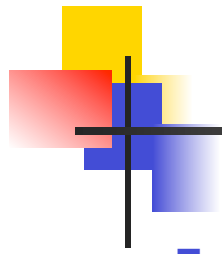




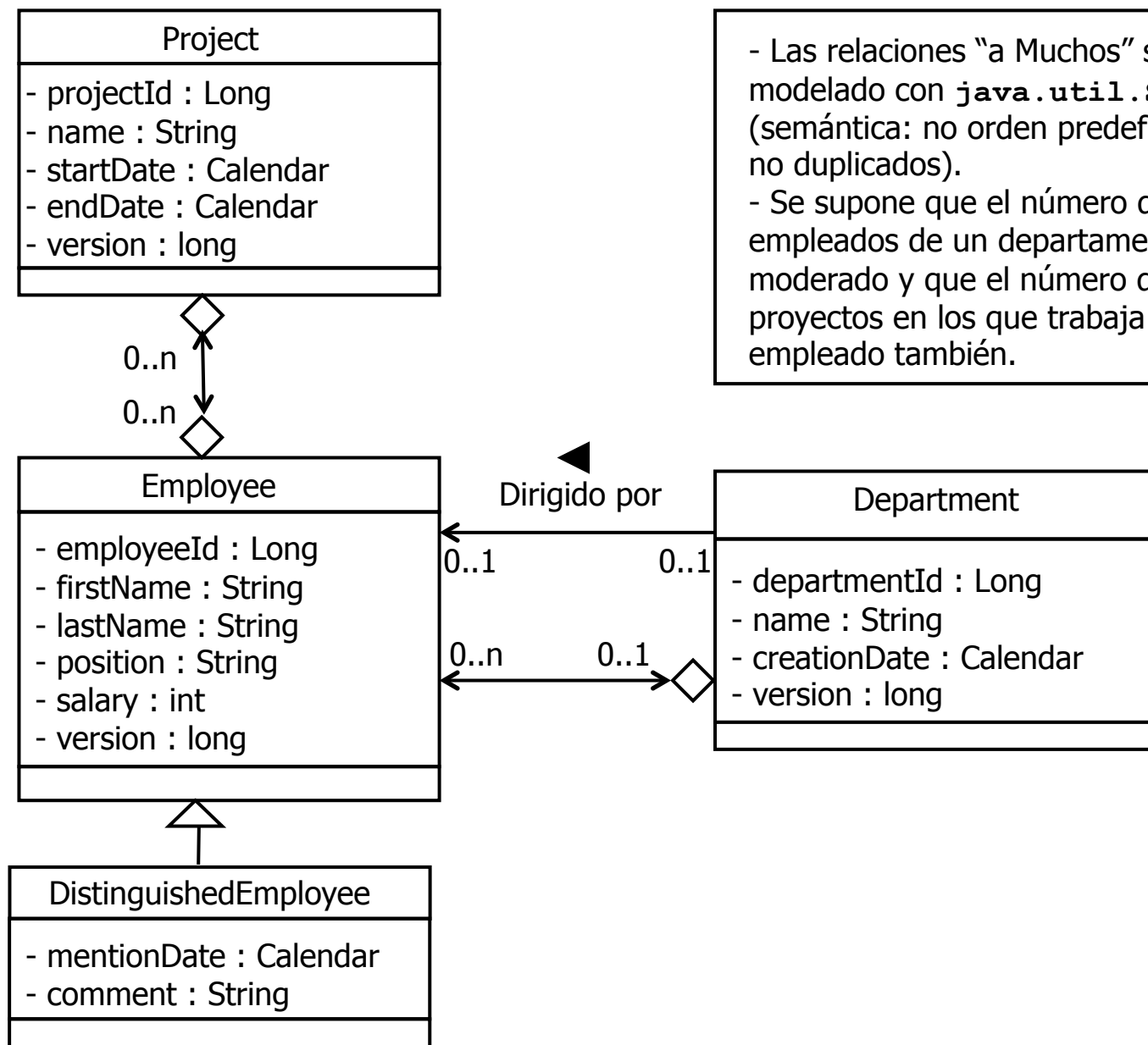
3.6 Conceptos Avanzados de Hibernate



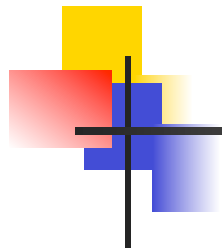
Introducción

- En este apartado se cubren los siguientes aspectos
 - Relaciones Muchos a Muchos
 - Herencia de entidades
 - Opcionalidad en las relaciones
 - Tipos de caché en Hibernate
 - Más sobre HQL
 - Optimizaciones de navegación entre entidades
 - Establecimiento de relaciones bidireccionales
 - Creación de objetos en cascada
 - Patrón “Open Session in View”
- Para ilustrar estos aspectos se utiliza como ejemplo la capa modelo de una aplicación de recursos humanos ficticia

