Taller de Persistencia con Java

Encuentro 4:

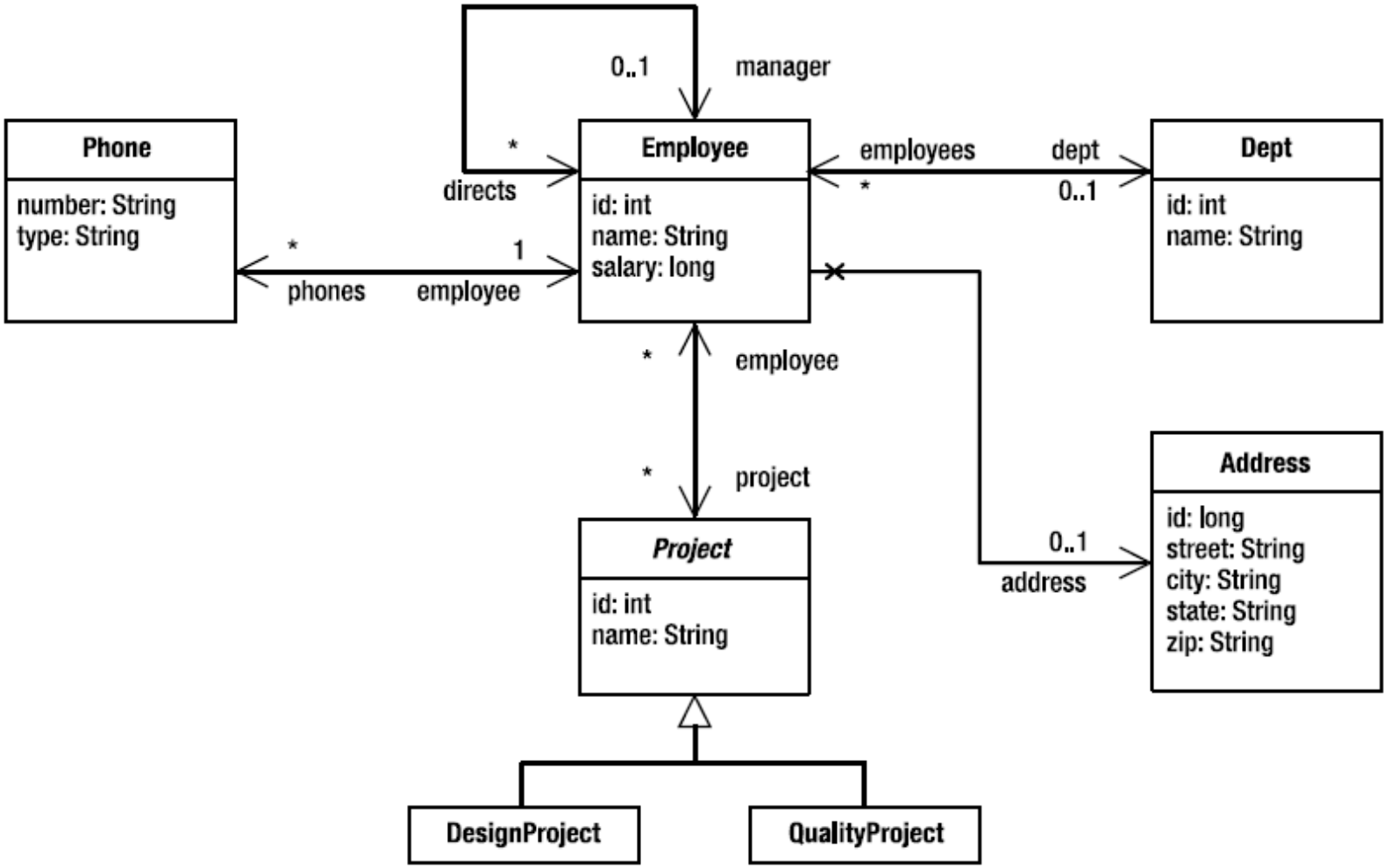
“Consultas”

JPQL

- Java Persistence Query Language (JPQL), es un lenguaje de consultas portable diseñado para combinar la sintaxis simple y la potencia semántica de SLQ, con la expresividad de las expresiones orientadas a objetos.
- Es basado en el lenguaje “EJB Query Language (EJB QL) introducido en la versión de EJB 2.0
- Las consultas escritas en este lenguajes pueden ser portables y compiladas a SQL propietario para ser ejecutados en la mayoría de los RDBMS.
- **JPQL NO ES SQL**

JPQL

- No se debe intentar escribir consultas JPQL pensando en SQL. Esto provoca muchos problemas y errores.
- JPQL es un lenguaje de consultas de ENTIDADES.
- Podemos expresar las consultas en términos de objetos y sus relaciones, definidas en el modelo abstracto de objetos.
- Si JPA permite usar consultas SQL nativas, ¿porque usar otro lenguaje?
  - Asegura portabilidad
  - Se trabaja contra el modelo y no con tablas


0..1
manager
*
directs
Phone
number: String
type: String
Employee
id: int
name: String
salary: long
employees
*
dept
0..1
Dept
id: int
name: String
*
phones
1
employee
*
employee
*
project
Project
id: int
name: String
0..1
address
Address
id: long
street: String
city: String
state: String
zip: String
DesignProject
QualityProject

Consultas de selección

SELECT <select_expression>

FROM <from_clause>

[WHERE <conditional_expression>]

[ORDER BY <order_by_clause>]

Ejemplo:

SELECT e FROM Employee e

Esta consulta retorna una lista con una instancia de Employee por cada registro de empleado.

Internamente la consulta JPQL es traducida a SQL, ej:

SELECT id, name, salary, manager_id, dept_id, address_id FROM emp

La estructura es parecida a SQL, pero posee diferencias importantes.

En primer lugar la consulta no se realiza sobre una tabla sino sobre una entidad persistente.

El alias (variable identificatoria)es obligatoria (en SQL es opcional).

El SELECT no enumera los campos.

El resultado es una lista de instancias.

Path Expressions

- Las “path expressions” son utilizadas para navegar a través de:
  - las relaciones de las entidades
  - los atributos persistentes de una entidad.
- Se representan con un “.”
- Si la navegación nos dirige a una entidad única, se denomina “*single-valued association path*”, y si es un conjunto de entidades “*collection-valued association path*”.
- Ejemplo: *SELECT e.department.name FROM Employee e*

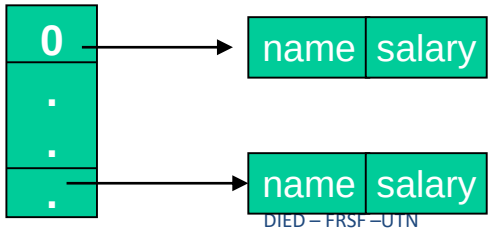
Tipos de resultados.

- La consulta *"SELECT d FROM Department d"* retorna una lista de instancias de departamentos.
- "Path expressions" para acceder a un atributo persistente. Ejemplo: *"SELECT e.name FROM Employee e"*
  - En este caso se están retornando una lista de String.

- En el caso de las relaciones "*single-valued association path*", se devuelve una lista de instancias del objeto seleccionado:
  - "SELECT e.department FROM Employee e".
- En este caso se retorna con duplicados.

