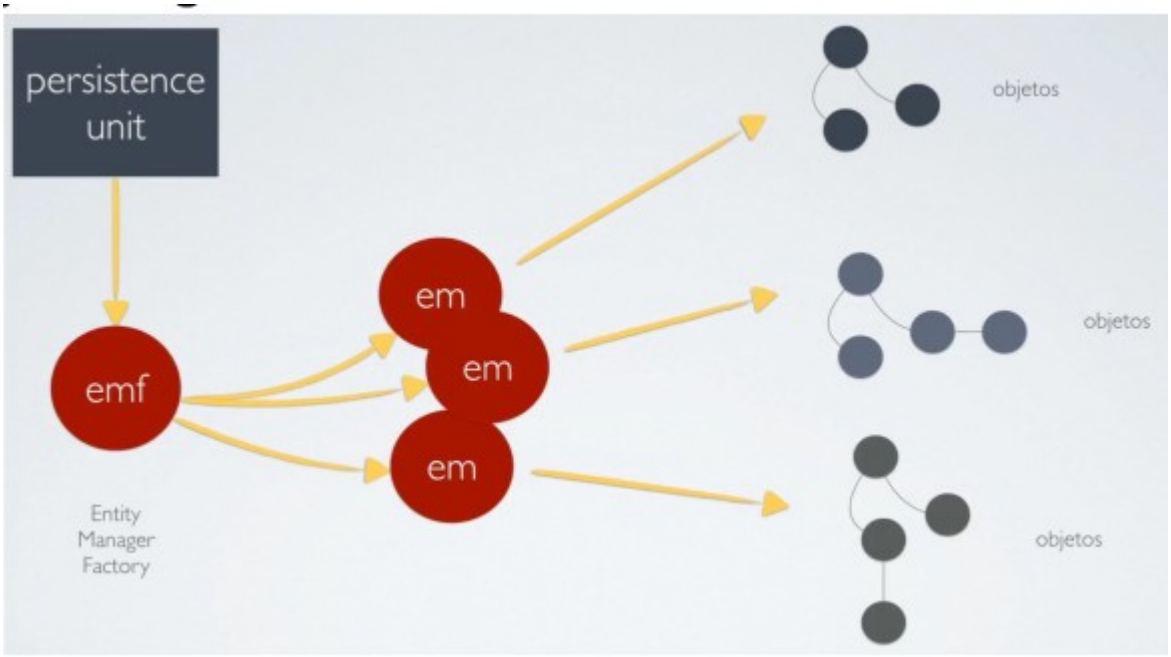
Jakarta Persistence.

JPA = Jakarta Persistence.

Entidad → Objeto de dominio ligero y persistente.
Una clase de entidad debe anotarse como *Entity,* con un Constructor público sin argumentos.
*Pueden ser abstractas.
*Pueden extender de clases no entidades.

Jakarta maneja un almacén de persistencia, en el que se pueden hacer operaciones de entidades, y solo se persistirá a la Base de Datos cuando se mande llamar al método de manera explícita.

Entity Manager.
API que provee de métodos para añadir, modificar o eliminar objetos del contexto.
Se usa en lugar del SQL y de las llamadas a JDBC.
Aunque también permiten consultas con JPQL y SQL.