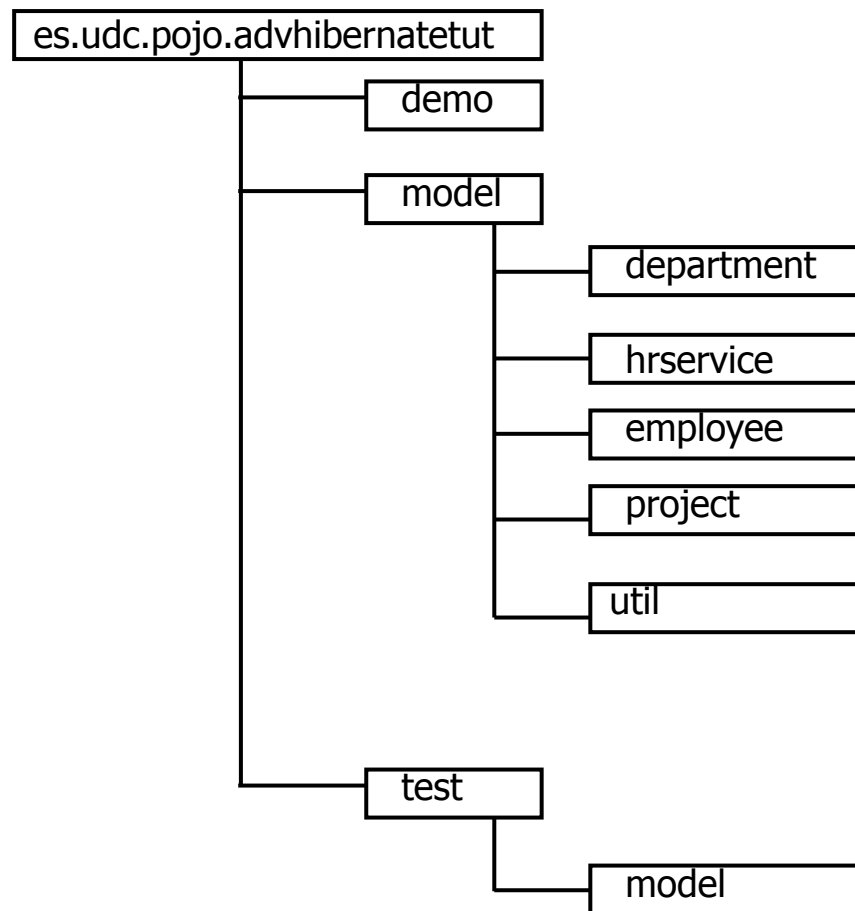
Entidades

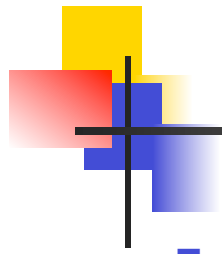


- Las relaciones "a Muchos" se han modelado con `java.util.Set` (semántica: no orden predefinido y no duplicados).
- Se supone que el número de empleados de un departamento es moderado y que el número de proyectos en los que trabaja un empleado también.



Estructura de paquetes (1)

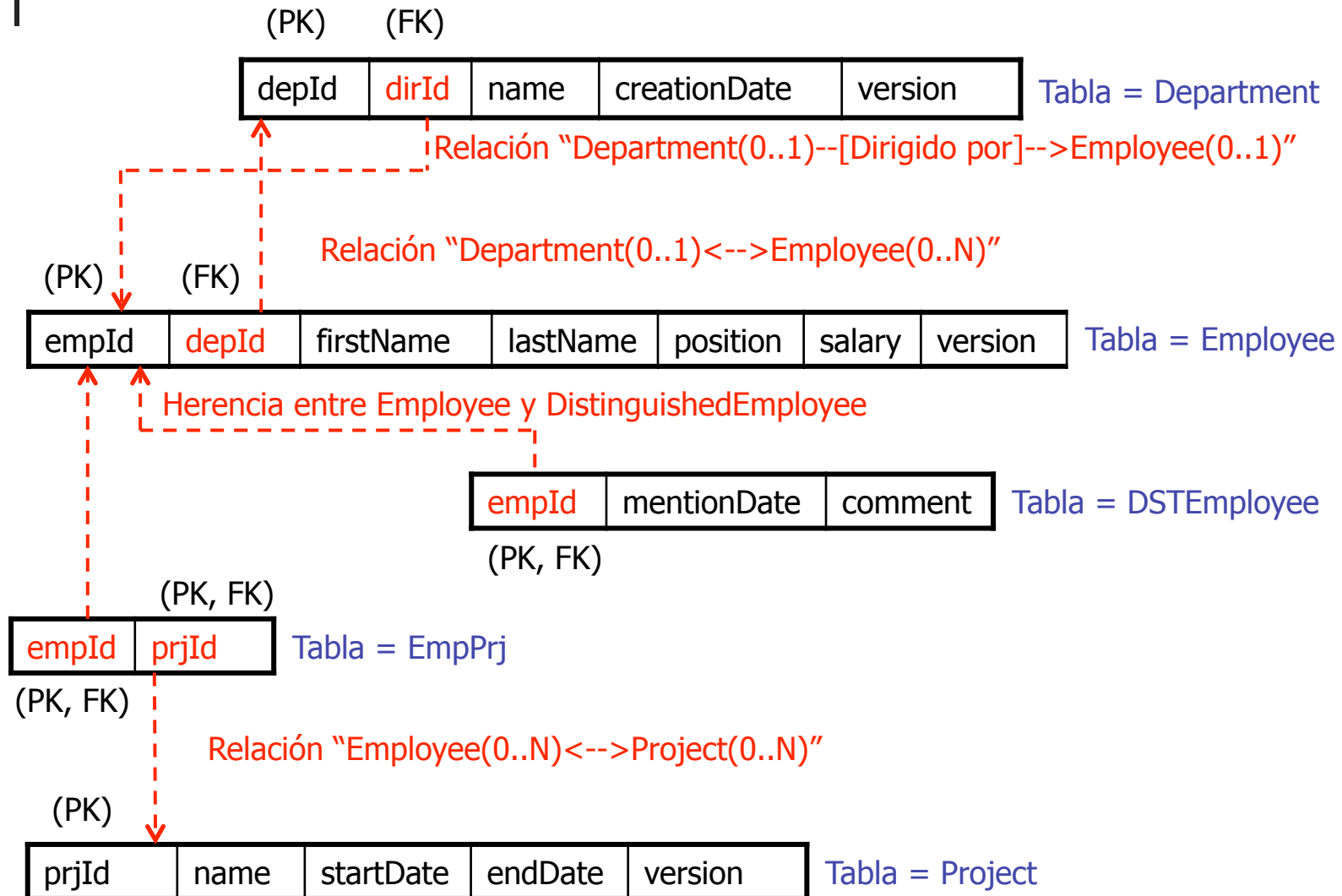




Estructura de paquetes (y 2)

- Además de las entidades, el ejemplo incluye
 - Un servicio (subpaquete **hrservice**) con operaciones para
 - Crear departamentos, empleados y proyectos
 - Hacer asignaciones (director de un departamento y empleado a un proyecto)
 - Realizar consultas
 - Realizar borrados en masa
 - Un sencillo cliente Web de prueba (subpaquete **demo**)
 - Dos casos de prueba (subpaquete **test**) para el servicio del modelo

Tablas





Relación "Employee(0..N)<-->Project(0..N)" (1)

■ En Employee

```
@Entity
public class Employee {

    // ...

    @ManyToMany
    @JoinTable(
        name="EmpPrj",
        joinColumns=@JoinColumn(name="empId"),
        inverseJoinColumns=@JoinColumn(name="prjId"))
    public Set<Project> getProjects() {
        return projects;
    }

    public void setProjects(Set<Project> projects) {
        this.projects = projects;
    }

    // ...
}
```



Relación "Employee(0..N)<-->Project(0..N)" (2)

- En Project

```
@Entity
public class Project {

    // ...

    @ManyToMany(mappedBy="projects")
    public Set<Employee> getEmployees() {
        return employees;
    }

    public void setEmployees(Set<Employee> employees) {
        this.employees = employees;
    }

    // ...

}
```




Relación "Employee(0..N)<-->Project(0..N)" (3)

- Relaciones Muchos-a-Muchos

- Se utiliza **@ManyToMany** sobre los campos/propiedades que definen la relación
 - Si la relación es unidireccional, el lado propietario es el que permite navegar hacia el otro
 - Si la relación es bidireccional, se puede elegir cualquier lado como propietario (y el otro será el lado inverso)
- En el ejemplo, la relación es bidireccional y se ha elegido **Employee** como lado propietario
- En **Project** (lado inverso)
 - **@ManyToMany(mappedBy="projects")** sobre **getEmployees**