Tipos de resultados.

- Si se desea sin duplicados
  - →"SELECT DISTINCT e.department FROM Employee e" *(se utiliza la PK de la entidad)*
- En el caso de las relaciones "*collection-valued association path*" es ilegal que aparezcan en la cláusula SELECT como valor de retorno.
  - EJ: "SELECT d.employees FROM Department d"
- "**Proyección**": es posible combinar múltiples expresiones para obtener un subconjunto de atributos en un arreglo de Objetos como resultado para cada fila encontrada:
  - *"SELECT e.name, e.salary FROM Employee e"*



Expresión de Constructor.

- Una forma mas “potente” de selección es la “expresión de constructor” → Construir un objeto en la cláusula SELECT.
- El resultado es una lista de instancias de la clase cuyo constructor indicada en el Select
  - SELECT NEW example.EmployeeDetails(e.name, e.salary, e.department.name) FROM Employee e
- El constructor debe invocarse con el nombre completo de la clase.
- La clase debe tener definido un constructor con los tipos especificados.
- Útiles para construir objetos de granularidad mas “gruesa”

Herencia y Polimorfismo

- JPA soporta herencia entre entidades, y en consecuencia los resultados pueden ser polimórficos cuando se invocan múltiples subclases de una clase en la misma consulta.
- Ejemplo: Project es superclase para QualityProject and DesignProject. Si una variable de identificación de una consulta se asocia a la superclase, entonces los resultados van a incluir un mix de instancias de ambas subclases.
  - Para procesar los resultados puede ser necesario realizar casting.

- Ejemplo consulta:
  - SELECT p FROM Project p WHERE p.employees IS NOT EMPTY

TREAT – Nuevo en JPA 2.1

- La función TREAT permite realizar un downcast, y así realizar una consulta sobre la clase base, pero tratar en alguna condición al objeto como una clase hija.

```
1. SELECT b.name, b.ISBN
2. FROM Order o JOIN TREAT(o.product AS Book) b

3. SELECT b.name
4. FROM User e JOIN TREAT(e.projects AS LargeProject) b

5. SELECT e FROM Employee e
6. WHERE TREAT(e AS Exempt).vacationDays > 10
7. OR TREAT(e AS Contractor).hours > 100
```

Cláusula FROM

- La cláusula FROM permite definir una o más variables de identificación para cada una de las entidades consultadas.
  - Cada consulta debe tener al menos una variable de identificación.
  - Son insensibles a Mayúsculas/minúsculas.
- Join: consulta que combina resultados de múltiples entidades (similar a SQL). Ocurre cuando:
  - Dos o más variables son listadas en un FROM con una cláusula WHERE
  - Se utiliza el operador JOIN sobre una path expresión en una variable de identificación.
  - Cuando con una “path expression” se navega a través de una relación en cualquier parte de la consulta.
  - ***(inner join – outer join – producto cartesiano)***

Inner Joins – Collection Association Fields

- Ejemplo: *"SELECT p FROM Employee e JOIN e.phones p"*
- Se realiza un "join" entre Employee y Phone, a través de la relación que los une.
  - SQL: *SELECT p.id, p.phone_num, p.type, p.emp_id FROM emp e, phone p WHERE e.id = p.emp_id*
- La condición de JOIN se define en la relación de mapeo entre las entidades, no se necesitan criterios adicionales.
- Esta consulta retornará todas las instancias de "Phone" asociadas a un "Employee".
- Cada ocurrencia de la variable de ident. fuera del FROM puede ser usada como un objeto. Ej:

```
SELECT p.number FROM Employee e JOIN e.phones p
```

Inner Join - Single-Valued Association Fields

- Con lo visto hasta aquí una operación de JOIN entre Employee y Deparment puede ser:
  - *SELECT d FROM Employee e JOIN e.department d*
- Del mismo modo, se puede realizar un join implicito empleando “path expressions”
  - *SELECT e.department FROM Employee e*
- Siempre que se utiliza una “path expression” se produce un JOIN Implícito (las consultas anteriores son equivalentes)

Inner Join - Single-Valued Association Fields

- SELECT DISTINCT e.department FROM Project p JOIN p.employees e WHERE p.name = 'Release1' AND e.address.state = 'CA‘
- **es equivalente a**
- SELECT DISTINCT d FROM Project p JOIN p.employees e JOIN e.department d JOIN e.address a WHERE p.name = 'Release1' AND a.state = 'CA‘
- SQL Ejecutado:
  - SELECT DISTINCT d.id, d.name
  - FROM project p, emp_projects ep, emp e, dept d, address a
  - WHERE p.id = ep.project_id AND ep.emp_id = e.id AND e.dept_id = d.id AND e.address_id = a.id AND p.name = 'Release1' AND a.state = 'CA'

Inner Joins – Otras expresiones

- Join en WHERE
  - *SELECT DISTINCT d FROM Department d, Employee e WHERE d = e.department*
  - Este estilo de consulta permite para compensar la falta de una expresión que muestre de manera explicita la relación entre dos entidades del dominio del modelo.
- Multiple Joins: mas de un JOIN se puede invocar sobre una consulta (se interpretan de der. a izq.):
  - *SELECT DISTINCT p FROM Department d JOIN d.employees e JOIN e.projects p*

OUTER JOIN

- Produce un dominio, donde solo una de las partes de la relación es requerido que esté completo, en tanto la otra parte de la relación debe recuperar de su dominio, solo aquellas instancias que cumplen con la relación de **JOIN.**
  - *SELECT e, d FROM Employee e LEFT JOIN e.department d*
- Retorna **TODOS** los empleados, y aquellos que están asignados a un departamento, serán retornados con dicha relación cargada.
- Pero los departamentos que no tengan empleados, no serán devueltos en esta consulta.

Fetch Joins

- Permite optimizar el acceso a la base de datos y preparar resultados de consultas para ser pasados al estado “detached”.
- Se especifican en las consultas, las relaciones que deben ser navegadas y precargadas por el motor de consultas, para que no sean cargadas de un modo lazy mas tarde en tiempo de ejecución.
- Una operación de fetch join se distingue de una operación de join común, agregando la palabra clave FETCH a el operador JOIN.
  - SELECT e FROM Employee e JOIN FETCH e.address
  - No hay una variable de identificación para la path expression, “e.address”, porque no es parte del tipo de resultado de la consulta.
  - El resultado es una colección de instancias de empleados con la dirección (pre)cargada.

Cláusula WHERE

- Permite especificar condiciones de filtro para reducir la cantidad de instancias devueltas.
- Se debe usar la palabra clave WHERE seguida de una expresión condicional.
- Se pueden filtrar los resultados a través de parámetros de entrada de 2 tipos:
  - Parámetros nombrados: `SELECT e FROM Empleados e WHERE e.salario > :s1`
  - Parámetros ordinales: `SELECT e FROM Empleados e WHERE e.salario > ?1`

Expresión Between

- Permite determinar si un valor se encuentra dentro de un rango o no.
- Se pueden evaluar expresiones :
  - numéricas,
  - String
  - fechas.
- Ejemplo:

```
SELECT e FROM Employee e
WHERE e.salary BETWEEN 400 AND 500
```

Expresión LIKE

- Permite buscar patrones dentro de un String.
- Los caracteres comodín son:
  - (_) para un carácter simple
  - (%) para múltiples caracteres.

```
SELECT d FROM Department d
WHERE d.name LIKE '__PR%'
```

- Si el patrón contiene algun carácter comodín, entonces se debe usar la cláusula ESCAPE.

```
SELECT d FROM Department d
WHERE d.name LIKE 'QA\_%' ESCAPE '\'
```

- Así “QA_Norte” coincide pero no “QANorte”.

