Operaciones CRUD con Hibernate

- El método get(<T> class, id) recupera un objeto asociado a una tabla mediante su clave principal. Devuelve nulo si no existe ninguna tupla con ese valor de clave

Inserción y Actualización

- Para insertar una nueva tupla o actualizar una existente se utiliza el método saveOrUpdate(). Si el objeto ya existe en la base de datos, Hibernate actualizará la tupla. En caso contrario, se insertará una nueva tupla

```
public void insertaActualizaMonitor(Session sesionMonitor monitor) throws Exception {
      sesion.saveOrUpdate(monitor);
  }
```

Borrado

- Para eliminar una tupla se utiliza el método delete()

```
public void eliminaMonitor(Session sesion, Monitor codMonitor) throws Exception {
       sesion.delete(monitor);
  }
```

Mapeo de relaciones

Relaciones "uno a muchos"

- En general, en *Hibernate* las relaciones se definen de forma bidireccional
- En una relación "uno a muchos", un objeto de clase A está relacionado con un conjunto de objetos de clase B, y un objeto de clase B tendrá un campo que referencie a un objeto de clase A

Monitor.java

```
@OneToMany(mappedBy = "monitorResponsable")
private Set<Actividad> actividadesResponsable = new HashSet<Actividad>();
```

Actividad.java

```
@JoinColumn(name = "monitorResponsable", referencedColumnName = "codMonitor")
@ManyToOne
private Monitor monitorResponsable;
```

Mapeo de relaciones

Creación de una nueva actividad asignándole un monitor responsable de la base de datos

```
sesion = sessionFactory.openSession();
tr = sesion.beginTransaction();
try {
        Monitor monitor = sesion.get(Monitor.class, "M010");
        if (monitor == null) {
          vMensajes.mensajeConsola("Monitor no existe en la BD");
        } else {
                Actividad actividad = new Actividad("AC99", "Nueva Actividad","Muy dura",30);
                actividad.setMonitorResponsable(monitor);
                monitor.getActividadesResponsable().add(actividad);
                sesion.saveOrUpdate(actividad);
                tr.commit();
      }
 } completar con catch y finally
```

Con el método **setMonitorResponsable()** de la clase **Actividad**, asignamos el monitor responsable a la actividad

Método para hacer persistente (almacenar) la nueva actividad en la base de datos

Con el método **getActividadesResponsable()** de la clase **Monitor**, recuperamos el conjunto de actividades (Set<Actividad>) de las que es responsable el monitor, y con el método **add()** la añadimos a su conjunto de actividades.

Mapeo de relaciones

Actualización del responsable de una actividad

```
sesion = sessionFactory.openSession();
tr = sesion.beginTransaction();
try {
        Monitor monitor = sesion.get(Monitor.class, "M010");
        if (monitor == null) {
                vMensajes.mensajeConsola("Monitor no existe en la BD");
        } else {
                Actividad actividad = sesion.get(Actividad.class, "AC99");
                actividad.setMonitorResponsable(monitor);
                monitor.getActividadesResponsable().add(actividad);
                sesion.saveOrUpdate(actividad);
                tr.commit();
        }
} completar con catch y finally
```

En este ejemplo estamos actualizando el monitor responsable de la actividad "AC99", asignándole el monitor "M010"

Mapeo de relaciones

Listado del nombre de las actividades junto con el nombre y correo electrónico del monitor responsable

```
sesion = sessionFactory.openSession();
tr = sesion.beginTransaction();
try {

        Query consulta = sesion.createQuery("SELECT a FROM Actividad a", Actividad.class);
        List<Actividad> actividades = consulta.list();
        for (Actividad actividad : actividades) {
                System.out.println(actividad.getNombre() + "\t"
                        + actividad.getMonitorResponsable().getNombre() + "\t"
                        + actividad.getMonitorResponsable().getCorreo());
        }
        tr.commit();

} completar con catch y finally
```

En este caso hemos devuelto el resultado de la consulta en una estructura de tipo **List<Actividad>**. En realidad, se puede usar cualquier estructura que ofrezca el lenguaje de programación. La elección dependerá de factores como la eficiencia, mantener compatibilidad con código ya existente, etc.

NOTA: este ejemplo es una "mezcla" de código de modelo, vista y controlador Sólo es de uso didáctico para que se aprecie todo en una única diapositiva

Mapeo de relaciones

Listado del nombre de las actividades junto con el nombre y correo electrónico del monitor responsable

De una forma más eficiente, sin necesidad de recuperar todos los campos de la tabla ACTIVIDAD

```
sesion = sessionFactory.openSession();
tr = sesion.beginTransaction();
try {

        Query consulta = sesion.createQuery("SELECT a.nombre, a.monitorResponsable
                                                FROM Actividad a");
        List<Object[]> actividades = consulta.list();
        for (Object[] actividad : actividades) {
                Monitor monitorResponsable = (Monitor) actividad[1];
                System.out.println(actividad[0] + "\t"
                        + monitorResponsable.getNombre() + "\t"
                        + monitorResponsable.getCorreo());
        }
        tr.commit();

} completar con catch y finally
```