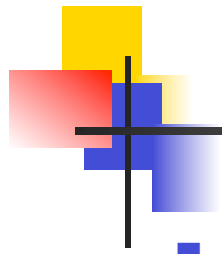
Relación "Employee(0..N)<-->Project(0..N)" (y 4)

- Relaciones Muchos-a-Muchos (cont)
 - En Employee (lado propietario), además de **@ManyToMany**, se utiliza **@JoinTable** sobre **getProjects**
 - **name**: nombre de la tabla en la que se mapea la relación
 - **joinColumns**: claves foráneas (normalmente una) que referencian las claves primarias de la tabla en la que se mapea la entidad del lado propietario
 - **inverseJoinColumns**: claves foráneas (normalmente una) que referencian las claves primarias de la tabla en la que se mapea la entidad del lado inverso



Estrategias de mapeo de herencia (1)

- El tipo enumerado **InheritanceType** define tres estrategias para mapear una relación de herencia
- **InheritanceType.SINGLE_TABLE**
 - Utiliza una única tabla que contiene una columna por cada campo/propiedad presente en las clases entidad
 - Se necesita incluir una columna que actúe como discriminador
 - Permite saber a qué entidad corresponde una fila
 - Ventaja: ejecución eficiente de consultas polimórficas
 - E.g. encontrar todos los empleados (del tipo que sean) que ocupen un determinado cargo
 - Se pueden implementar con una sola consulta sobre una única tabla
 - Desventaja: las columnas correspondientes a los campos/propiedades de las clases que extienden (directa o indirectamente) de la clase raíz tienen que admitir **NULL**
 - Quizás haya muchas filas con valor **NULL** para algunas de esas columnas



Estrategias de mapeo de herencia (y 2)

- **InheritanceType.TABLE_PER_CLASS**
 - Utiliza una tabla por cada entidad, que contiene una columna por cada campo/propiedad, propio o heredado, de la entidad
 - Desventaja: ejecución ineficiente de consultas polimórficas
 - Requiere lanzar consultas sobre cada tabla
- **InheritanceType.JOINED**
 - Utiliza una tabla por cada clase entidad, que contiene una columna por cada campo/propiedad específico a esa clase
 - La clave primaria de las tablas no raíz actúa como clave foránea de la clave primaria de la tabla raíz
 - Ventaja: ahorro de espacio y ejecución razonablemente eficiente de consultas polimórficas
 - Desventaja: requiere uno o varios JOINS para resolver las consultas polimórficas (prohibitivo en jerarquías profundas)
 - Esta es la opción que se ha elegido en el ejemplo



Herencia entre Employee y DistinguishedEmployee

■ En Employee

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Employee {
    // ...
}
```

■ En DistinguishedEmployee

```
@Entity
@Table(name="DSTEmployee")
public class DistinguishedEmployee extends Employee {
    // ...
}
```



Opcionalidad en las relaciones (1)

- Ejemplo: **Department**

- El modelo permite que un departamento no tenga un director asociado, e incluso que no tenga empleados
- Supongamos que recuperamos un departamento sin director asociado y sin empleados
- ¿Qué devuelve `getDirector()`?
 - Devuelve `null`
- ¿Qué devuelve `getEmployees()`?
 - Una colección (en este caso un `Set`) vacía

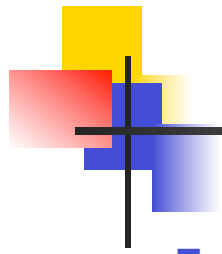


Opcionalidad en las relaciones (2)

- Ejemplo: **Department** (cont)

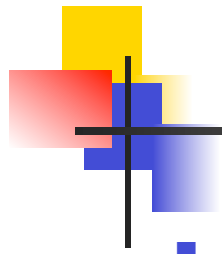
- En consecuencia:

```
public class Department {  
  
    private Long departmentId;  
    private String name;  
    private Calendar creationDate;  
    private Set<Employee> employees = new HashSet<Employee>();  
    private Employee director;  
    private long version;  
  
    public Department() {}  
  
    public Department(String name, Calendar creationDate) {  
        this.name = name;  
        this.creationDate = creationDate;  
  
    }  
}
```



Opcionalidad en las relaciones (y 3)

- Ejemplo: **Department** (cont)
 - El constructor sin argumentos es obligatorio para cualquier entidad
 - El segundo constructor es el que normalmente usará nuestro código para crear departamentos y refleja la semántica que hemos decidido (poder no tener asociado un director, e incluso no disponer de empleados)
 - En ambos casos, después de crear una instancia de **Department**, **director** es **null** y **employees** es un conjunto vacío
 - Posteriormente podemos asignar los empleados y designar el director invocando los métodos **set** correspondientes



Tipos de caché en Hibernate

- El uso de técnicas de caché puede reducir el número de consultas que se lanzan contra la BD
- Caché de primer nivel
 - Dentro de una transacción, Hibernate cachea los objetos que se recuperan de BD en la sesión (objeto **Session**)
 - Nunca se recupera más de una vez un mismo objeto
 - Sólo se cachean mientras dura la transacción
 - Este tipo de caché es implícita al modelo de Hibernate
- Caché de segundo nivel
 - Permite cachear el estado de objetos persistentes a nivel de proceso
 - Este tipo de caché es opcional y requiere configuración
- Caché de resultados de búsqueda
 - Permite cachear resultados de una búsqueda particular
 - Este tipo de caché está integrada con la caché de segundo nivel



Más sobre HQL – Aspectos generales (1)

- Literales

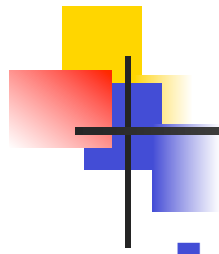
- Numéricos (enteros o reales)
 - Ejemplo: `e.salary >= 1000`
- Los literales de tipo cadena de caracteres se entrecomillan con comillas simples
 - Ejemplo: `d.departmentId = 'tic'`
 - Si el literal incluye una comilla simple, se sustituye por dos
 - Ejemplo: `d.name = 'uno' 'dos'`
- Literales booleanos: **TRUE** y **FALSE**

- Consultas polimórficas

- En la consulta ...