Subconsultas

- Pueden ser usadas en las cláusulas WHERE y HAVING de una consulta.
- Una subconsulta es en realidad un consulta completa embebida dentro de una expresión condicional.
- El resultado de ejecutar la subconsulta, el cual puede ser o bien un escalar, o bien una colección, es evaluado en el contexto de la expresión condicional.
  - Permiten resolver escenarios de consultas muy complejas.

Subconsultas

- Por ejemplo, la siguiente consulta

```
SELECT e FROM Empleados e
WHERE e.salario = (SELECT MAX(e.salario)
                   FROM Empleados e)
```

- Esta consulta retorna el empleado con el máximo salario entre todos los empleados que hay en la tabla empleados.
- En este caso se utiliza una subconsulta para retornar el valor del máximo salario, a través de una función de agregación y compararlo con el salario de todos y cada uno de los empleados, para asi encontrar cuales son los que tienen dicho ingreso.

Subconsultas

- Una subconsulta puede ser empleada en la mayoría de las expresiones condicionales y puede aparecer tanto en el lado derecho como en el izquierdo de la expresión.
  - El ámbito donde un identificador de variable es válido, comienza en la consulta externa, y puede ser usado en cada una de las subconsultas contenidas.
  - En el caso de que una subconsulta declare una variable de identificación con el mismo nombre que una variable definida en una consulta mas externa, la misma es sobrescrita por lo que dentro de la subconsulta no se puede referir a la variable padre. Esto sucede por ejemplo en el ejemplo anterior con la variable de identificación **“e”** para **Empleados**.
- No garantiza portabilidad.

Expresiones IN

- Este tipo de expresiones permiten chequear si un valor simple es miembro de una colección.
- La colección puede ser definida como un conjunto de valores literales, o como el resultado de una colección.

```
SELECT e FROM Empleado e
WHERE e.cargo.nombre IN (‘Programador', ‘Arquitecto‘,’Analista’)
```

- La expresión puede ser negada también con la expresión NOT IN (...).

Expresiones de colección

- El operador IS EMPTY permite verificar si una colección es vacía o no.

```
SELECT e FROM Empleados e
WHERE e.proyectos IS EMPTY
```

- La expresión IS EMPTY por lo general es traducida a SQL como una subconsulta.
- La operación MEMBER OF verifican si un elemento es miembro de una colección.

```
SELECT e FROM Empleados e
WHERE :p MEMBER OF e.proyectos
```

Expresiones EXISTS – ANY –ALL - SOME

- Los operadores EXIST , ANY, ALL y SOME pueden ser usados para comparar una expresión con el resultado de una subconsulta.
  - La condición ALL retorna true si todos los resultados de la subconsulta cumplen la condición especificada.
  - La condición SOME o ANY retorna true si al menos uno de los resultados de la subconsulta cumplen la condición especificada.
  - La condición EXIST retorna true, si una subconsulta retorna alguna fila.

Expresiones Condicionales

- Las expresiones condicionales pueden emplear funciones que pueden ser usadas en las cláusulas WHERE o HAVING para enriquecer las comparaciones realizadas en las expresiones de estas consultas.

- ABS(number)
- MOD(number1, number2)
- SQRT(number)
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- LENGTH(string)
- LOWER(string)
- UPPER(string)
- SIZE(collection)

- SUBSTRING(string, start, end)
- TRIM([[LEADING|TRAILING|BOTH] [char] FROM] string)
- CONCAT(string1, string2)
- LOCATE(string1, string2 [, start])

Expresiones

- Merece especial atención la función SIZE, dado que implica siempre una subconsulta con una función de agregación. Por ejemplo, la siguiente consulta EJB-QL :

```
SELECT d
FROM Departmento d
WHERE SIZE(d.empleados) = 2
```

- Se traduce de la siguiente manera a ANSI-SQL

```
SELECT d FROM Departmento d
WHERE (SELECT COUNT(e)
FROM d.empleados e) = 2
```

FUNCTION / Nuevo en JPA 2.1 !

- En JPA 2.1 se han agregado algunas nuevas palabras reservadas como:
  - FUNCTION,
  - ON,
  - TREAT
- FUNCTION permite ejecutar funciones almacenadas como procedimientos en la base de datos. La sintaxis es FUNCTION(function_name {, function_arg}*)

```
1. SELECT c
2. FROM Customer c
3. WHERE FUNCTION('hasGoodCredit', c.balance, c.creditLimit)

4. SELECT FUNCTION('YEAR', e.startDate) AS year, COUNT(e)
   FROM Employee e GROUP BY year
```

Cláusula ORDER BY

- Las consultas pueden ser opcionalmente ordenadas usando una o mas expresiones comprendidas por variables de una expresión de camino
- Se puede usar las palabras claves ASC o DESC
- Ejemplo

```
SELECT e
FROM Empleado e
ORDER BY e.nombre DESC
```

Funciones de Agregación

- Son usadas para obtener información sumarizada de grupos de resultados de una consulta.
- Una consulta se considera de agregación, si usa una función de agregado o contiene la cláusula GROUP BY y/o HAVING
- Sintáxis

```
SELECT <select_expression>
FROM <from_clause>
[WHERE <conditional_expression>]
[GROUP BY <group_by_clause>]
[HAVING <conditional_expression>]
[ORDER BY <order_by_clause>]
```

Funciones de Agregación

- Ejemplo 1:

```
SELECT AVG(e.salario) FROM Empleado e
```

- Ejemplo 2:

```
SELECT d.nombre, AVG(e.salario)
FROM Departamento d JOIN d.empleados
GROUP BY d.nombre
```

- Ejemplo 3:

```
SELECT d.nombre, AVG(e.salario)
FROM Departamento d JOIN d.empleados
WHERE e.cargo.nombre LIKE 'A%'
GROUP BY d.nombre
HAVING AVG(e.salario) > 200
```

Nuevo en JPA 2.0

- Se permite un parámetro de tipo «Collecction» en las cláusulas «IN»
  - **SELECT** e **FROM** Employee e **WHERE** e.id **IN :param**

¿Como ejecutamos las consultas?

- Hay diferentes formas, las vistas hasta aquí requieren de obtener un objeto Query
- NO obstante para ejecutar consultas sobre la base de datos tenemos 6 enfoques diferentes.
  - Dynamic queries: es la forma más simple, escribimos la consulta en el momento de usarla y la ejecutamos..
  - Named queries: se definen las consultas en tiempo de despliegue y luego pueden ser usadas cuando se necesitna
  - Criteria API: es válido desde JPA 2.0 e introduce el concepto de un API OO para consultas
  - Native queries: SQL nativo en el proveedor de datos.
  - Stored procedure queries: Nuevo en JPA 2.1 y permite realizar consultas sobre procedimientos almacenados.