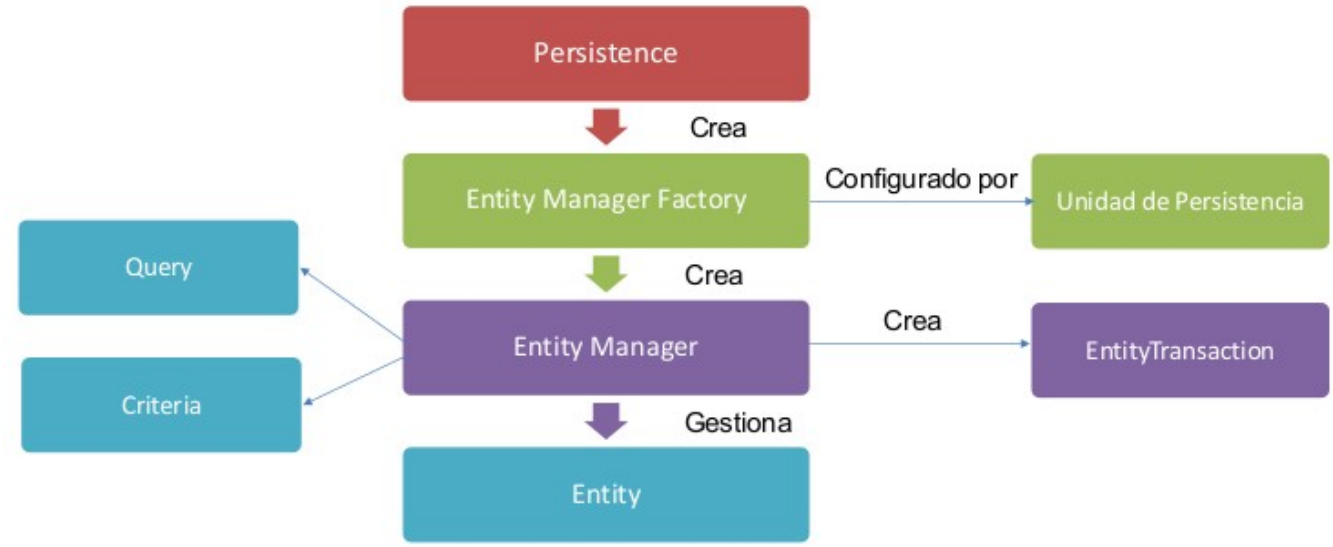


Es una relación 1:1 de la *UNIDAD DE PERSISTENCIA* y *EL ENTITY MANAGER FACTORY.*

Contexto de persistencia.
También llamado *sesión.* Posee todas las instancias de las entidades de la Base de Datos.
Se crea a partir de un archivo xml llamado *pesistence.xml*.

Ciclo de vida.
- Nuevo (Transient). No tiene vinculación con la Base de Datos ni con el contexto.
- Persistente, Gestionado (Persistent, Managed). La entidad pertenece al contexto.
- Desligado (Detached). La entidad ya no está en el contexto y sus cambios no pasan a la BD.
- Eliminada (Removed). Entidad eliminada del contexto, sin haber eliminado el contexto.

Jakarta Persitence



Unidades de persistencia.

- Debe tener un nombre único (Permiten elegir qué *Entity Manager Factory* se va a usar).
- Contiene lo siguiente:
  - Entity Manager Factory y sus Managers.
  - El conjunto de clases gestionadas en el archivo de persistencia.

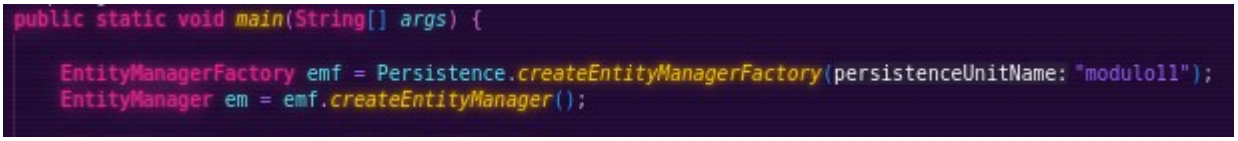
Persistence.xml
Define las entidades persistidas.
Se ubica en el directorio META-INF (la raíz de la unidad de persistencia).