```
SELECT e FROM Employee e WHERE e.position = :position
```

- ... los empleados devueltos pueden ser empleados normales (instancias de **Employee**) o empleados distinguidos (instancias de **DistinguishedEmployee**)



Más sobre HQL – Aspectos generales (2)

■ Proyecciones

- Las consultas ilustradas en el apartado 3.3 devolvían entidades
- Es posible proyectar campos/propiedades o resultados agregados
 - Ejemplo: `SELECT d.name FROM Department d`
 - Ejemplo: `SELECT COUNT(d) FROM Department d`
 - NOTA: las funciones agregadas se explican más adelante
 - `Query.list` devuelve una lista de elementos, donde cada elemento es del tipo correspondiente (`String` en el primer ejemplo y `Long` en el segundo)
 - En las consultas lanzadas con `Query.uniqueResult`, el `Object` devuelto también es del tipo correspondiente



Más sobre HQL – Aspectos generales (3)

- Proyecciones (cont)

- También es posible proyectar varios elementos (entidades, campos/propiedades y resultados agregados)
 - Para cada departamento obtener su nombre y su fecha de creación

```
SELECT NEW es...DepartmentInfo(d.name, d.creationDate) FROM Department d
```

- Expresión constructor: después de la cláusula **NEW** se especifica un constructor de una clase
 - Se tiene que usar su nombre completo
 - La anterior consulta devuelve una lista con instancias de **DepartmentInfo** (una por cada departamento)
 - No es una clase entidad
 - Tiene que tener un constructor de tipo **DepartmentInfo(String, Calendar)**
 - Cada elemento devuelto por **Query.list**, o el único elemento devuelto por **Query.uniqueResult** (aunque en este ejemplo no es aplicable), es una instancia de **DepartmentInfo**



Más sobre HQL – Aspectos generales (4)

- Proyecciones (cont)

- Si se desea, no es necesario emplear expresiones constructor

```
SELECT d.name, d.creationDate FROM Department d
```

- Cada elemento devuelto por `Query.list`, o el único elemento devuelto por `Query.uniqueResult` (aunque en este ejemplo no es aplicable), es una instancia de `Object[]`
- Cada elemento del vector corresponde a un elemento (en el mismo orden que figura en la consulta)
- En el ejemplo
 - Cada elemento de la lista devuelta por `Query.list()` es una instancia de `Object[]`
 - Cada `Object[]` tiene dos elementos: el nombre y la fecha (en este orden)



Más sobre HQL – Aspectos generales (y 5)

- **ORDER BY**

- Al igual que en SQL, se puede ordenar por varios criterios simultáneamente, especificando opcionalmente **ASC** (por defecto) o **DESC**

- Ejemplos

- ```
SELECT e FROM Employee ORDER BY e.lastName, e.firstName
```

- ```
SELECT e FROM Employee ORDER BY e.lastName DESC,  
e.firstName DESC
```

- **Subconsultas**

- Es posible anidar subconsultas en las cláusulas **WHERE** y **HAVING**
- Iremos viendo ejemplos ...



Más sobre HQL – Expresiones condicionales (1)

- Operadores matemáticos (+, -, *, /), de comparación (=, >, <, >=, <=, <>) y lógicos (AND, OR, NOT)
 - Ejemplo

```
SELECT e FROM Employee e WHERE
    e.position = 'atp' AND e.salary >= 1000
```
 - Explicación
 - Obtener todos los empleados que ocupan el cargo de **atp** y su salario es **>= 1000**
- **[NOT] BETWEEN**
 - Ejemplo

```
SELECT e FROM Employee e WHERE e.salary BETWEEN 1000 AND 2000
```
 - Explicación

```
SELECT e FROM Employee e WHERE
    e.salary >= 1000 AND e.salary <= 2000
```



Más sobre HQL – Expresiones condicionales (2)

- **[NOT] IN**

- Ejemplo

- ```
SELECT d FROM Department d WHERE
d.departmentId IN (1, 2)
```

- Explicación

- ```
SELECT d FROM Department d WHERE  
d.departmentId = 1 OR d.departmentId = 2
```

- **[NOT] LIKE**

- Ejemplo

- ```
SELECT e FROM Employee e WHERE e.firstName LIKE 'F%o'
```

- Explicación

- Devuelve todos los empleados cuyo nombre empieza por **F** y termina en **o**

- Metacaracteres

- % (secuencia de 0 o más caracteres), \_ (cualquier carácter)
      - Se puede utilizar la cláusula **ESCAPE** para indicar un carácter de escape
      - Ejemplo: `e.firstName LIKE '%\_ %' ESCAPE '\'` devuelve **TRUE** para cualquier nombre que incluya un subrayado ()





## Más sobre HQL – Expresiones condicionales (3)

### ■ **IS [NOT] NULL**

#### ■ Ejemplo

```
SELECT d FROM Department d WHERE d.name IS NULL
```

#### ■ Explicación

- Devuelve todos los departamentos para los que no se ha especificado un valor para el campo **name**
- **IS [NOT] NULL** permite comprobar si un campo o colección es **NULL**

```
■ SELECT d FROM Department d WHERE d.director IS NULL
```

### ■ **IS [NOT] EMPTY**

#### ■ Ejemplo

```
SELECT d FROM Department d WHERE d.employees IS NOT EMPTY
```

#### ■ Explicación

- Devuelve todos los departamentos que tienen empleados
- **IS [NOT] EMPTY** permite comprobar si un campo o colección es vacío



## Más sobre HQL – Expresiones condicionales (4)

---

- **[NOT] EXISTS**

- Ejemplo

```
SELECT d FROM Department d WHERE
 EXISTS (SELECT e FROM Employee e WHERE
 e.position = :position AND e.department = d)
```

- Explicación

- Devuelve todos los departamentos que tengan al menos un empleado desempeñando un determinado cargo
- **EXISTS** devuelve **TRUE** si la subconsulta devuelve uno o más resultados



## Más sobre HQL – Expresiones condicionales (y 5)

---

- **ALL y ANY/SOME**

- Ejemplo

```
SELECT e FROM Employee e WHERE e.salary >= ALL
 (SELECT e.salary FROM Employee e)
```

- Explicación

- Devuelve todos los empleados que tengan el salario más alto

- **ALL**

- **TRUE** si la comparación es **TRUE** para todos los valores devueltos por la subconsulta, o si la subconsulta no devuelve ningún resultado

- **ANY/SOME**

- **TRUE** si la comparación es **TRUE** para alguno de los valores devueltos por la subconsulta (si la subconsulta no devuelve ningún resultado, la expresión es **FALSE**)
- **ANY** y **SOME** son sinónimos



## Más sobre HQL – Funciones no agregadas (1)

---

- Funciones de cadenas

- **CONCAT(*String*, *String*)**
  - Devuelve un *String* que es una concatenación de los dos pasados como parámetro
- **LENGTH(*String*)**
  - Devuelve el número (*int*) de caracteres del *String*
- **LOCATE(*String*, *String*, [*start*])**
  - Busca el segundo *String* en el primero
  - El tercer parámetro (opcional) indica la posición (de 1 en adelante) desde la que comenzar la búsqueda (por defecto, desde el primer carácter)
  - Devuelve la posición (*int*) en la que lo encontró (0 si no lo encontró)
- **SUBSTRING(*String*, *start*, *length*)**
  - Devuelve la subcadena (*String*) que comienza en la posición *start* y tiene longitud *length*



## Más sobre HQL – Funciones no agregadas (2)

---

- Funciones de cadenas (cont)

- **TRIM**

- Por defecto, **TRIM(String)** devuelve el **String** sin los blancos iniciales y finales

- **LOWER(String)**

- Devuelve el **String** en minúsculas

- **UPPER(String)**

- Devuelve el **String** en mayúsculas
    - Ejemplo: obtener los empleados cuyo nombre empieza por **F/f** y termina en **O/o**