Definición de consultas

- JPA provee la interface Query para configurar y ejecutar consultas.
- Las consultas pueden ser dinámicas, especificada en tiempo de ejecución, o puede ser configurada como metadatos en alguna unidad de persistencia, y se denominan consultas nombradas.

La interface Query

```
1.  package javax.persistence;
2.  public interface Query {
3.      public List getResultList( );
4.      public Object getSingleResult( );
5.      public int executeUpdate( );
6.      public Query setMaxResults(int maxResult);
7.      public Query setFirstResult(int startPosition);
8.      public Query setHint(String hintName, Object value);
9.      public Query setParameter(String name, Object value);
10.     public Query setParameter(String name,Date d,TemporalType t);
11.     public Query setParameter(String name,Calendar c,TemporalType t);
12.     public Query setParameter(int position, Object value);
13.     public Query setParameter(int position, Date d, TemporalType t);
14.     public Query setParameter(int position, Calendar c, TemporalType t);
15.     public Query setFlushMode(FlushModeType flushMode);
```

Definición de consultas dinámicas

- Una consulta puede ser definida dinámicamente pasando el string de la consulta al método createQuery() de la interface EntityManager.
- Cada vez que una consulta dinámica solicita ser creada, se debe compilar y traducir a SQL, lo que implica que consuma algunos recursos para este procesamiento.

Definición de consultas dinámicas

```
1. @Stateless
2. public class ReportesBean implements Reportes {
3. @PersistenceContext(unitName="TP4")
4. EntityManager em;
5. public long queryEmpSalary(String dN, String eN) {
6.     String q = "SELECT e.salario FROM Empleados e WHERE
   e.departamento.nombre ='"+dN+"' AND e.nombre = '" + eN +
   "'";
7.     return (Long) em.createQuery(q).getSingleResult();
8. }
9. }
```

Definición de consultas dinámicas

- En los ejemplos que hemos analizado, las consultas retornan solo un único valor.
- La consulta es ejecutada en el momento en que el método getSingleResult() es invocado, no antes.
- Este método, espera que la llamada retorne uno y solo un valor como resultado.
- En caso de que no se encuentren resultados, se lanza una excepción del tipo javax.persistence.EntityNotFoundException.
- En cambio, si los resultados obtenidos son mayores a un unico registro, entonces una excepción del tipo javax.persistence.NonUniqueResultException.

Definición de consultas dinámicas

- En caso de que usemos una consulta un poco mas general, que nos pueda devolver múltiples registros de una tabla, es necesario invocar el método getResultList().
- Este método, en caso de no encontrar ningún resultado no lanza ninguna excepción, sino que retorna una lista vacía.

Definición de consultas dinámicas con parámetros

```
1. @Transactional
2. public class ReportesBean implements Reportes {
3. @PersistenceContext(unitName="TP4")
4. EntityManager em;
5. public List queryEmpList(String dN) {
6.     String qs = "SELECT e FROM Empleados e ";
7.     Query q = em.createQuery(qs);
8.     return q. getResultList();
9. }
10.}
```

Parámetros

- Los parámetros se pueden reemplazar por nombre o por posición a través del método “setParameter()”

```
1.public List findByName(Strig nom,String ape){
2. Query query = entityManager.createQuery(
3. “Select e from Empleados e where e.nombre =?1 “+
4. “AND w.apellido = ?2”);
5. query.setParameter(1,nom);
6. query.setParameter(2,ape);
7. return query.getResultList( );
8.}
```

Parámetros

- Reemplazo de parámetros nombrados.

```
1.public List findByName(Strig nom,String ape){
2. Query query = entityManager.createQuery(
3. "Select e from Empleados e where e.nombre =:p1 "+
4. "AND w.apellido = :p2");
5. query.setParameter("p1",nom);
6. query.setParameter("p2",ape);
7. return query.getResultList( );
8.}
```

Parámetros de tipo fecha.

- En caso de que se necesite pasar a una consulta, parámetros de las clases java.util.Date o java.util.Calendar, no se puede usar el método setParameter visto anteriormente, sino que debe emplearse un método especial
  - **“setParameter”, es sobrecargado agregándole un tercer parámetro.**
- La necesidad de un nuevo parámetro, se debe a que un objeto del tipo Date o del tipo Calendar, puede representar una fecha real, un horario del día, o una marca de tiempo.
- Este tercer parámetro, del tipo javax.persistence.TemporalType es el que cumple dicha función.

Consultas Nombradas

- Las consultas nombradas son un potente mecanismo para organizar la definición de consultas y mejorar la performance de la aplicación.
- Mediante anotaciones, una consulta nombrada se define a través de la anotación @NamedQuery la cual puede aparecer en la definición de un entity bean.
- Normalmente las consultas nombradas son ubicadas en el entity bean que mayor correspondencia tiene con los resultados mostrados por la consulta.

Consultas Nombradas

- Ejemplo
    1. @NamedQuery(name="findTareaNoConcluida",
    2. query="SELECT t.id FROM tarea t WHERE t.concluida :P_CONCLUIDA")
- En este ejemplo el lugar apropiado para definir esta consulta sería el bean Tarea.
- El nombre de una consulta tiene como alcanze toda la unidad de persistencia por lo que debe ser único en dicho ámbito. Esta restricción es importante, dado que nombres muy genéricos, como “findAll” no pueden ser usados ampliamente y deberían ser calificados para cada entidad.
- Una posibilidad para evitar esta situación, es prefijar las consultas, por lo que podriamos usar una consulta nombrada del tipo “Tareas.findAll”.
- En caso de que en un entity bean se necesite definir mas de una consulta para una misma clase, la anotación que se debe usar es @NamedQueries la cual acepta una array de una o mas anotaciones @NamedQuery.

Consultas nombradas múltiples

```
1.  @NamedQueries({
2.      @NamedQuery(name="Empleado.findAll",
3.      query="SELECT e FROM Empleado e"),
4.      @NamedQuery(name="Empleado.findByPK",
5.      query="SELECT e FROM Empleado e WHERE e.id = :id"),
6.      @NamedQuery(name="Empleado.findByName",
7.      query="SELECT e FROM Empleado e WHERE e.nombre = :name")
8.  })
9.  @Entity
10. Public class Empleado { . . . . . }
```

Uso de consultas nombradas

```
1.  @Stateless
2.  public class Reportes implements Reportes {
3.      @PersistenceContext(unitName="empleados")
4.  EntityManager em;
5.  public Employee findEmpleadoByName(String name) {
6.  return (Empleado)
    em.createNamedQuery("Empleado.findByName")
7.  .setParameter("name", name)
8.  .getSingleResult();
9.  }
10. // ...
11. }
```

Consultas de actualizacion masiva

- La ejecución de sentencias de actualización de datos y borrado de los mismos en forma masiva son muy comunes cuando se trabaja con bases de datos relacionales.
- La idea subyacente detrás de estas sentencias es la de eliminar o modificar de una sola vez y con una sola ejecución, un conjunto de entidades.
- **Sintáxis de Delete**
- `delete_statement ::= delete_clause [where_clause]`
- `delete_clause ::=DELETE FROM abstract_schema_name [[AS ] identification_variable]`

Consultas de actualizacion masiva

- **Sintáxis de Update**
- update_statement ::= update_clause [where_clause]
- update_clause ::=UPDATE abstract_schema_name [[AS] identification_variable] SET update_item {, update_item}*
- update_item ::= [identification_variable.]{state_field | single_valued_association_field} = new_value
- new_value ::= simple_arithmetic_expression | string_primary | datetime_primary | boolean_primary | enum_primary |
- simple_entity_expression | NULL

Reglas para actualizaciones masivas

1. La operación se aplica a la entidad especificada y a todas las subclases.
2. La operación no se ejecuta en cascada con ninguna de las entidades relacionadas.
3. EL nuevo valor especificado en un cláusula UPDATE debe coincidir con el tipo esperado en el atributo específico.
4. Las sentencias UPDATE se ejecutan directamente en la base de datos, esto significa que no se chequean bloqueos o cambios que se estén haciendo en entidades administradas.
5. El contexto de persistencia no es sincronizado con el resultado de la operación.