El esquema es:
https://jakarta.ee/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
    version="3.0">
    <!-- Unidad donde se persistirán las unidades-->
    <persistence-unit name="modulo11">
        <class>unam.dgtic.diplomado.ml100base.dominio.Alumno</class>
        <class>unam.dgtic.diplomado.ml100base.dominio.Employee</class>      You, 2 weeks ago • Finish base exercise …
        <properties>
            <property name="jakarta.persistence.jdbc.driver" value="org.mariadb.jdbc.Driver" />
            <property name="jakarta.persistence.jdbc.url" value="jdbc:mariadb://localhost:3306/modulo11" />
            <property name="jakarta.persistence.jdbc.user" value="root" />
            <property name="jakarta.persistence.jdbc.password" value="MaPassw" />
        </properties>
    </persistence-unit>
    <persistence-unit name="modulo7">
        <class>unam.dgtic.diplomado.ml100base.dominio.Alumno</class>
        <properties>
            <property name="jakarta.persistence.jdbc.driver" value="org.mariadb.jdbc.Driver" />
            <property name="jakarta.persistence.jdbc.url" value="jdbc:mariadb://localhost:3306/modulo7" />
            <property name="jakarta.persistence.jdbc.user" value="root" />
            <property name="jakarta.persistence.jdbc.password" value="MaPassw" />
        </properties>
    </persistence-unit>
</persistence>
```

¿Cómo obtener un Entity Manager?

Se obtienen después de haber creado un *Entity Manager Factory*.



Persistir una entidad.

```
Alumno alumno = new Alumno(matricula);
em.persist(alumno);
```

*Si no es posible llevar a cabo la persistencia, se lanzará una *PersistenceException*.
Al terminar el método de persistencia, "alumno" se convierte en una entidad gestionada dentro del contexto de persistencia.

Localizar una entidad.

```
em.find(Alumno.class, matricula);
```

Se pasa la clase de la entidad y el *Id*.
Si no lo encuentra, devuelve null.

Eliminar una entidad.

```
Alumno alumno = findAlumno(matricula);
if (alumno != null) {
    em.remove(alumno);
}
```

Actualizar una entidad.

```
Alumno alumno = em.find(Alumno.class, matricula);
if (alumno != null) {
    alumno.setEstatura(alumno.getEstatura() + raise);
}
```

Mapeo de entidades.

Las clases deberán tener la anotación *@Entity* y si el nombre es diferente a la tabla, se deberá usar *@Table(name = "…").*

Campos y propiedades.

Deben ser privados o protegidos.
Deben tener *Getters y Setters* públicos.
Si son colecciones, usas *Collection, List, Map o Set.*
Los atributos pueden ser:

- Primitivos.
- String.
- Enums.
- Otras entidades.
- Clases definidas por el programados.
- Tiempo:
  Otros tipos que implementan la interfaz java.io.Serializable (tipos de contenedor primitivo, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql .Hora, java.sql.Timestamp, byte[], Byte[], char[], Carácter[], java.time.LocalDate, java.time.LocalTime, java.time.LocalDateTime, java.time.OffsetTime, java .time.OffsetDateTime)

Mapeo de Columnas.

Se puede especificar el nombre de la columna con *name* en el Atributo.

Tipos temporales.

La lista de tipos temporales admitidos incluye los tres tipos de

- java.sql
  - java.sql.Date
  - java.sql.Time
  - java.sql.Timestamp
- java.util
  - java.util.Date
  - java.util .Calendario.

Los tipos *util* deben ser tratados de manera especial para poder comunicarse con el Controlador del JDBC.
Se debe usar la anotación *@Temporal*.

```
@Entity public class Employee {
   @Id private long id;
   @Temporal(TemporalType.DATE)
   private Calendar dob;
   @Temporal(TemporalType.DATE)
   @Column(name="S_DATE")
   private Date startDate;
   // ...
}
```

Propiedades transitorias.

Se deben anotar con *@Transient* aquellas propiedades que no se planeen pasar a la Base de Datos.

Primary Key.

Cuando una una estrategia de generación para llaves primarias, se debe anotar el Id con *@GeneratedValue(strategy = GenerationType…)*
Auto.
Table.
Sequence.
Identity.

- Tabla.
  La generación por tabla debe tener dos columnas.
  Una para la secuencia del generador.
  Otra para almacenar el ID.

- Secuencia.

