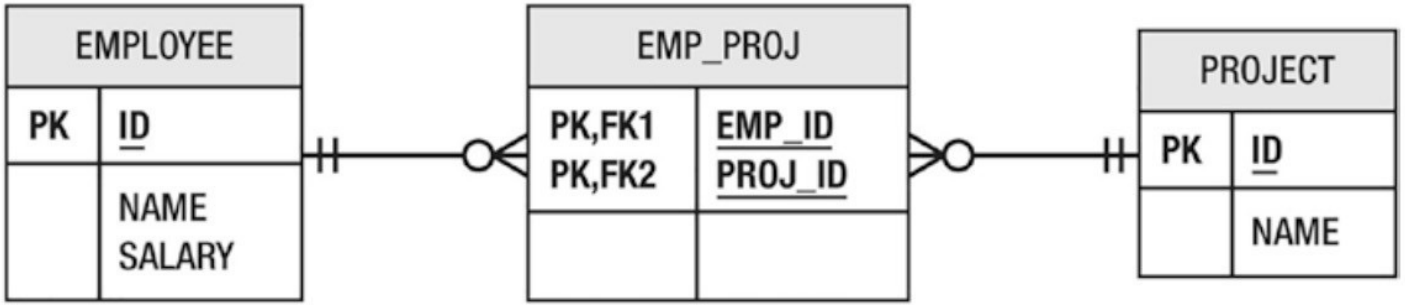
- En una relación de N:M ninguna de las dos tablas de entidades puede almacenar un conjunto ilimitado de valores de PK en una sola fila de entidad.

- Se debe usar una **tercera tabla** para asociar los dos tipos de entidad conocida como **Join Table**

- Consta de dos **foreign key** o **join columns** para hacer referencia a cada uno de los dos tipos de entidad en la relación.

- Se asigna una colección de entidades como **filas múltiples** en la tabla, cada una de las cuales asocia una entidad con otra que representa la **colección de entidades relacionadas** con esa entidad determinada.

# Join Tables




```
@Entity
public class Employee{
   @Id private long id;
   private String name;
   @ManyToMany
   @JoinTable(name="EMP_PROJ",
        joinColumns=@JoinColumn(name="EMP_ID"),
        inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
   private Collection<Project> projects;
   // ...
}
```

# Anotaciones

- Se definen en el paquete jakarta.persistence

- AssociationOverride
- AttributeOverride
- Convert
- JoinColumn
- MapKeyJoinColumn
- NamedEntityGraph
- NamedNativeQuery
- NamedQuery
- NamedStoredProcedureQuery
- PersistenceContext
- PersistenceUnit
- PrimaryKeyJoinColumn
- SecondaryTable
- SqlResultSetMapping

# Anotaciones

- Las anotaciones de persistencia se pueden aplicar en tres niveles diferentes
  - Clase
  - Método
  - Campo

- Para anotar cualquiera de estos niveles, la anotación debe colocarse antes de la definición de código del artefacto que se está anotando.
- En algunos casos, se coloca en la misma línea justo antes de la clase, el método o el campo; en otros casos, en la línea de arriba.

# Principales anotaciones

- @Table
  - Con esta anotación se establece el nombre de la tabla.
- @Id
  - Es obligatorio que toda entidad tenga un identificador único.
- @UniqueConstraint
  - Define la unicidad de una o varias columnas en conjunto
- @Index
  - Permite la definición de índices tanto únicos como no únicos.

# Principales anotaciones

```
@Entity
@Table(name="cities", uniqueConstraints={
          @UniqueConstraint(name="CITY_REGION_UK",
           columnNames = {"name" , "region"})})
public class City {  @Id
   private Long id;  private
   String name;  private String
   region;
}
```

# Principales anotaciones

- @Column
  - Personaliza la relación entre los atributos de la entidad y las columnas de la tabla.
  - *name*. El nombre de la columna.
  - *insertable* (true, false, por omisión true). Incluye\excluye el atributo de las operaciones de inserción, realizadas con una sentencia INSERT de SQL.
  - *updatable* (true, false, por omisión true). Incluye\excluye el atributo de las operaciones de actualización, realizadas con una sentencia UPDATE de SQL.

```
@Column(updatable = false, nullable = false)
private LocalDateTime creation;
```

# Principales anotaciones

```
@Entity  @Table(name="measures")  public class
Measure {
 @Id
 @Column(insertable = false, updatable = false)  private Long
 id;

 @Column(insertable = false, updatable = false)  private
 BigDecimal value;

 @Column(insertable = false, updatable = false)  private
 LocalDateTime moment;
}
```

# Principales anotaciones

- @Temporal
  - Indica cómo debe tratarse un atributo de tipo Date o Calendar según un valor del enumerado TemporalType: fecha, tiempo, o fecha y hora.
  - Si esta anotación no se usa, se considera fecha y hora.