Reglas para actualizaciones masivas

- Debe quedar claro para el desarrollador, que el contexto de persistencia no es actualizado para reflejar los cambios de la operación de actualización.
- La mejor opción es ejecutar la operación en una transacción propia, dado que se elimina la posibilidad que se lean datos antes de que sean afectados por una operación de actualización masiva.
- Una estrategia típica que llevan a cabo los proveedores de persistencia para trabajar con estas operaciones de actualización masivas, es la de invalidar todo los datos “en memoria” relacionados con la entidad que está siendo afectada.
- Esto fuerza a que los datos sean recargados desde la base de datos la próxima vez que la entidad sea requerida.
- Un punto negativo de esta estrategia es que puede involucrar la invalidación de muchas entidades.

ANEXO

Otros aspectos de consultas

Actividad práctica

- Resolver las reglas de negocio que permita inscribir un alumno si:
  - No tomo ya el curso.
  - Si hay cupo en caso de que sea un curso general, o si tiene los créditos necesarios en caso de que sea un curso especifico.
- Listar:
  - Todos los cursos de un alumno.
  - Todos los cursos donde existe cupo disponible.
  - Todos los alumnos que tomaron 2 o más cursos generales y 2 o más cursos específicos.

Consultas con polimorfismo y Herencia

- JPA 1.0 soporta polimorfismo y herencia.
- Supongamos que tenemos una clase abstracta «Proyecto» que es extendida por dos clases concreta «ProyectoDeCalidad» y «ProyectoDeDiseño».
- La consulta:
  - SELECT p FROM Proyecto p WHERE p.empleados IS NOT EMPTY
- Retornará todos los proyectos. No tenemos forma de filtrar por Tipo en JPQL.

Consultas con polimorfismo y Herencia (nuevo en JPA 2.0)

- JPA 2.0 incluye el operador **TYPE** que permite filtrar en una consulta los objetos por su tipo.
- Ejemplo

  SELECT p FROM Proyecto *p*

  WHERE TYPE(p) = ProyectoDeCalidad *OR*

  TYPE(p) = ProyectoDeDiseño
- *Notar que no son necesrias las comillas, dado que* ProyectoDeDiseño y ProyectoDeCalidad son tipos (clases) reconocidos por el sistema.

Consultas sobre Maps: KEY, VALUE (nuevo en JPA 2.0)

- Supongamos como vimos anteriormente que tenemos una clase Empleado, que posee un Map, con los telefonos *(«phones»)* que posee, donde la clave del Map es el tipo del telefono y el valor es el número.
- Con la siguiente consulta, obtendremos los telefonos laborales y celular de los empleados:
  - SELECT e.name, **KEY**(p), **VALUE**(p)
  - FROM Employee e JOIN e.phones p
  - WHERE **KEY**(p) IN ('Work', 'Cell')
- Como se puede apreciar, KEY y VALUE, permiten acceder a la clave o el valor deldel Map respectivamente, y usarlos en una comparación o como un tipo de retorno.

Valores Escalares en SELECT (nuevo en JPA 2.0)

- En JPA 2.0, en un select se pueden retornar valores escalares (es decir valores fijos).
- Select 1 from Producto p

Expresiones CASE (nuevo en JPA 2.0)

- JPA 2.0 Soporta la estructura CASE en consultas.
  - Se utilizan en la clausula Select.
  - Permite transformar los datos de las entidades en información útil para un reporte.
- **Ejemplo**

```
1. SELECT p.name,
2. CASE WHEN TYPE(p) = DesignProject THEN 'Development'
3. WHEN TYPE(p) = QualityProject THEN 'QA'
4. ELSE 'Non-Development'
5. END
6. FROM Project p
7. WHERE p.employees IS NOT EMPTY
```

Expresiones CASE (nuevo en JPA 2.0 JavaEE6)

- Existe una variación, que es comparar el valor antes (como se aprecia en la línea 2):

```
1. SELECT p.name,
2. CASE TYPE(p)
3. WHEN DesignProject THEN 'Development'
4. WHEN QualityProject THEN 'QA'
5. ELSE 'Non-Development'
6. END
7. FROM Project p
8. WHERE p.employees IS NOT EMPTY
```

Expresiones CASE (nuevo en JPA 2.0 JavaEE6)

- Una tercer variante de expresión CASE en JPQL, es la función COALESCE.
  - La sintaxis de esta función es la siguiente:

    COALESCE(<scalar_expr> {,<scalar_expr>}+)
  - Esta función recibe una lista de expresiones (la lista puede tener N expresiones), y muestra como resultado la primera no nula.
- Ejemplo:
  - SELECT COALESCE(d.name, d.city,d.id) FROM Department d

    Si el nombre del departamento es nulo retorna la ciudad y si este es nulo también retorna el «id»

Otras mejoras en JPQL (JPA 2.0 JavaEE6)

- Se permite el AS en cláusulas SELECT
  - **SELECT** AVG(e.salary) **AS** sueldoMedio, e.address.city
  - **FROM** Employee e
  - **GROUP BY** e.address.city
  - **ORDER BY** s
- Se permite la función “CONCAT” con múltiples argumentos
  - **SELECT** e
  - **FROM** Employee e
  - **WHERE** CONCAT(e.address.street, e.address.city, e.address.province) = :address

Paginado de resultados

- Cuando, los resultados retornados por una consulta son muchos mas registros de los que puede visualiza un usuario en su pantalla, una opción muy común es la de paginar los resultados cuando estos exceden cierta cantidad de registros.
- Anteriormente, la paginación debía ser programada íntegramente por el usuario, y por lo general, no siempre de la manera mas eficiente.
- Para mejorar esta situación, JPQL provee dos funciones incorporadas en su API, que se pueden usar en situaciones como la planteada:
  - setMaxResults( )
  - setFirstResult( )

Paginado de resultados

- Ejemplo: Obtener todos los empleados

```
1. public List getPersonas(int first,int max){
2. String qs ="SELECT e FROM Empleados e";
3.     Query q = em.createQuery(qs);
4.     query.setMaxResults(max);
5.     query.setFirstResult(index);
6.     return query.getResultList( );
7. }
```

Paginado de resultados

- Aquí el método getEmpleados ejecuta una consulta que obtiene todas las personas de la base de datos.
  - Para poder mostrar los resultados de una manera ordenada, limitamos el número de resultados que se obtienen a través del parámetro max, el cual es pasado como argumento al método setMaxResults(int)
  - A través del método setFirstResult(int), le indicamos desde que posición del conjunto de resultados debe empezar a contar hasta alcanzar el máximo de registros indicados.

