MP_0486. Acceso a datos
UF3. Herramientas de mapeo objeto relacional

3.1. Conceptos básicos sobre mapeo objeto-relacional. Frameworks ORM.

Índice

Introducción

Objetivos

En esta lección perseguimos los siguientes objetivos:

- Sentar algunos conceptos previos para comprender mejor el funcionamiento de los *frameworks ORM*: *beans* y anotaciones.
- Describir el significado de *framework* ORM y la técnica ORM (mapeo objeto relacional).
- Enumerar algunos fremeworks comerciales que implementan el ORM.
- Describir el funcionamiento de las herramientas ORM.

Conceptos previos

Los JavaBeans

Un *JavaBean* es una clase que debe complir una serie de condiciones.

Estas condiciones son:

- Debe implementar la interfaz *Serializable*, lo que otorga a sus objetos la capacidad de persistencia.
- Debe tener un constructor vacío (que no reciba argumentos), aunque luego puede tener otros constructores. De esta forma se podrán crear objetos estándar.
- Todas las propiedades del objeto serán privadas y accesibles mediante métodos *get/set* que serán públicos.
- Para los métodos *get/set*, hay que seguir cuidadosamente la nomenclatura estándar. Para una propiedad privada llamada *precio*, los métodos *get/set* serán *getPrecio* y *setPrecio*. Las palabras *get* y *set* en minúscula y el nombre de la propiedad con la primera letra mayúscula.

El concepto *JavaBean* fue creado por Sun Microsystems, aunque luego esta compañía fue adquirida por Oracle en 2010. Los *JavaBean* nacieron con el objetivo de ser utilizarados como componentes software reutilizables.

Veamos un ejemplo de clase que cumple con las especificaciones de los *JavaBeans*:

```
import java.io.Serializable;
import java.time.LocalDateTime;

public class Llamada implements Serializable {
	private static final long serialVersionUID = 6164080316086841480L;

	private LocalDateTime fechaHora;
	private String emisor; // Nombre de la persona que llamo.
	private String motivo; // Motivo de la llamada.

	public Llamada() {
		this.fechaHora = LocalDateTime.now();
	}

	public LocalDateTime getFechaHora() {
		return fechaHora;
	}
	public void setFechaHora(LocalDateTime fechaHora) {
		this.fechaHora = fechaHora;
	}
	public String getEmisor() {
		return emisor;
```

```
	}
	public void setEmisor(String emisor) {
		this.emisor = emisor;
	}
	public String getMotivo() {
		return motivo;
	}
	public void setMotivo(String motivo) {
		this.motivo = motivo;
	}

	@Override
	public String toString() {
		return this.emisor + " llamo el " + this.fechaHora + " para " +
this.motivo;
	}

}
```

La clase *Llamada* cumple con la especificación JavaBeans porque es serializable, tiene un constructor sin argumentos y sus métodos son privados y accesibles mediante métodos *get/set*, generados cumpliendo la nomenclatura estándar.

Podríamos construir un objeto en una clase con método *main()* de la siguiente forma:

```
public class Principal {

	public static void main(String[] args) {
		Llamada unaLlamada = new Llamada();
		unaLlamada.setEmisor("Carlos Pérez");
		unaLlamada.setMotivo("Pedir información");
		System.out.println(unaLlamada); // Invoca al método toString();
	}

}
```

¿Qué es el mapeo objeto-relacional?

En unidades anteriores aprendimos cómo crear programas Java capaces de comunicarse con una base de datos relacional a través del API JDBC.

Ahora vamos a dar un paso más, aprendiendo a utilizar otra técnica muy empleada en aplicaciones empresariales Java: **el Mapeo Objeto-Relacional.**