```
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;
```

- Identity.
  Simplemente se aplica para los valores autoincrementales.

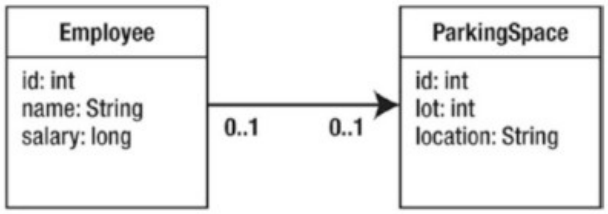
Cardinalidad.

Las cardinalidades disponibles en Jakarta Persistence son:

- One-to-one
- One-to-many
- Many-to-one
- Many-to-many

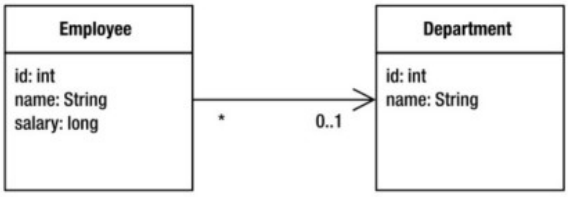
Pueden ser tanto unidireccionales como bidireccionales.

Ejemplos.



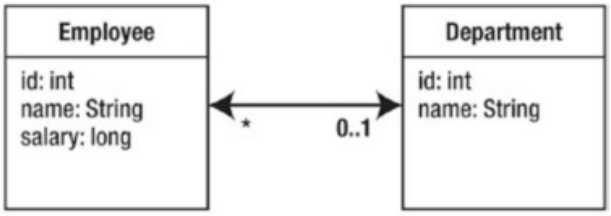

```
@Entity
public class Employee { @Id private
    long id; private String name;
    @OneToOne
    @JoinColumn(name="PSPACE_ID") private
    ParkingSpace parkingSpace;
    // ...
}
```

ManyToOne Unidireccional.



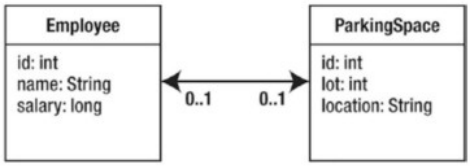

```
@Entity
public class Employee {
    // ...
    @ManyToOne
    private Department department;
    // ...
}
```

OneToOne Unidireccional.




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

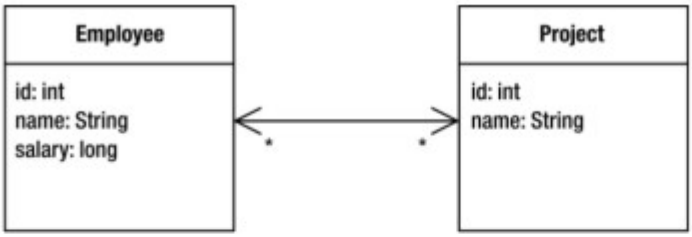
OneToOne Bidireccional.




```
@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.