Paginado de resultados ejemplo

```
1.  .. …… ..
2.  List results;
3.  int first = 0;
4.  int max = 10;
5.  do {
6.    results = getPersonas (first ,max);
7.    Iterator it = results.iterator( );
8.    while (it.hasNext( )) {
9.      Empleado e = (Empleado)it.next( );
10.     . . .
11.   }
12.   entityManager.clear( );
13.   first = first + results.getSize( );
14. }
15. while (results.size( ) > 0);
16. .. ……..
```

ANEXO

Interceptores y otros tipos de consultas.

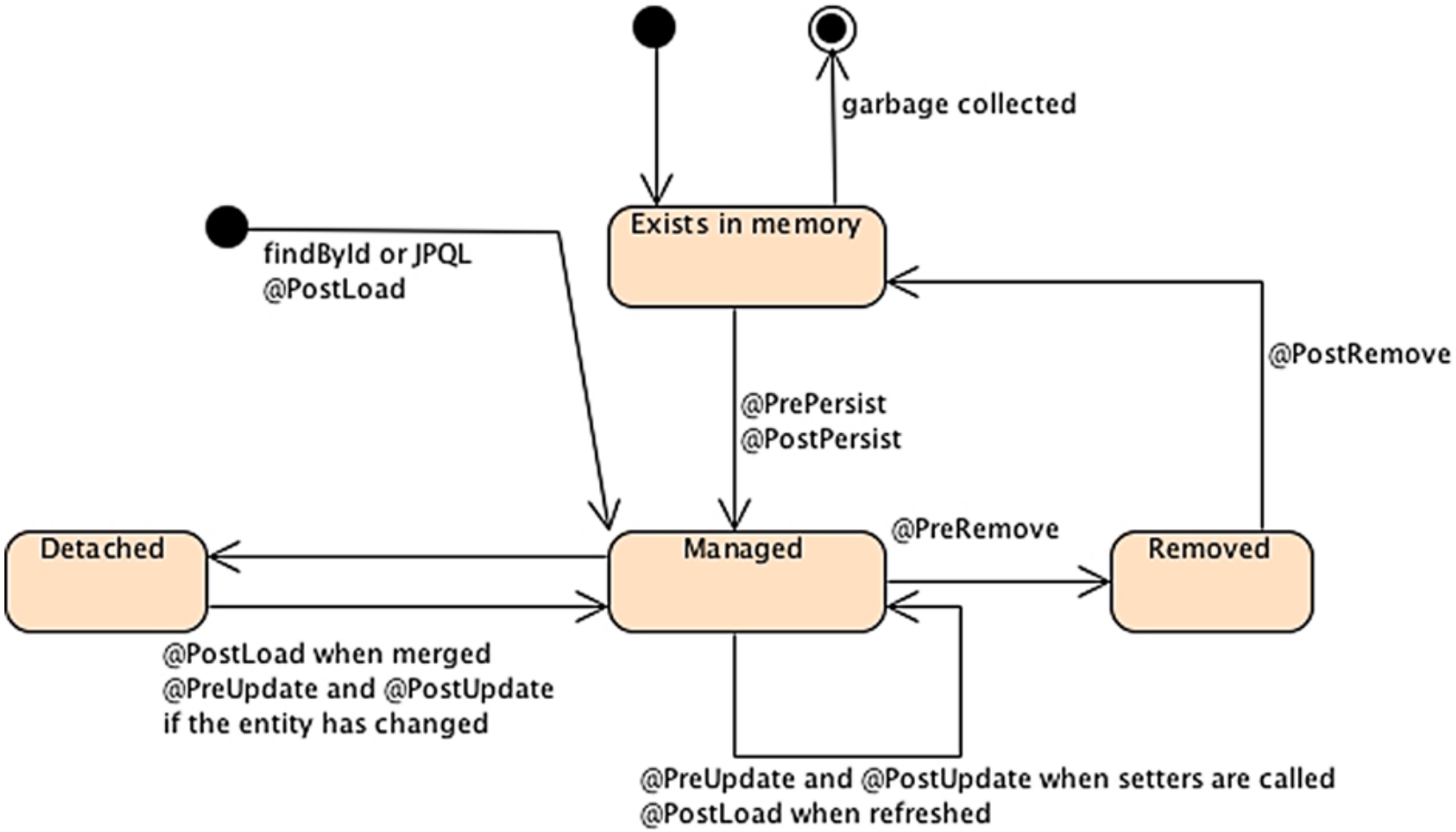
Interceptores

- El ciclo de vida de una entidad se puede dividir en 4 categorías:
  - persistir → insert
  - actualizar → update
  - remover → delete
  - cargar. → select
- Cada una de estas etapas pueden ser interceptadas por código nuestro, tanto antes como después de realizarse, para trabajar o adaptar la entidad según necesitemos.
- Para ello tenemos las anotaciones @PreXXX y @PostXXX y si anotamos un método con cada una de ellas se invocará:

Interceptores

- @PrePersist antes de EntityManager.persist().
- @PostPersist: luego de persist(). Si la entidad autogenera su clave primaria este valor estará disponible.
- @PreUpdate antes de que se ejeucte un update por un merge o un setter de la entidad.
- @PostUpdate luego del update en la base de datos.
- @PreRemove antes de que la entidad sea marcada para eliminar.
- @PostRemove luego que la entidad es removida de la db.
- @PostLoad luego de que se invoca el find o se refresca el contenido de la base de datos. No tiene sentido el

Interceptores



Interceptores

```
1.  @Entity
2.  public class Customer {
3.  @Id @GeneratedValue
4.  private Long id;
5.  private String firstName;
6.  private String lastName;
7.  private String email;
8.  private String phoneNumber;
9.  @Temporal(TemporalType.DATE)
10. private Date dateOfBirth;
11.  @Transient
12. private Integer age;
13. @Temporal(TemporalType.TIMESTAMP)
14. private Date creationDate;
```

Interceptores

```
1. @PrePersist
2. @PreUpdate
3. private void validate() {
4. if (firstName == null || "".equals(firstName))
5. throw new IllegalArgumentException("Invalid first name");
6. if (lastName == null || "".equals(lastName))
7. throw new IllegalArgumentException("Invalid last name");
8. }
```

Interceptores

```
1.  @PostLoad
2.  @PostPersist
3.  @PostUpdate
4.  public void calculateAge() {
5.  if (dateOfBirth == null) { age = null; return; }
6.  Calendar birth = new GregorianCalendar();
7.  birth.setTime(dateOfBirth);
8.  Calendar now = new GregorianCalendar();
9.  now.setTime(new Date());
10. int adjust = 0;
11. if (now.get(DAY_OF_YEAR) - birth.get(DAY_OF_YEAR) < 0) { adjust = -1; }
12. age = now.get(YEAR) - birth.get(YEAR) + adjust;
13. }
14. }
```