- @Enumerated
  - Aunque algunas bases de datos incorporan un tipo enumerado, JPA define una conversión válida para cualquiera de ellas.
  - Por omisión, los enumerados se traducen a un entero que representa la posición, empezando en cero, de cada valor según el orden en el que declaran.

# Principales anotaciones

- @Lob
  - Las bases de datos cuentan con tipos específicos para las columnas que tienen que almacenar cadenas de texto de gran tamaño y datos binarios

- @Embeddable
  - Los atributos incrustados o «embebidos» son una forma directa y sencilla de usar clases propias como tipos de atributos en JPA.
  - Estas clases no son entidades, lo que implica que carecen de identificador, y sus datos se guardan en la tabla de la entidad que las contengan.

# Principales anotaciones

```
@Entity
@Table(name = "users")  public class User {
   @Id
   private Long id;  @Embedded
   private Name name;
  @Embeddable
  public class Name {
     @Column(name = "FIRST_NAME")
     private String firstName;
     @Column(name = "LAST_NAME")  private String lastName;
  }
  }
```

# Transacciones

- El servicio de transacciones que debe usarse en Java SE es el servicio **jakarta.persistence.EntityTransaction**.

- Al ejecutar en Java SE, se deben usar los métodos ***begin*** y ***commit*** para realizar la transacción

- Una transacción se inicia llamando a ***getTransaction()*** en el entity manager para obtener **EntityTransaction** y luego se invoca ***begin()***

- Para confirmar la transacción, se invoca la llamada ***commit()*** sobre el objeto ***EntityTransaction*** obtenido del entity manager

# Transacciones

```
em.getTransaction().begin();
alumno = service.raiseAlumnoEstatura(matricula, 1);
em.getTransaction().commit();
```

# Queries

- En Jakarta Persistence, una consulta es similar a una consulta de base de datos, excepto que en lugar de usar SQL para especificar los criterios de consulta, se consulta sobre entidades y se usa un lenguaje llamado Jakarta Persistence QL (***JPQL***).
- Una consulta se implementa en el código como un objeto ***Query*** o ***TypedQuery<X>***.
- Se construye utilizando EntityManager que incluye una variedad de llamadas API que devuelven un nuevo objeto Query o TypedQuery<X>.
- Como objeto de primera clase, una consulta puede a su vez personalizarse según las necesidades de la aplicación.

# Queries

- Una consulta se puede definir de forma estática o dinámica.
  - Una consulta **estática** normalmente se define en una anotación o metadatos XML y debe incluir los criterios de consulta, así como un nombre asignado por el usuario.
  - Se puede emitir una consulta **dinámica** en tiempo de ejecución proporcionando los criterios de consulta QL de persistencia de Jakarta o un objeto de criterios.
- Es posible que sean un poco más costosos de ejecutar porque el proveedor de persistencia no puede realizar ninguna preparación de consulta de antemano, pero las consultas QL de Persistencia de Jakarta son, sin embargo, muy fáciles de usar y se pueden emitir en respuesta a la lógica del programa o incluso a la lógica del usuario.

# Queries

```
public Collection<Alumno> findAllAlumnos() {
        TypedQuery<Alumno> query =
    em.createQuery("SELECT e FROM Alumno e",
    Alumno.class);
        return query.getResultList();
}
```

# Queries

- Una vez que se tenga la consulta, se ejecuta con alguno de estos métodos
  - ***getSingleResult***
    - SELECT que devuelve un único resultado. Si retorna más de uno, se lanza la excepción NonUniqueResultException. Si no hay ninguno, lanza NoResultException.
  - ***getResultList***
    - SELECT que puede devolver más de un resultado en una lista (List), lo que garantiza el orden. Si no hay datos, estará vacía.
  - ***getResultStream***
    - EjecutA una consulta SELECT y devuelva los resultados de la consulta como *java.util.stream.Stream* sin tipo. De forma predeterminada, este método delega a getResultList().stream().
  - ***executeUpdate***
    - Ejecuta una sentencia UPDATE, DELETE o, solo en HQL, INSERT.

# Contacto

Dr. Omar Mendoza González

omarmendoza564@aragon.unam.mx