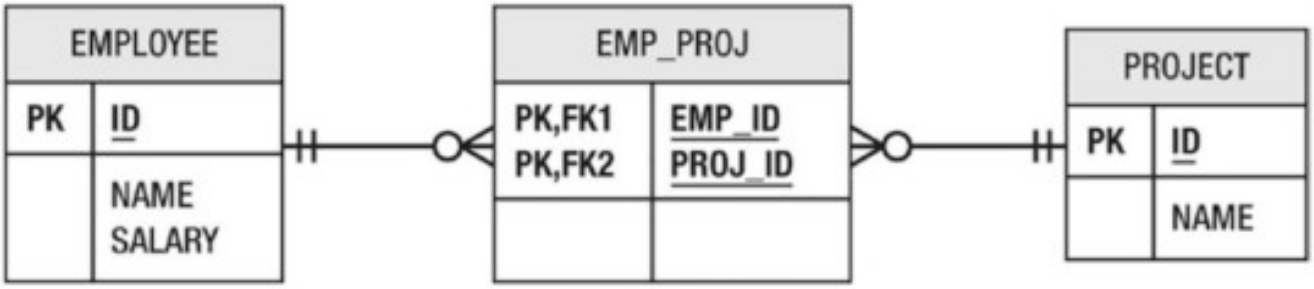

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @ManyToMany
    private Collection<Project> projects;
    // ...
}
```

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

ManyToMany Bidireccional.

Join Tables.

Para relaciones M:N, no se pueden almacenar colecciones de ambas tablas, sino que debe haber una tercera tabla conocida como *Join Table*.
Consta de dos Join Columns con los Id de las otras tablas.




```
@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;
   //...
}
```

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

@Embeddable.

Las clases anotadas así, no son entidades, lo que significa que carecen de identificador y sus datos se resguardan en la tabla de la entidad que las contengan.

```
@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.

Se debe iniciar con *begin* y terminar con *commit* para que los cambios se vean reflejados.

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

Queries.

Se usa el Jakarta Persistence Query Language (JQPL) en vez del SQL ordinario.
Pueden ser dinámicas o estáticas.
La primera es a través de Anotaciones, mientras que la segunda es mediante QL de Jakarta en tiempo de ejecución.

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

Para obtener el resultado de una consulta, es posible elegir entre varias opciones:

- ***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.

Clases embebidas en las entidades.

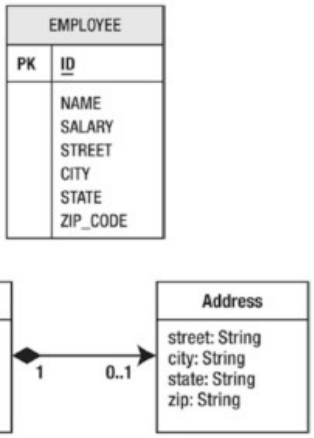
Las colecciones de las entidades pueden almacenar tres tipos de datos:

- Colecciones de entidades.
- Embeddables.
- Tipos básicos.

Las últimas dos no son relacionales, solo son colecciones de elementos.
Un objeto embebido es aquél que depende de una entidad. No tiene identidad propia, sino que es parte del estado de la entidad que ha sido definido y almacenado en un objeto Java.

La clase debe ser anotada como *@Embeddable*. Así, sus campos serán persistidos como parte de una entidad.

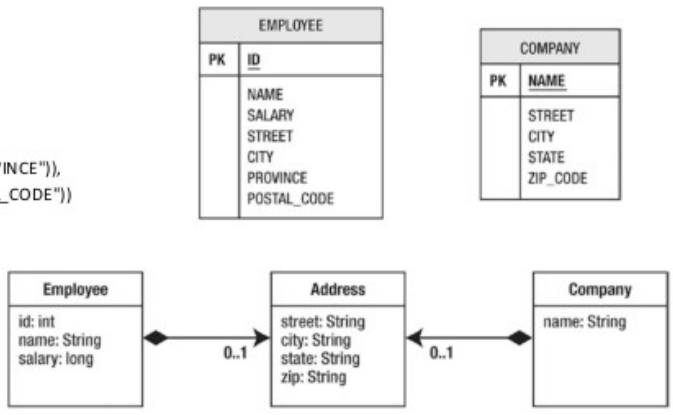
```
@Entity
public class Employee {
   @Id private long id;
   private String name;
   private long salary;
   @Embedded private Address address;
   // ...
}
@Embeddable @Access(AccessType.FIELD)
public class Address {
   private String street;
   private String city;
   private String state;
   @Column(name="ZIP_CODE")
   private String zip;
   // ...
}
```



Attribute Override:

```
@Entity
public class Employee {
   @Id private long id;
   private String name;
   private long salary;
   @Embedded
   @AttributeOverride(name="state", column=@Column(name="PROVINCE")),
   @AttributeOverride(name="zip", column=@Column(name="POSTAL_CODE"))
   private Address address;
// ... }

@Entity
public class Company {
   @Id private String name;
   @Embedded
   private Address address;
   // ...
}
```



Collection Mapping.