Interceptores

- En las operaciones @PrePersist y @PreUpdate, se validan los datos y si no cumplen las reglas previstas se arroja una excepción.
- En el PostUpdate se calcula un campo “transient” (no persistente).
- Un interceptor del ciclo de vida no puede ser estático, y puede ser anotado con múltiples anotaciones de ciclo de vida, pero no podemos tener 2 métodos con la misma anotación en la misma entidad.
- Pueden lanzar excepciones “unchecked” no “checked”, dado que las primeras realizan rollbacks de las transacciones JTA.
- En un método podemos invcar JNDI, JDBC, JMS, y EJBs pero no operaciones del EntityManager o de consultas

Criteria API (Object-Oriented Queries)

- Una de las desventajas de las consultas JPQL, es que están basadas en la declaración de un string de cierta longitud y complejidad, que no puede ser chequeado en tiempo de compilación.
- Una idea superadora podría ser que las consultas sean escritas con un enfoque de POO, donde obtenemos un resultado solicitando determinados comportamientos en objetos que colaboran para una responsabilidad específica.
- Armar una consulta dando instrucciones y no escribiéndola. El chequeo es en tiempo de compilación!
- En JPA 2.se introdujo un nuevo API, llamdo Criteria API que define en el paquete javax.persistence.criteria la posibilidad de escribir consultas con un estilo orientado a objetos.

Criteria API (Object-Oriented Queries)

- Ejemplo
- Consulta en JPQL

1. List<Customer> = (List<Customer>) em.createQuery(SELECT c FROM Customer c WHERE c.firstName = 'Vincent').getResultList();

- Consulta en Criteria API.

1. CriteriaBuilder builder = em.getCriteriaBuilder();
2. CriteriaQuery<Customer> criteriaQuery = builder.createQuery(Customer.class);
3. Root<Customer> c = criteriaQuery.from(Customer.class);
4. criteriaQuery.select(c).where(builder.equal(c.get("firstName"), "Vincent"));
5. Query query = em.createQuery(criteriaQuery).getResultList();
6. List<Customer> customers = query.getResultList();

Consultas masivas en Criteria API (Nuevo en JPA 2.1)

- Una de las mejoras fundamentales incluidas en JPA 2.1 es la posibilidad de ejecutar consultas “masivas” a través del api de criteria.

```
1. CriteriaUpdate<Customer> q =
   cb.createCriteriaUpdate(Customer.class);
2. Root<Customer> c = q.from(Customer.class);
3. q.set(c.get(Customer_.status), "outstanding")
4.  .where(cb.lt(c.get(Customer_.balance), 10000));
5. Query query = em.createQuery(q);
6. query.executeUpdate();
```

son equivalentes

```
1. UPDATE Customer c
2. SET c.status = 'outstanding'
3. WHERE c.balance < 10000
```

Consultas masivas en Criteria API (Nuevo en JPA 2.1)

- Consultas equivalentes para borrado.

```
1. CriteriaDelete<Customer> q =
   cb.createCriteriaDelete(Customer.class);
2. Root<Customer> c = q.from(Customer.class);
3. q.where(cb.equal(c.get(Customer_.status), "inactive"),
4.         cb.isEmpty(c.get(Customer_.orders)));
5. Query query = em.createQuery(q);
6. query.executeUpdate();
```

```
1. DELETE FROM Customer c
2. WHERE c.status = 'inactive'
3. AND c.orders IS EMPTY
```

Consultas Nativas

- Si queremos usar características nativas de una base de datos, entonces podemos directamente escribir la consulta en formato SQL específico de nuestro motor RDBMS.
- Toman una consulta SELECT, UPDATE, o DELETE y retornan una instancia de Query para ejecutar.
- No son portables, pero la principal ventaja es que cada objeto que retorna lo convierten a entidad.
- Ejemplo:
  - Query query = em.createNativeQuery("SELECT * FROM t_customer", Customer.class);
  - List<Customer> customers = query.getResultList();
- Se pueden definir mediante anotaciones
  - @Entity
  - @NamedNativeQuery(name = "findAll", query="select * from t_customer")
  - @Table(name = "t_customer")
  - public class Customer {...}

Constructor result (Nuevo JPA 2.1)

- Permite definir para una consulta nativa que no retorna una instancia de una entidad ,un objeto personalizado, como debe asociarse cada columna retornada con los parámetros del constructor.
- La clase destino debe tener un constructor que coincida con el especificado.

```
1.  @NamedNativeQuery(   name="findAllEmployeeDetails",
2.    query="SELECT E.EMP_ID, E.F_NAME, E.L_NAME, A.CITY from EMP E, ADDRESS A WHERE E.EMP_ID
    = A.EMP_ID",  resultSetMapping="employee-details")
3.  @SqlResultSetMapping(name="employee-details",
4.    classes={
5.      @ConstructorResult(targetClass=EmployeeDetails.class, columns={
6.          @ColumnResult(name="EMP_ID", type=Integer.class),
7.          @ColumnResult(name="F_NAME", type=String.class),
8.          @ColumnResult(name="L_NAME", type=String.class),
9.          @ColumnResult(name="CITY", type=String.class)
10.     })
11.   }
12. )
13. public class Employee {
14.   ...
15. }
```

```
1.  public class EmployeeDetails {
2.  public EmployeeDetails(Integer id,String fn,String ln,String
    cit){...}
3.  }
```

Stored Procedure Queries (nuevas en JPA 2.1)

- Un procedimiento almacenado es una subrutina disponible para todas las aplicaciones que acceden a una base de datos relacional.
- El uso típico es cuando un proceso requiere la ejecución de muchos y complejos SQL, o son tareas repetitivas sobre grandes volúmenes de datos, lo cual hace aconsejable realizar el tratamiento en la base de datos, y luego llevar a la aplicación, la información ya pre-procesada.
- Por lo general no son portables pero tienen algunas ventajas:
  - Muchas veces son más performantes.
  - Reducen la cantidad de datos que pasan a través de la red.
  - Centralizamos parte del código y se puede compilar de manera independiente a la aplicación.
  - Ofrece características anexas de seguridad.

```
1. CREATE PROCEDURE sp_archive_books @archiveDate DATE, @warehouseCode VARCHAR AS
2. UPDATE T_Inventory
3. SET Number_Of_Books_Left - 1
4. WHERE Archive_Date < @archiveDate AND Warehouse_Code = @warehouseCode;
5. UPDATE T_Transport
6. SET Warehouse_To_Take_Books_From = @warehouseCode;
7. END
```

```
1. @Entity
2. @NamedStoredProcedureQuery(name = "archiveOldBooks", procedureName =
   "sp_archive_books",
3. parameters = {
4. @StoredProcedureParameter(name = "archiveDate", mode = IN, type = Date.class),
5. @StoredProcedureParameter(name = "warehouse", mode = IN, type = String.class)
6. }
7. )
8. @Id @GeneratedValue
9. private Long id;
```

Invocar Stored Procedures

- Invocar el SP
  1. StoredProcedureQuery q = em.createNamedStoredProcedureQuery("archiveOldBooks");
  2. q.setParameter("archiveDate", new Date());
  3. q.setParameter("maxBookArchived", 1000);
  4. q.execute();
- Invocación dinámica
  1. StoredProcedureQuery query = em.createStoredProcedureQuery("sp_archive_old_books");
  2. query.registerStoredProcedureParameter("archiveDate", Date.class, ParameterMode.IN);
  3. query.registerStoredProcedureParameter("maxBookArchived", Integer.class, ParameterMode.IN);
  4. query.setParameter("archiveDate", new Date());
  5. query.setParameter("maxBookArchived", 1000);
  6. query.execute();