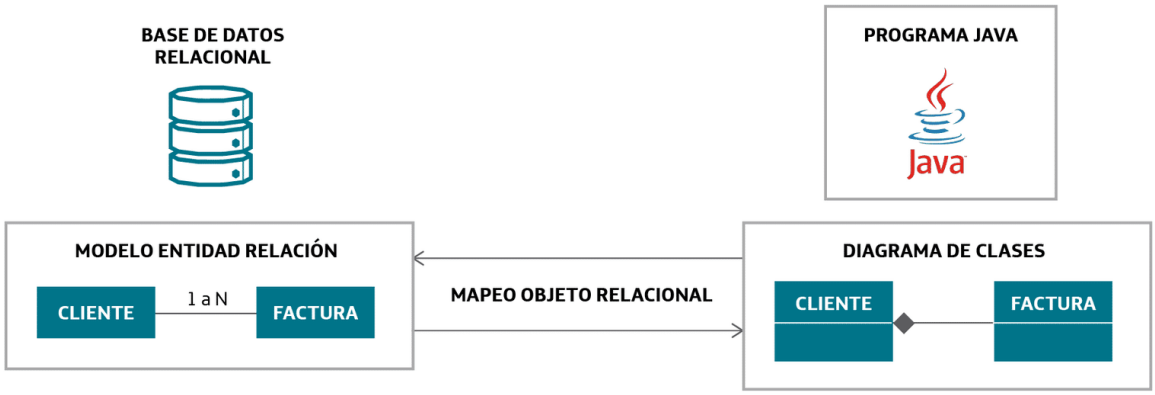
Una base de datos relacional está compuesta por un conjunto de tablas asociadas entre sí, donde cada tabla tiene una clave principal. Por otro lado, un programa Java funciona mediante un conjunto de objetos que se comunican entre sí para lograr un determinado objetivo.

¿Hay similitudes entre ambas cosas?

BD = conjunto de tablas asociadas entre ellas.
Programa java = conjunto de objetos relacionados entre ellos.

El "mapeo objeto-relacional" pretende acercar el mundo de las bases de datos al mundo de los objetos Java.

Para ello se parte del Modelo Entidad-Relación de la base de datos relacional y, a través del proceso de "Mapeo Objeto-Relacional" u ORM, se construye una base de datos *Orientada a Objetos virtual*, es decir, en memoria. En dicho proceso, para cada entidad física o registro de la BD con la que tengamos que trabajar, existirá su correspondiente objeto Java con la misma estructura, de modo que programaremos de una forma más cercana a la filosofía de clases y objetos de Java.



Por un lado, tenemos una base de datos con su estructura relacional, con sus tablas físicas y sus asociaciones, y por otro lado, tenemos clases Java relacionadas que imitan la estructura de la base de datos.

Será una herramienta denominada ***Framework ORM*** la encargada de mapear las clases Java con las tablas físicas de base de datos.

Pero, ¿qué significa *mapear*?

Pensando en nuestra base de datos FERRETERIA, mapear un cliente correspondería a crear automáticamente un objeto Java de la clase *Cliente* con sus objetos *Factura* asociados, a través de la lectura de un registro de la tabla *CLIENTE* de la base de datos *FERRETERIA*. Este proceso lo realiza automáticamente el *framework ORM*.

También podemos crear un nuevo objeto *Cliente*, con sus objetos *Factura* asociados si es necesario, y a continuación hacerlo persistir, es decir, guardarlo de forma física en la base de datos.

Las clases de entidad

Hemos visto cómo en la técnica de mapeo objeto-relacional tenemos, por un lado, una base de datos con su estructura relacional y, por otro lado, clases Java que imitan la estructura de la base de datos.

Estas clases Java se denominan **clases de entidad** y deben cumplir unas normas:

- Serán *JavaBeans*, es decir, cumplirán con las mismas normas que deben cumplir éstos: tener capacidad de persistencia, contar con un constructor vacío, y atributos privados y accesibles mediante métodos *get/set*.
- Utilizarán anotaciones para indicar información adicional a cada atributo o clase, como si se corresponde con una clave principal, con una clave ajena, etc. Las anotaciones van precedidas del símbolo @.

¿Cómo sería la implementación de la clase de entidad *Cliente?*

```
package model;

import java.io.Serializable;
import javax.persistence.*;
import java.util.List;

@Entity
@NamedQuery(name="Cliente.findAll", query="SELECT c FROM Cliente c")
public class Cliente implements Serializable {
	private static final long serialVersionUID = 1L;

	@Id
	private String nif;

	private String ciudad;

	private String domicilio;

	private String nombre;

	private String tlf;

	//bi-directional many-to-one association to Factura
	@OneToMany(mappedBy="cliente")
	private List<Factura> facturas;

	public Cliente() {
	}

	public String getNif() {
		return this.nif;
	}
```

```
        public void setNif(String nif) {
                this.nif = nif;
        }

        public String getCiudad() {
                return this.ciudad;
        }

        public void setCiudad(String ciudad) {
                this.ciudad = ciudad;
        }

        public String getDomicilio() {
                return this.domicilio;
        }

        public void setDomicilio(String domicilio) {
                this.domicilio = domicilio;
        }

        public String getNombre() {
                return this.nombre;
        }

        public void setNombre(String nombre) {
                this.nombre = nombre;
        }

        public String getTlf() {
                return this.tlf;
        }

        public void setTlf(String tlf) {
                this.tlf = tlf;
        }

        public List<Factura> getFacturas() {
                return this.facturas;
        }

        public void setFacturas(List<Factura> facturas) {
                this.facturas = facturas;
        }

        public Factura addFactura(Factura factura) {
                getFacturas().add(factura);
                factura.setCliente(this);

                return factura;
        }

        public Factura removeFactura(Factura factura) {
                getFacturas().remove(factura);
                factura.setCliente(null);

                return factura;
        }

}
```