```
SELECT e FROM Employee e WHERE UPPER(e.firstName) LIKE 'F%O'
```



## Más sobre HQL – Funciones no agregadas (y 3)

---

- Funciones aritméticas

- **ABS (number)**

- Valor absoluto de un `int`, `float` o `double`

- **SQRT (number)**

- Raíz cuadrada
    - Recibe un argumento numérico y devuelve un `double`

- **MOD (number, base)**

- Módulo
    - Recibe dos `int` y devuelve un `int`

- **SIZE (collection)**

- Tamaño de una colección
    - Devuelve un `int`
    - Ejemplo: obtener todos los departamentos que tienen empleados

```
SELECT d FROM Department d WHERE SIZE(d.employees) > 0
```



## Más sobre HQL – Funciones agregadas (1)

- Todas aceptan como argumento una expresión que haga referencia a un campo/propiedad no relación
- Adicionalmente, **COUNT** acepta como argumento una variable o una expresión que haga referencia a un campo/propiedad relación
- **AVG**
  - Calcula la media
  - Recibe un argumento numérico y devuelve un **Double**
  - Ejemplo: calcular el salario medio de los empleados (**EmployeeDaoHibernate**)

```
public float getAverageSalary() {
 Double averageSalary = (Double) getSession().createQuery(
 "SELECT AVG(e.salary) FROM Employee e").uniqueResult();
 return averageSalary.floatValue();
}
```



## Más sobre HQL – Funciones agregadas (2)

### ■ COUNT

- Devuelve el número (**Long**) de resultados
- Ejemplo: calcular el número de departamentos

```
SELECT COUNT(d) FROM Department d
```

### ■ MAX/MIN

- Calcula el valor máximo/mínimo
- Requiere un argumento de tipo numérico, **String**, **char** o fechas
- Ejemplo: obtener todos los empleados que tengan el salario más alto (**EmployeeDaoHibernate**)

```
public List<Employee> findByTheHighestSalary() {
 return getSession().createQuery(
 "SELECT e FROM Employee e WHERE e.salary >= " +
 "(SELECT MAX(e.salary) FROM Employee e) " +
 "ORDER BY e.lastName, e.firstName").list();
}
```





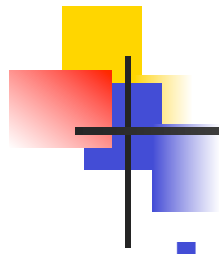
## Más sobre HQL – Funciones agregadas (y 3)

---

### ■ SUM

- Calcula la suma
- Devuelve **Long** si se aplica a enteros, **Double** si se aplica a reales y **BigInteger/BigDecimal** cuando se aplica a argumentos **BigInteger/BigDecimal**
- Ejemplo: calcula el salario total de los empleados

```
SELECT SUM(e.salary) FROM Employee e
```



# Más sobre HQL – JOIN (1)

- INNER JOIN implícito

- Producto cartesiano en la cláusula **FROM** + condición en la cláusula **WHERE**
- Ejemplo: obtener todos los departamentos que tengan al menos un empleado desempeñando un determinado cargo

```
SELECT DISTINCT d FROM Department d, Employee e WHERE
e.department = d AND e.position = :position
```

- NOTAS:

- Equivalente a la consulta SQL

```
SELECT DISTINCT d.* FROM Department d, Employee e
WHERE e.depId = d.depId AND e.position = 'XXX'
```

- `e.department = d` es equivalente a `e.department.departmentId = d.departmentId`
- **DISTINCT**: mismo significado que en SQL (evita que se repitan departamentos cuando hay más de un empleado en un mismo departamento desempeñando el cargo especificado)



## Más sobre HQL – JOIN (2)

- INNER JOIN explícito

- Usa la cláusula `[INNER] JOIN`

- Ejemplo: el anterior

- `DepartmentDaoHibernate -> findByEmployeeInPosition`

```
SELECT DISTINCT d
 FROM Department d JOIN d.employees e
 WHERE e.position = :position ORDER BY d.name
```

- Equivalente a la consulta SQL

```
SELECT DISTINCT d.* FROM Department d JOIN Employee e
 ON e.depId = d.depId WHERE e.position = 'XXX' ORDER BY d.name
```

- Con respecto a la consulta SQL, la sintaxis HQL evita la condición sobre las claves (entre otras cosas)
- Con respecto a un INNER JOIN implícito, el uso de la cláusula `JOIN` evita la condición `e.department = d` (equivalente a la condición sobre las claves)



# Más sobre HQL – JOIN (y 3)

---

- También existen **LEFT [OUTER] JOIN** y **RIGHT [OUTER] JOIN**
- Otros ejemplos
  - Obtener todos los proyectos en los que trabaja un empleado
    - `ProjectDaoHibernate -> findByEmployeeId`

```
SELECT p FROM Project p JOIN p.employees e WHERE
e.employeeId = :employeeId ORDER BY p.name
```

- Obtener todos los proyectos en los que trabaja un departamento
  - `ProjectDaoHibernate -> findByDepartmentId`

```
SELECT DISTINCT p FROM Project p JOIN p.employees e
JOIN e.department d WHERE d.departmentId = :departmentId
ORDER BY p.name
```



## Más sobre HQL – GROUP BY, HAVING (1)

- Similares a las cláusulas SQL
  - **GROUP BY** forma grupos en función de uno o varios campos/propiedades
  - **HAVING** (opcional) permite especificar una condición que tienen que cumplir los grupos

- Ejemplo

```
SELECT e.department.name, AVG(e.salary)
FROM Employee e
GROUP BY e.department.departmentId
HAVING COUNT(e) >= 2
```

- Explicación

- Devuelve el nombre y salario medio de los departamentos que tengan al menos dos empleados



## Más sobre HQL – GROUP BY, HAVING (2)

- Otro ejemplo

- Obtener estadísticas para cada departamento
  - `DepartmentDaoHibernate -> getStatistics`
- Los datos estadísticos de cada departamento incluyen: nombre del departamento, número de empleados, salario medio, salario mínimo, salario máximo y salario total