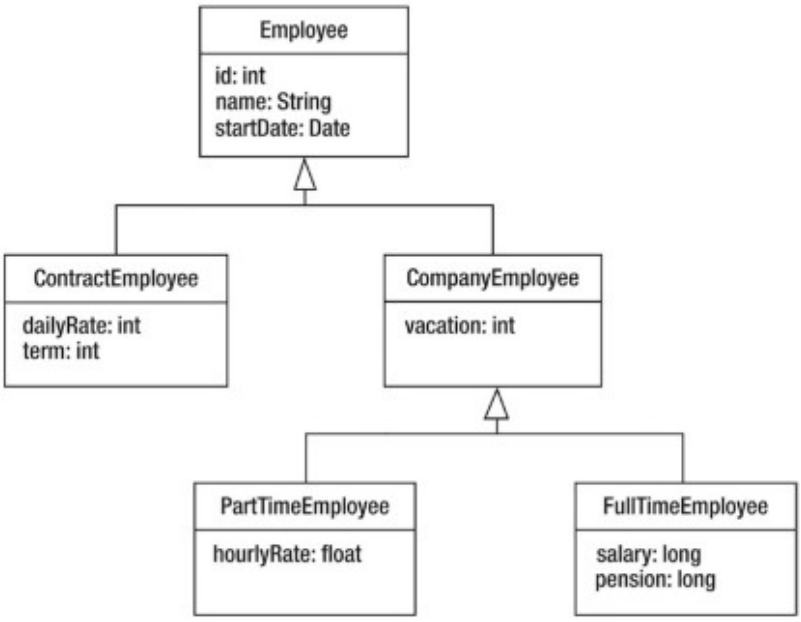
Las colecciones de elementos requieren una tabla separada llamada *tabla de colección*.
Cada tabla debe tener una *join column*.

```
@Entity
public class Employee {
   @Id private int id;
   private String name;
   private long salary;
   // ...
   @ElementCollection(targetClass=VacationEntry.class)
   private Collection vacationBookings;
   @ElementCollection
   private Set<String> nickNames;
// ... }
```

Herencia.

- Ignorar la jerarquía (**Mapped Superclass**)
- Una tabla para todo (**Single Table**)
- Una tabla por clase (**Table per Class**)
- Tablas unidas (**Joined Table**)

Discriminator value.



```
@Entity
public class Employee {
   @Id private int id;
   private String name;
   @Temporal(TemporalType.DATE)
   @Column(name="S_DATE")
private Date startDate;
// ... }

@Entity
public class ContractEmployee extends Employee {
   @Column(name="D_RATE")
   private int dailyRate;
   private int term;
   // ...
}
```

```
@MappedSuperclass
public abstract class CompanyEmployee extends Employee {
   private int vacation;
// ... }
@Entity
public class FullTimeEmployee extends CompanyEmployee {
   private long salary;
   private long pension;
   // ...
}
@Entity
public class PartTimeEmployee extends CompanyEmployee {
   @Column(name="H_RATE")
   private float hourlyRate;
   // ...
}
```

Single table.

```
@Entity
@Table(name="EMP")
@Inheritance
@DiscriminatorColumn(name="EMP_TYPE")
public abstract class Employee { ... }
@Entity
public class ContractEmployee extends Employee { ... }
@MappedSuperclass
public abstract class CompanyEmployee extends Employee { ... }
@Entity
@DiscriminatorValue("FTEmp")
public class FullTimeEmployee extends CompanyEmployee { ... }
@Entity(name="PTEmp")
public class PartTimeEmployee extends CompanyEmployee { ... }
```

EMPLOYEE	
PK	ID
	NAME
	S_DATE
	D_DATE
	TERM
	VACATION
	H_RATE
	SALARY
	PENSION
	EMO_TYPE

Table per class.