Observa cómo se utilizan las siguientes anotaciones:

- *@Entity* delante de la cabecera de la clase para indicar que se trata de una clase de entidad.
- *@Id* delante del *nif* para indicar que se trata de la clave principal.
- *@OneToMany(mappedBy="cliente")* delante del atributo *facturas* para indicar que dicho atributo servirá para representar la asociación *uno a muchos* entre el cliente y la factura.

Aprenderemos cómo crear estas clases de entidad y comprenderemos mejor el por qué de cada una de las anotaciones que contienen.

Las anotaciones

Ya sabemos que las clases de entidad utilizan anotaciones. Pero, ¿qué son realmente las anotaciones?

Las anotaciones Java proveen de un sistema para añadir metadatos al código fuente que estarán disponibles para la aplicación en tiempo de ejecución.

¿Qué son los *metadatos*?

La traducción literal de metadato es *sobre dato* y hace referencia a un dato que describe a otro dato. En aplicaciones empresariales se utiliza para añadir información de configuración a un objeto cuya misión es servir como recurso a un proyecto.

En la clase de entidad del apartado anterior, la anotación *@Id*servía como dato para indicar que el dato *dni* es una clave principal.

Obtén más información en la wikipedia. https://es.wikipedia.org/wiki/Metadatos

Un ejemplo de anotación que con seguridad te has encontrado en más de una ocasión es la anotación *@Override* delante de un método, para indicar que dicho método viene heredado de una clase base y está siendo sobrescrito.

Ejercicio práctico

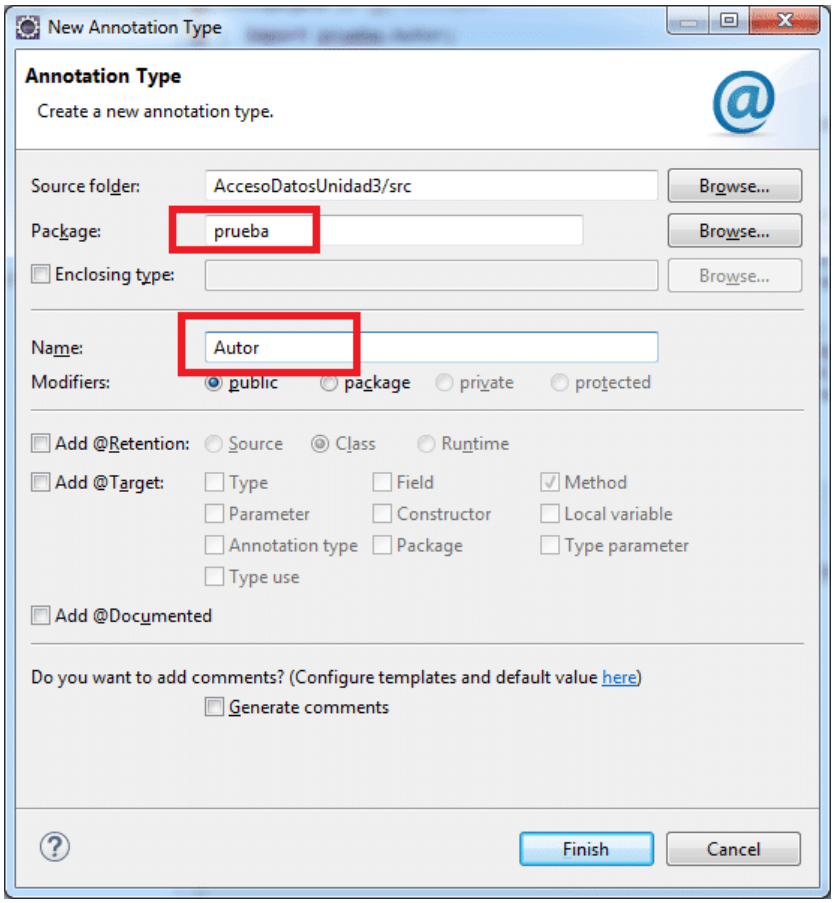
Para que comprendas mejor lo que son las anotaciones Vas a crear tu propia anotación personalizada y a utilizarla después.