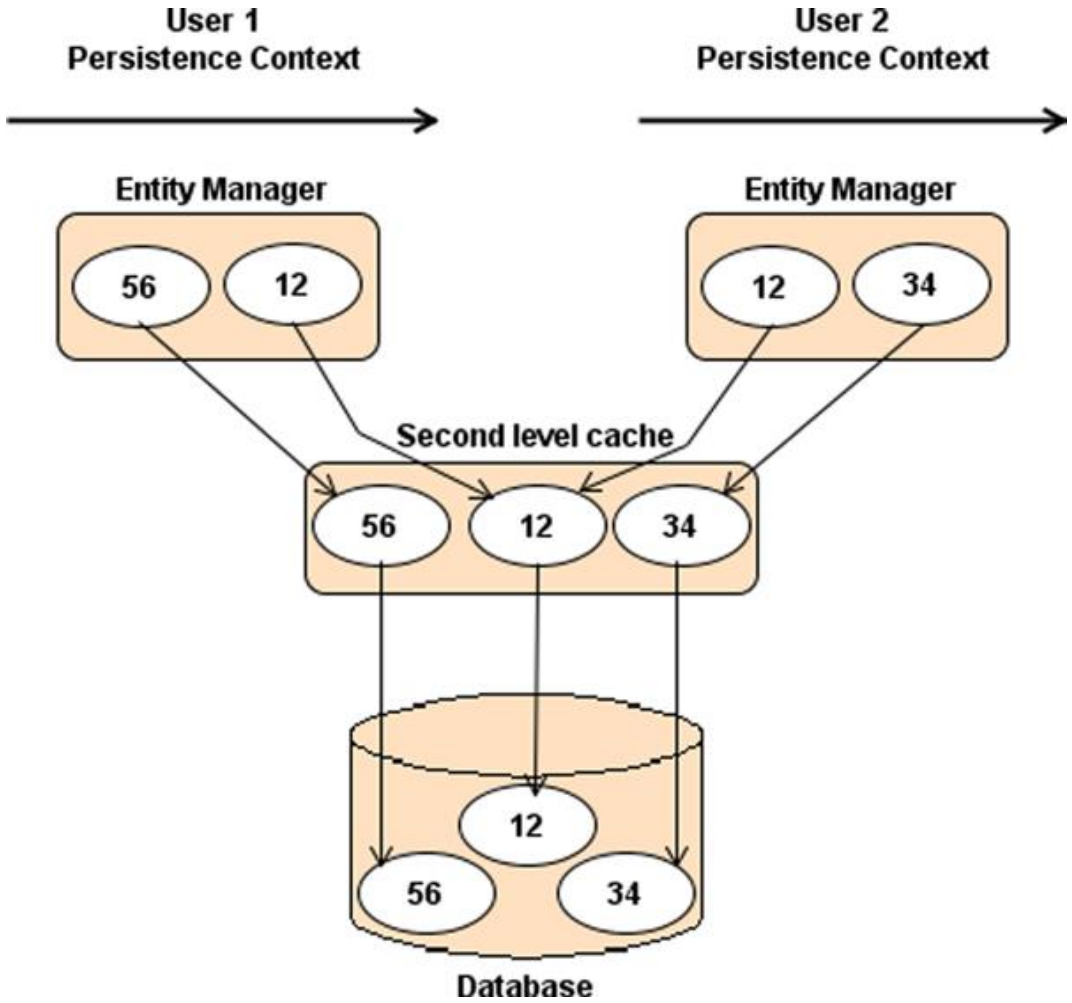
Cache API

- Es interesante como la mayoría de las especificaciones hacen foco en requerimientos funcionales y dejan de lado algunos no funcionales, como escalabilidad, seguridad o performance, y lo delegan como detalles de implementación.
- Así surgen diversas alternativas que van brindando mayores ventajas en su uso pero, se vuelven parte clave de cualquier aplicación, **y no están estandarizados**, por lo que cualquier cambio se vuelve un “dolor de cabeza”. El caso típico de esto en JPA fue el caché, que siempre fue implementado por los proveedores, con varias alternativas pero no definido en el estándar.

Cache API

- El entitiy manager siempre fue un caché de primer nivel, donde tenemos las entidades que necesitamos y lo usamos para ahorrar consultas a la base de datos. Pero es de corta duración, por transacción. → si un atributo x de un objeto es modificado 10 veces en la misma transacción, no se van a generar 10 sentencias UPDATE sino una sola con el estado final.
- Para optimizar el acceso a las bases de datos, se suele utilizar una caché de segundo nivel, donde en lugar de buscar una entidad directamente en la base de datos se la busque primero en esta caché.

Cache API


User 1
Persistence Context
User 2
Persistence Context
Entity Manager
Entity Manager
56
12
12
34
Second level cache
56
12
34
12
56
34
Database

Cache API

- Cada implementación tiene su propia forma de realizar el caché, lo que busca estandarizar JPA es un mecanismo para consultar y remover entidades de la caché de segundo nivel.
- Al igual que el entitiy Manager, tenemos una interface javax.persistence.Cache que nos brinda estas funcionalidades

```
1. public interface Cache {
2. // consulta si el objeto está en caché o no
3. public boolean contains(Class cls, Object id);
4. // quita el objeto de caché
5. public void evict(Class cls, Object id);
6. // quita todas las instancias de una entidad de caché
7. public void evict(Class cls);
8. // limpia todo el caché
9. public void evictAll();
10. // retorna la implementación de bajo nivel.
11. public <T> T unwrap(Class<T> cls);
12. }
```

Cache API

- Además podemos definir de manera explícita, que una entidad es “Cacheable” o no, usando la anotación @Cacheable.-
- Por defecto las entidades no son cacheables, sino hasta que agreguemos esta anotación

```
1. @Entity
2. @Cacheable(true)
3. public class Customer {
4. @Id @GeneratedValue
5. private Long id;
6. . . . .
7. }
```

Cache API

- Debemos indicar el mecanismo de caché que usaremos, en el archivo “persistence.xml” en el atributo “shared-cache-mode”.
  - ALL: todas las entidades son cacheadas
  - DISABLE_SELECTIVE: caché está habilitado para todos excepto las que tienen @Cacheable(false).
  - ENABLE_SELECTIVE: caché solo habilitado para las que tienen la anotación @Cacheable(true).
  - NONE: ninguna entidad será almacenada en caché de segundo nivel
  - UNSPECIFIED: el comportamiento en caché será el que defina por defecto el proveedor de persistencia.

Cache API

```
1.  Customer customer = new Customer("Patricia", "Jane", "plecomte@mail.com");
2.  tx.begin();
3.  em.persist(customer);
4.  tx.commit();
5.  // Uses the EntityManagerFactory to get the Cache
6.  Cache cache = emf.getCache();
7.  // Customer should be in the cache
8.  assertTrue(cache.contains(Customer.class, customer.getId()));
9.  // Removes the Customer entity from the cache
10. cache.evict(Customer.class);
11. // Customer should not be in the cache anymore
12. assertFalse(cache.contains(Customer.class, customer.getId()));
```