```
SELECT NEW es.udc...DepartmentStatistics (
 d.name, COUNT(e), AVG(e.salary),
 MIN(e.salary), MAX(e.salary), SUM(e.salary))
FROM Department d LEFT JOIN d.employees e
GROUP BY d.departmentId
ORDER BY d.name
```

- NOTAS

- El uso de **LEFT JOIN** es necesario porque queremos obtener datos de **todos** los departamentos (aunque no tengan empleados)
- Para los departamentos sin empleados, **AVG**, **MIN**, **MAX** y **SUM** devuelven **null**



## Más sobre HQL – GROUP BY, HAVING (y 3)

- Otro ejemplo (cont)
  - Constructor de `DepartmentStatistics`

```
public DepartmentStatistics(String name,
 Long numberOfEmployees, Double averageSalary,
 Integer minimumSalary, Integer maximumSalary, Long totalSalary) {

 this.name = name;
 this.numberOfEmployees = numberOfEmployees;
 this.averageSalary =
 averageSalary == null ? 0 : averageSalary.floatValue();
 this.minimumSalary =
 minimumSalary == null ? 0 : minimumSalary;
 this.maximumSalary =
 maximumSalary == null ? 0 : maximumSalary;
 this.totalSalary =
 totalSalary == null ? 0 : totalSalary;

}
```

- Para borrados o actualizaciones individuales
  - Borrado de una entidad
    - `GenericDaoHibernate -> remove`
  - Actualización de una entidad
    - Recuperar la entidad
    - Modificar los campos/propiedades
    - Antes de terminar la ejecución del caso de uso, Hibernate actualiza en BD (aunque, por convención, nuestro código invocará explícitamente el método `update` del DAO)
- ¿Y si queremos eliminar/actualizar un conjunto (potencialmente) grande de entidades?
  - Con un enfoque básico, sería necesario localizar las entidades y borrar/actualizar cada una individualmente
  - Problema: gran número de consultas
    - Una para localizar las entidades y otra para borrar/actualizar cada una



- Para resolver este problema, HQL proporciona las sentencias **DELETE** y **UPDATE**, que permiten borrados y actualizaciones en masa

- Ejemplos

- Eliminar todos los proyectos

```
DELETE FROM Project
```

- Eliminar todos los empleados con cargo **atp**

```
DELETE FROM Employee e WHERE e.position = 'atp'
```

- Subirle el sueldo (en 100) a todos los empleados **atp**

```
UPDATE Employee e SET e.salary = e.salary + 100
WHERE e.position = 'atp'
```

- Ejecución de sentencias **UPDATE** y **DELETE**

- Mediante `Query.executeUpdate`
  - Devuelve el número de entidades borradas/actualizadas
- Ejemplo: en `ProjectDaoHibernate`

```
public void removeAll() {
 getSession().createQuery("DELETE FROM Project").
 executeUpdate();
}
```

- Las sentencias **UPDATE** y **DELETE** actúan directamente contra la BD
  - NOTA: En el fichero .sql de creación de tablas del ejemplo se usa **DELETE CASCADE** para que
    - Cuando se elimine un departamento, se eliminen sus empleados
    - Cuando se elimine un empleado, se eliminen las filas asociadas en `EmpPrj`
    - Cuando se elimine un proyecto, se eliminen las filas asociadas en `EmpPrj`

- Las sentencias **UPDATE** y **DELETE** actúan directamente contra la BD (cont)
  - Se ignora la sesión de Hibernate (caché de primer nivel), y en consecuencia no se tienen en cuenta los objetos que estén cargados en ella
  - Por defecto, la sentencia **UPDATE** no actualiza el campo especificado con **@Version**
    - Si se desea que lo haga, es necesario emplear **VERSIONED**
    - Ejemplo:

```
UPDATE VERSIONED Employee e SET e.salary = e.salary + 100
WHERE e.position = 'atp'
```
  - Un enfoque sencillo para evitar problemas con la sesión de Hibernate consiste en que los casos de uso que utilicen este tipo de sentencias, o bien sólo lancen estas sentencias, o bien las lancen al principio (antes de haber usado los métodos de **Session**)



## Optimizaciones de navegación entre entidades (1)

---

- Para evitar que cuando un objeto se recupera de BD, se recuperen también todos los relacionados con éste, la documentación de Hibernate aconseja, en general, el uso de **FetchType.LAZY** en todo tipo de relaciones
  - En **@OneToMany** y **@ManyToMany** el valor por defecto de **fetch** es **FetchType.LAZY**
  - En **@OneToOne** y **@ManyToOne** el valor por defecto de **fetch** es **FetchType.EAGER**
  - El código de pojo-examples, siempre especifica **fetch=FetchType.LAZY** para **@OneToOne** y **@ManyToOne**
- De esta manera, por ejemplo, se evita que cuando se recupere un departamento, se recuperen también su director y sus empleados
  - Y dado que los empleados también utilizan la estrategia LAZY en la relación con los proyectos, también se evita que se carguen los proyectos asociados a cada empleado



## Optimizaciones de navegación entre entidades (2)

- Sin embargo, el uso de LAZY sin ningún tipo de optimización adicional puede conducir al “problema de las n+1 consultas”
- Ejemplo: supongamos un caso de uso cuya lógica consiste en
  - `List<Department> departments = departmentDao.findAll(); // (1)`
  - Por cada departamento “dep” en “departments”
    - `Employee director = dep.getDirector();`
    - Si `director != null => procesar(director.getFirstName()); // (2)`
- Los objetos **Department** devueltos en la línea (1) contienen un proxy de **Employee** en el atributo **director** y una colección especial de Hibernate en el atributo **employees**



## Optimizaciones de navegación entre entidades (3)

- Los proxies los genera Hibernate dinámicamente en tiempo de ejecución y son clases que extienden a la clase original
  - En el caso de un proxy de una entidad que herede de otra, el proxy extiende a la clase raíz
  - NOTA: Observar el código de `equals` en `DistinguishedEmployee`
- Cuando se invocan operaciones sobre un proxy no inicializado o una colección no inicializada, Hibernate inicializa el proxy o la colección
  - Un proxy siempre tiene el atributo clave correctamente inicializado => `department.getDirector().getEmployeeId()` no causa que se cargue el objeto
- Cuando se ejecuta la línea (2), Hibernate inicializa el proxy del empleado director de ese departamento => lanza una consulta para recuperar los datos de ese empleado director
- Número de consultas lanzadas en el ejemplo:  $1 + n$ 
  - 1 para recuperar los departamentos
  - 1 para recuperar los datos de cada director

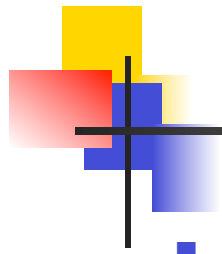


## Optimizaciones de navegación entre entidades (4)

- Existen varias estrategias para minimizar el número de consultas que se lanzan
- En el ejemplo se ha empleado

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@org.hibernate.annotations.BatchSize(size = 10)
public class Employee {
 // ...
}
```