Seguiremos los siguientes pasos:

- Crearemos la anotación personalizada *@Autor*, que servirá para añadir información a las clases y métodos que queramos sobre quién es el autor que desarrolló dichas clases o métodos. La anotación contará con las propiedades *nombre* y *direccion*.
- Crearemos una clase llamada *Coche* y la anotaremos. Dentro de la clase *Coche* implementaremos un método llamado *acelerar()*, que también anotaremos.
- En la clase *Principal* crearemos un objeto de la clase *Coche* y accederemos a los datos suministrados por la anotación *@Autor*.

1. Creamos la anotación personalizada *@Autor*

- Crea un proyecto nuevo denominado *pruebaAnotaciones* (*File / New / Java Project*).
- Haz clic derecho sobre el nombre del nuevo proyecto y selecciona en el menú contextual *New / Annotation*.



En el cuadro de diálogo *New Annotation Type* rellena los datos como ves en la imagen para crear la anotación *Autor* en el paquete *prueba*. Termina haciendo clic en el botón *Finish*.

```
package prueba;

public @interface Autor {

}
```

Por ahora tu anotación tiene este aspecto. Las anotaciones son como interfaces especiales. Observa que la palabra interface va precedida por el símbolo de arroba.

Ahora, completa el código de la anotación de la siguiente manera:

```
package prueba;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
public @interface Autor {
        String nombre() default "Telefónica";
        String direccion() default "Distrito Telefónica";

}
```

Ahora la anotación cuenta con los atributos o datos *nombre* y *direccion* cuyos valores predeterminados son "*Teléfonica*" y "*Distrito Telefónica*".

La anotación *@Retention(RetentionPolicy.RUNTIME)* permite que la anotación *@Autor* esté disponible en tiempo de ejecución.

2. Creamos una clase llamada *Coche* y la anotamos

La nueva clase *Coche* tendrá el siguiente código:

```
import prueba.Autor;

@Autor
public class Coche {
        private String marca;
        private String modelo;
        private int velocidad;

        public Coche(String marca, String modelo) {
                this.marca = marca;
                this.modelo = modelo;
                this.velocidad = 0;
        }

        public String getMarca() {
                return marca;
        }
```

```
	public String getModelo() {
		return modelo;
	}
	public int getVelocidad() {
		return velocidad;
	}

	@Autor(nombre="Perico de los palotes", direccion="C/ Palotes, 54")
	public void acelerar() {
		this.velocidad = this.velocidad + 10;
	}

	@Override
	public String toString() {
		return "Coche [marca=" + marca + ", modelo=" + modelo + ",
velocidad=" + velocidad + "]";
	}

}
```

Hemos anotado la nueva clase *Coche* sin especificar ningún valor para los atributos *nombre* y *dirección*, por lo tanto, habrá tomado los valores predeterminados "*Telefónica*" y "*Distrito Telefónica*". Si embargo, el método *acelerar()* ha sido anotado especificando el valor "*Perico de los Palotes*" para el atributo nombre y "*C/ Palotes, 54*" para el atributo *direccion*.

3. En la clase *Principal* creamos un objeto de la clase *Coche* y accedemos a los datos suministrados por la anotación *@Autor*.

Por último, crea la clase *Principal* con el siguiente código:

```
import prueba.Autor;

public class Principal {

	public static void main(String[] args) throws NoSuchMethodException,
SecurityException {
		Coche miCoche = new Coche("Ford", "Fiesta");
		System.out.println(miCoche);

		// Accediendo a los datos de la anotación del método.
		Autor a1 =
miCoche.getClass().getMethod("acelerar").getAnnotation(Autor.class);
		System.out.println("Nombre autor: " + a1.nombre());
		System.out.println("Dirección autor: " + a1.direccion());

		// Accediendo a los datos de la anotación de la clase.
		Autor a2 = miCoche.getClass().getAnnotation(Autor.class);
		System.out.println("Nombre autor: " + a2.nombre());
		System.out.println("Dirección autor: " + a2.direccion());

	}

}
```

Veamos las partes más importantes de la clase *Principal*:

Coche miCoche = new Coche("Ford", "Fiesta");

Creamos un objeto de la clase *Coche* y después mostramos su estado invocando al método *toString()* con esta línea:

System.out.println(miCoche);

Recuerda que *toString()* es el método al que se invoca por defecto cuando no se especifica el método a ejecutar.

Autor a1 = miCoche.getClass().getMethod("acelerar").getAnnotation(Autor.class);

La expresión *miCoche.getClass()* devuelve un objeto de tipo *Class* que provee información sobre la clase a la que pertenece el objeto. De esta forma, estamos también obteniendo información sobre el método *acelerar()* usando la expresión *getMethod("acelerar")*. Concretamente, estamos accediendo a los datos de la anotación.