Mapeo de relaciones

Listado del nombre de los monitores y las actividades de las que son responsables

```
sesion = sessionFactory.openSession();
tr = sesion.beginTransaction();
try {

        Query consulta = sesion.createQuery("SELECT m FROM Monitor m", Monitor.class);
        List<Monitor> monitores = consulta.list();
        for (Monitor monitor : monitores) {
                System.out.println(monitor.getNombre());
                Set<Actividad> actividades = monitor.getActividadesResponsable();
                for (Actividad actividad : actividades)
                        System.out.println(actividad.getNombre());
        }
        tr.commit();

} completar con catch y finally
```

Mapeo de relaciones

Relaciones "muchos a muchos"

- En una relación "muchos a muchos" no se crea una clase para la tabla intermedia. Un objeto de clase A está relacionado con un conjunto de objetos de clase B y viceversa

- Una de las entidades actúa como "propietaria" de la relación. Esa entidad indica, mediante la anotación @JoinTable, la tabla de la BD que une a las dos tablas (en nuestro ejemplo, la tabla REALIZA). Además, tendrá un campo para almacenar el conjunto de objetos de la otra entidad (en nuestro ejemplo, SOCIO) y se anotará con @ManyToMany

Actividad.java

```
@JoinTable(name = "REALIZA",
joinColumns = { @JoinColumn(name = "idActividad", referencedColumnName = "idActividad")},
inverseJoinColumns = { @JoinColumn(name = "numeroSocio", referencedColumnName = "numeroSocio")})

@ManyToMany
private Set<Socio> socios = new HashSet<Socio>();
```

Mapeo de relaciones

Relaciones "muchos a muchos"

- La otra entidad (en nuestro ejemplo, SOCIO) simplemente tendrá la anotación @ManyToMany para indicar el campo con el que está relacionado de la otra tabla. Además, también tendrá un conjunto de objetos de la otra entidad (en nuestro ejemplo, ACTIVIDAD)

- Esta estructura hay inicializarla para que no se produzcan errores de "nulos"

- Al persistir cualquier objeto también se persiste su conjunto de objetos relacionados

Socio.java

```
@ManyToMany(mappedBy = "socios")
private Set<Actividad> actividades = new HashSet<Actividad>();
```

Mapeo de relaciones

- En este tipo de mapeo es muy conveniente mantener la consistencia en ambas direcciones. Una buena práctica consiste en implementar las 2 operaciones (en ambos objetos) en una única operación

Crea un nuevo socio, le asigna la actividad "AC01" y lo inserta en la BD

```
sesion = sessionFactory.openSession();
tr = sesion.beginTransaction();
try {
        Socio socioNuevo = new Socio("S999", "Nuevo Socio", "11222333F", "20/10/2022", 'A');
        Actividad actividad = sesion.get(Actividad.class, "AC01");
        actividad.altaSocio(socioNuevo);
        sesion.saveOrUpdate(socioNuevo);      // es necesario porque es un nuevo socio en la BD
        sesion.saveOrUpdate(actividad);
        tr.commit();

} completar con catch y finally
```

Al ejecutarse este código se insertará una tupla en la tabla SOCIO con los datos del nuevo socio y la tupla ("S999", "AC01") en la tabla REALIZA

¡¡ Hay que añadirlo al POJO correspondiente !!

Método de la clase Actividad

```
public void altaSocio(Socio socio) {
        this.socios.add(socio);
        socio.getActividades().add(this);  }
```

Mapeo de relaciones

Asigna una actividad que ya existe a un socio que también existe

```
sesion = sessionFactory.openSession();
tr = sesion.beginTransaction();
try {

        Socio socio = sesion.get(Socio.class, "S999");
        Actividad actividad = sesion.get(Actividad.class, "AC03");

        actividad.altaSocio(socio);

      // una vez asignadas las relaciones en los dos sentidos, se puede realizar
      // una operación saveOrUpdate() de cualquiera de los dos objetos para que
      // se almacene la tupla en la tabla intermedia

        sesion.saveOrUpdate(actividad);  // también sesion.saveOrUpdate(socio)

        tr.commit();

} completar con catch y finally
```

Mapeo de relaciones

Elimina una actividad de un socio y, de camino, un socio de una actividad

```
sesion = sessionFactory.openSession();
tr = sesion.beginTransaction();
try {

        Socio socio = sesion.get(Socio.class, "S999");
        Actividad actividad = sesion.get(Actividad.class, "AC03");
        actividad.bajaSocio(socio);
        sesion.saveOrUpdate(actividad);

        tr.commit();

} completar con catch y finally
```

Este código elimina la tupla ("S999", "AC03") de la tabla REALIZA

¡¡ Hay que añadirlo al POJO correspondiente !!

Método de la clase Actividad

```
public void bajaSocio(Socio socio) {
        this.socios.remove(socio);
        socio.getActividades().remove(this);
}
```

Mapeo de relaciones

Listado del nombre de los socios junto con el nombre de las actividades que realiza

```
sesion = sessionFactory.openSession();
tr = sesion.beginTransaction();
try {

        Query consulta = sesion.createQuery("SELECT s FROM Socio s", Socio.class);
        List<Socio> socios = consulta.list();
        for (Socio socio : socios) {
                System.out.println(socio.getNombre());
                Set<Actividad> actividades = socio.getActividades();
                for (Actividad actividad : actividades )
                        System.out.println(actividad.getNombre());
        }
        tr.commit();

} completar con catch y finally
```