- La anotación **@BatchSize** sobre **Employee** provoca que cuando se ejecute la línea (2), no sólo se inicialice el primer proxy, sino hasta un máximo de 10 proxies lanzando una única consulta



## Optimizaciones de navegación entre entidades (y 5)

- La consulta que se lanza es del tipo
  - `SELECT * FROM Employee e WHERE e.empId IN (A, B, C, ...)`
  - A, B, C, ... son las claves de los 10 primeros proxies sin inicializar
  - NOTA: en realidad la consulta que se lanza en este caso hace un JOIN con la tabla `DSEmployee` (debido a la relación de herencia)
- Si en la lista de departamentos hay más de 10, cuando se accede a `director.getFirstName()` en el undécimo departamento, se lanza otra consulta similar para los 10 siguientes proxies
- El uso de `@BatchSize` sobre una entidad es válido para la inicialización de proxies que se refieren a ella
- Hibernate dispone de otras anotaciones aplicables a la inicialización de colecciones y optimización de determinadas operaciones sobre colecciones
  - `org.hibernate.annotations.FetchMode(FetchType.SUBSELECT)`
  - `org.hibernate.annotations.LazyCollections(LazyCollectionOption.EXTRA)`





## Establecimiento de relaciones bidireccionales (1)

- Ejemplo **erróneo**: Relación Department(0..1) <--> Employee(0..N)

- En HrServiceImpl

```
public void createEmployee(Employee employee, Long departmentId)
 throws InstanceNotFoundException {

 Department department =
 (Department) departmentDao.find(departmentId);

 department.getEmployees().add(employee); // (1)
 employeeDao.create(employee)

}
```

- El empleado se crea, pero no queda asociado al departamento
- El comportamiento es consistente con el modelo de objetos en memoria
- Después de ejecutarse la línea (1), `department.employees` contiene al nuevo empleado, pero `employee.department` es null



## Establecimiento de relaciones bidireccionales (2)

---

- Hibernate no gestiona automáticamente el establecimiento de relaciones bidireccionales
- Para evitar problemas (el descrito anteriormente y otros relacionados), es buena práctica establecer las relaciones bidireccionales explícitamente de manera sistemática
- Ejemplo corregido

- En `Department`

```
public void addEmployee(Employee employee) {
 getEmployees().add(employee);
 employee.setDepartment(this);
}
```

- El uso de `getEmployees`, frente al uso directo del atributo `employees`, es necesario porque si `addEmployee` se aplica sobre un `Department` que es un proxy, `employees` podría estar sin inicializar



## Establecimiento de relaciones bidireccionales (3)

- Ejemplo corregido (cont)

- En HrServiceImpl

```
public void createEmployee(Employee employee, Long departmentId)
 throws InstanceNotFoundException {

 Department department =
 (Department) departmentDao.find(departmentId);

 department.addEmployee(employee);
 employeeDao.create(employee);

}
```



## Establecimiento de relaciones bidireccionales (y 4)

- Ejemplo `Employee(0..N) <--> Project(0..N)`

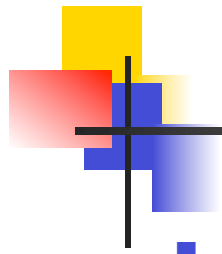
- En `Employee`

```
public void addProject(Project project) {
 getProjects().add(project);
 project.getEmployees().add(this);
}
```

- En `Project`

```
public void addEmployee(Employee employee) {
 getEmployees().add(employee);
 employee.getProjects().add(this);
}
```

- Para establecer una relación entre un empleado y un departamento basta con hacer  
`employee.addProject(project)` o  
`project.addEmployee(employee)`

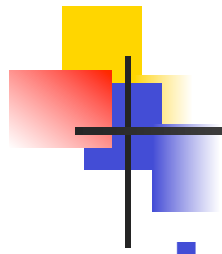


# Creación de objetos en cascada (1)

- La clase `es.udc.pojo.advhibernatetut.demo.DemoUtil` juega un papel parecido a las clases `DbUtil` del módulo "test" de MiniBank y MiniPortal
  - En particular dispone de los métodos `createDemoData` y `removeDemoData`
- En `createDemoData`, para crear un departamento y sus empleados se podría proceder de la siguiente manera

```
es = new Department(...);
arr = new Employee(...);
joo = new Employee(...);
```

```
es.addEmployee(arr);
es.addEmployee(joo);
es.setDirector(joo);
departmentDao.create(es);
employeeDao.create(arr);
employeeDao.create(joo);
```

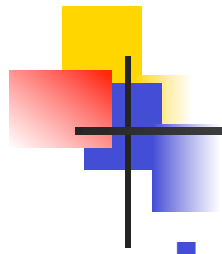


## Creación de objetos en cascada (2)

- Con el objetivo de ilustrar el soporte para operaciones en cascada, en `Department` se usa

```
@OneToMany(mappedBy = "department", cascade = CascadeType.PERSIST)
public Set<Employee> getEmployees() {
 return employees;
}
```

- El elemento `cascade` permite especificar una o un conjunto de operaciones que se aplicarán a las entidades relacionadas
- Con `CascadeType.PERSIST`, si se aplica `session.persist` (`GenericDaoHibernate -> create`) a un departamento, se aplicará también a los empleados asociados
- NOTA: no hace falta usar `cascade` en `@OneToOne` sobre `getDirector` dado que el director es también un empleado



## Creación de objetos en cascada (3)

---

- Ahora para crear un conjunto de departamentos y empleados es posible simplificar el código anterior

```
es = new Department(...);
arr = new Employee(...);
joo = new Employee(...);
```

```
es.addEmployee(arr);
es.addEmployee(joo);
es.setDirector(joo);
departmentDao.create(es);
```



# Creación de objetos en cascada (y 4)

---

- Creación de un pedido

```
Order order = new Order(...);
OrderLine orderLine1 = new OrderLine(...);
OrderLine orderLine2 = new OrderLine(...);
order.addOrderLine(orderLine1);
order.addOrderLine(orderLine2);
orderDao.create(order);
```





# Patrón "Open Session In View" (1)

- Consideremos el método (caso de uso) **findAllDepartments** en **HrService**
  - Por sencillez no se utiliza el patrón Page-by-Page Iterator