Autor a2 = miCoche.getClass().getAnnotation(Autor.class);

Aquí estamos accediendo a los datos de la anotación de la clase.

¿Qué son los frameworks ORM?

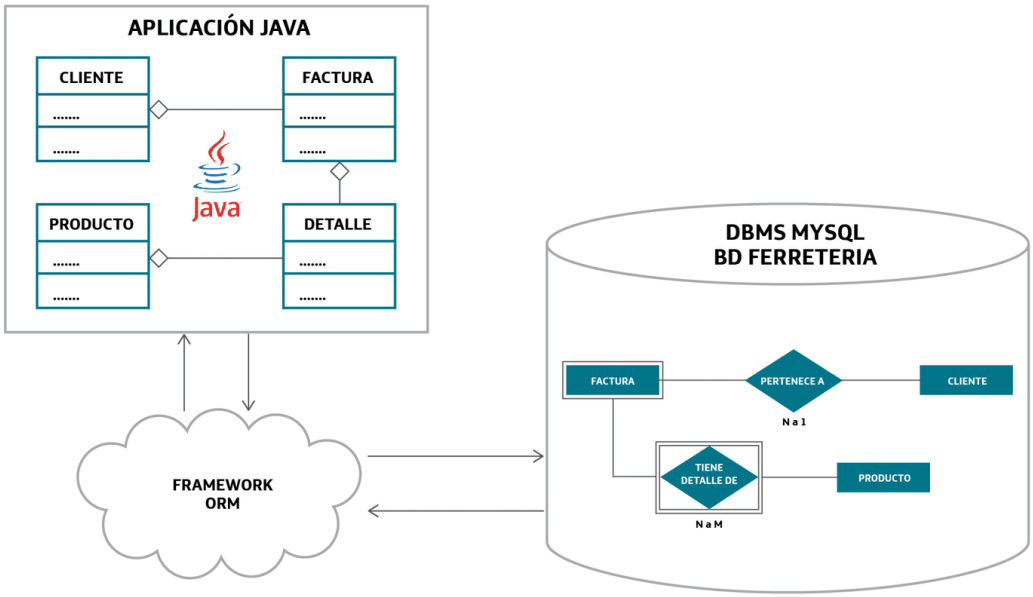
***Framework* significa *marco de trabajo*, y consiste en un conjunto de prácticas y normas que ayudan a solucionar un tipo de problema estandarizado.**

Los *frameworks* en el contexto de la ingeniería de software ofrecen, entre otras cosas, una **librería o API que da soporte completo para el desarrollo de un determinado tipo de aplicaciones**.

Pues bien, un ***Framework* ORM es un marco de trabajo que ofrece un soporte completo para el desarrollo de aplicaciones con bases de datos relacionales** y se basa en el procedimiento de Mapeo Objeto-Relacional.

Las siglas ORM vienen del inglés *Object Relational Mapping*

El F*ramework* ORM se sitúa en una capa intermedia entre las bases de datos y nuestra aplicación Java. Es el encargado de convertir las tablas físicas de la base de datos relacional en objetos Java que formarán una base de datos orientada a objetos virtual. La aplicación Java realizará las actualizaciones necesarias sobre los objetos Java y de nuevo será el *Framework* ORM el encargado de volcar dichos cambios en la base de datos física.



JPA (Java Persistence API)

Forma parte del estándar Java EE y está definida en el paquete *javax.persistence*. No se trata de un API completo, sino más bien de una especificación que requiere de una implementación. Varios productos comerciales implementan JPA; uno de ellos, el que vamos a utilizar en este curso, es *EclipseLink*.

Hibernate

Otro *Framework* ORM bastante extendido en la comunidad de desarrolladores Java, aunque también está disponible para la plataforma .NET. Fue desarrollado por la organización JBoss.

MyBatis

Otra herramienta ORM a cargo de Apache Tomcat Foundation.

Despedida

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- El **mapeo objeto-relacional** es un proceso a través del cual se crea, a partir de una base de datos relacional, un modelo de objetos que simula una base de datos orientada a objetos virtual. De esta manera, nuestra aplicación trabajará contra objetos Java en lugar de hacerlo directamente con la base de datos.
- Los ***Frameworks* ORM (*Object-Relational Mapping*)** son los encargados de llevar a cabo el mapeo objeto-relacional.
- Los *Frameworks* ORM utilizan **clases de entidad que cumplen la especificación de los *JavaBeans* y emplean anotaciones.**