CONTRACT_EMP	
PK	ID
	FULLNAME
	S_DATE
	D_RATE
	TERM

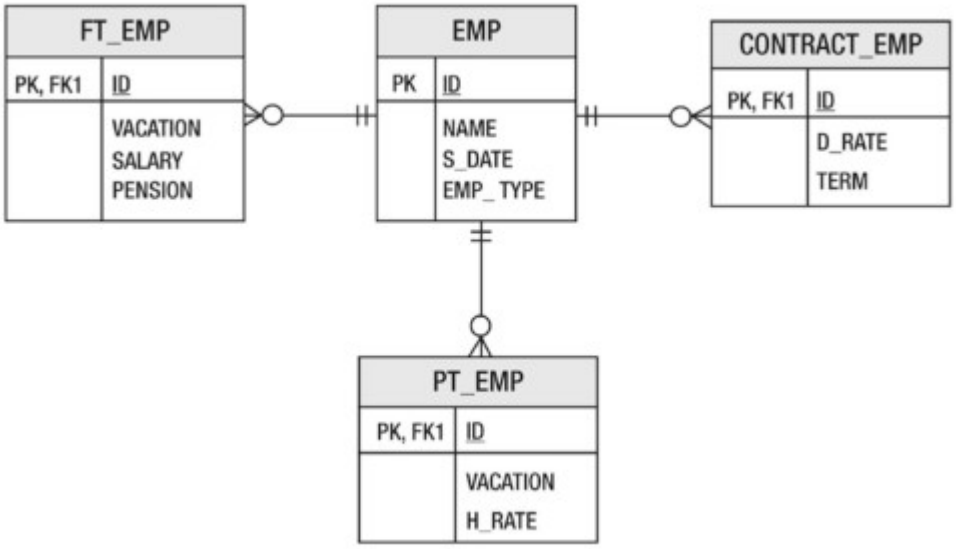
FT_EMP	
PK	ID
	NAME
	S_DATE
	VACATION
	SALARY
	FENSION
FK1	MANAGER

PT_EMP	
PK	ID
	NAME
	S_DATE
	VACATION
	H_RATE
FK1	MGR

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Employee {
   @Id private int id;
   private String name;
   @Temporal(TemporalType.DATE)
   @Column(name="S_DATE")
   private Date startDate;
   // ...
}
@Entity
@Table(name="CONTRACT_EMP")
@AttributeOverride(name="name", column=@Column(name="FULLNAME"))
@AttributeOverride(name="startDate", column=@Column(name="SDATE"))
public class ContractEmployee extends Employee {
   @Column(name="D_RATE")
   private int dailyRate;
   private int term;
   // ...
}
```

```
@MappedSuperclass
public abstract class CompanyEmployee extends Employee {
   private int vacation;
   @ManyToOne
   private Employee manager;
   // ...
}
@Entity @Table(name="FT_EMP")
public class FullTimeEmployee extends CompanyEmployee {
   private long salary;
   @Column(name="PENSION")
 private long pensionContribution;
// ... }
@Entity
@Table(name="PT_EMP")
@AssociationOverride(name="manager",
            joinColumns=@JoinColumn(name="MGR"))
public class PartTimeEmployee extends CompanyEmployee {
   @Column(name="H_RATE")
   private float hourlyRate;
   // ...
}
```

Joined table.




```
@Entity
@Table(name="EMP")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="EMP_TYPE",

discriminatorType=DiscriminatorType.INTEGER)
public abstract class Employee { ... }
@Entity
@Table(name="CONTRACT_EMP")
@DiscriminatorValue("1")
public class ContractEmployee extends Employee { ...
}
```

```
@MappedSuperclass
public abstract class CompanyEmployee extends
Employee { ... }
@Entity
@Table(name="FT_EMP")
@DiscriminatorValue("2")
public class FullTimeEmployee extends
CompanyEmployee { ... }
@Entity
@Table(name="PT_EMP")
@DiscriminatorValue("3")
public class PartTimeEmployee extends
CompanyEmployee { ... }
```