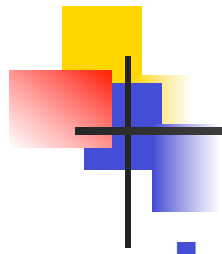
```
public List<Department> findAllDepartments();
```

- La implementación en **HrServiceImpl**

```
public List<Department> findAllDepartments() {
 return departmentDao.findAll();
}
```

- Y finalmente, la implementación de **findAll** en **DepartmentDaoHibernate**

```
public List<Department> findAll() {
 return getSession().createQuery("SELECT d FROM Department d " +
 "ORDER BY d.name").list();
}
```



## Patrón "Open Session In View" (2)

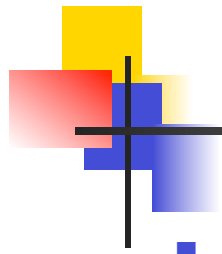
- Supongamos que el caso de uso que muestra los datos de todos los departamentos tiene que visualizar para cada departamento
  - Sus datos básicos (identificador, nombre y fecha de creación)
  - Nombre y apellidos de su director
- En consecuencia, cuando el usuario ejecuta este caso de uso, la interfaz gráfica ejecuta
  - `List<Department> departments = hrService.findAllDepartments();`
  - Por cada departamento "dep" en "departments"
    - `Imprimir dep.getDepartmentId();`
    - `Imprimir dep.getName();`
    - `Imprimir dep.getCreationDate();`
    - `Employee director = dep.getDirector();`
    - Si `director != null =>`
      - `Imprimir director.getFirstName(); // (*)`
      - `Imprimir director.getLastName();`



## Patrón "Open Session In View" (3)

---

- La línea (\*) provoca **`org.hibernate.LazyInitializationException`**
  - **`director`** es un proxy sin inicializar
    - Porque la relación entre **`Department`** y su **`Employee`** director es LAZY
  - No se puede inicializar porque no se dispone de conectividad a la BD (la transacción terminó y la conexión a la BD se cerró)
  - Lo mismo ocurriría si obtuviésemos la lista de empleados (**`department.getEmployees()`**) e intentásemos acceder a sus datos (la colección estaría sin inicializar)



## Patrón “Open Session In View” (4)

- En el caso de aplicaciones Web, se suele utilizar el patrón “Open Session in View” para hacer frente a este problema
- La idea global del patrón es que cada vez que llega una petición HTTP se abre una sesión de Hibernate que no se cierra hasta que termina de procesarse la petición
- Existen varias variantes del patrón que difieren en el momento que se hace el commit/rollback
- En particular nosotros utilizaremos una implementación del patrón proporcionada por Spring (se mostrará en el apartado 4.3) que
  - Hace commit/rollback al terminar la ejecución del método del servicio invocado
  - Mantiene una sesión Hibernate abierta (hasta que termina la petición HTTP) para poder acceder a la BD, lo que permite acceder a proxies o colecciones de proxies de manera segura una vez invocado un caso de uso



# Patrón "Open Session In View" (5)

- Una alternativa a este patrón es el uso de objetos a medida (no entidades) que devuelvan toda la información que se precise

- En **HrService**

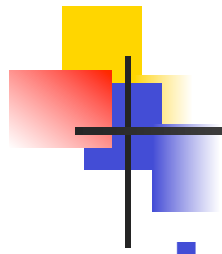
```
public List<DepartmentInfo> findAllDepartments();
```

- Y su implementación en **HrServiceImpl**

```
public List<DepartmentInfo> findAllDepartments() {
 return departmentDao.findAll();
}
```

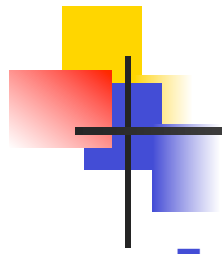
- Y finalmente, la implementación de **findAll** en **DepartmentDaoHibernate**

```
public List<DepartmentInfo> findAll() {
 return getSession().createQuery(
 "SELECT NEW es...DepartmentInfo(d.departmentId, d.name, " +
 "d.creationDate, d.director.firstName, d.director.lastName) " +
 "FROM Department d " +
 "ORDER BY d.name").list();
}
```



## Patrón “Open Session In View” (6)

- El uso de objetos a medida es una solución muy pura
  - Con Open Session in View, la capa modelo asume que la interfaz gráfica podrá navegar por las relaciones de entidades aunque éstas estén sin inicializar
- ... pero es muy tediosa para aplicaciones Web convencionales
- En el caso de aplicaciones Web en el que la capa modelo corra en el mismo proceso que la interfaz gráfica, Open Session in View representa una solución más ágil
- Cuando la interfaz gráfica está en un proceso distinto de la capa modelo (e.g. la capa modelo es un servicio Web), no es posible usar el patrón “Open Session in View”
  - La sesión existe en el proceso que ejecuta la capa modelo
  - En este caso, el uso de objetos a medida constituye una buena solución



## Patrón "Open Session In View" (7)

- Es posible experimentar con el uso de este patrón ejecutando
  - `cd pojo-advhbernattetut`
  - `mvn jetty:run`
  - `http://localhost:9090/pojo-advhbernattetut/DemoServlet?command=createDemoData`
  - `http://localhost:9090/pojo-advhbernattetut/DemoServlet?command=findAllDepartments`
  - Los departamentos se visualizan correctamente
  - En `src/main/webapp/WEB-INF/web.xml`, comentar las líneas

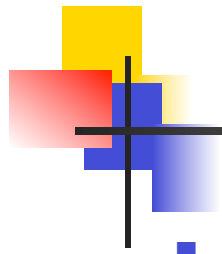
```
<filter-mapping>
```

```
 <filter-name>openSessionInView</filter-name>
```

```
 <url-pattern>/*</url-pattern>
```

```
</filter-mapping>
```

- `http://localhost:9090/pojo-advhbernattetut/DemoServlet?command=findAllDepartments` -> `LazyInitializationException`



# Patrón “Open Session In View” (y 8)

- Obsérvese que el uso de la anotación `@BatchSize(size=10)` sobre `Employee` provoca que la ejecución de `http://localhost:9090/pojo-advhibernatetut/DemoServlet?command=findAllDepartments` lance dos consultas
  - Una para recuperar todos los departamentos
  - Otra para recuperar los directores de los departamentos (en el ejemplo hay menos de 10 departamentos)
- Observación: pruebas de integración
  - Al usar `@Transactional` en las clases de prueba, es posible, dentro de un caso de prueba (métodos `@Test`), navegar por las entidades devueltas por el caso de uso que se prueba, dado que la sesión sigue abierta
    - Más aún, en este caso, la sesión no sólo sigue abierta después de invocar al caso de uso, sino que la transacción sigue activa hasta que el caso de prueba termine (y finalmente se hace `rollback`)