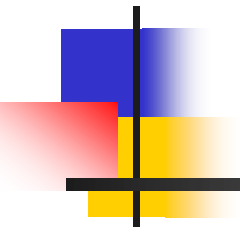
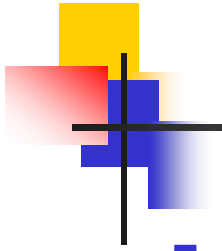
3.3 Implementación de la Persistencia con Hibernate



Índice



- Introducción
- Mapping objeto/relacional
  - Anotación de entidades
  - Modelado de relaciones
- Gestión de la Persistencia
  - API de Persistencia de Hibernate
  - Transacción típica
  - Ejemplos
  - Configuración e Inicialización de Hibernate
- Lenguaje de Consultas
- Implementación de DAOs con Hibernate

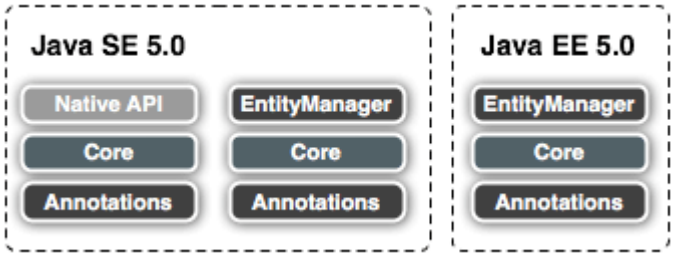
Introducción (1)

HIBERNATE

- Hibernate es un mapeador objeto/relacional (**ORM**), de código abierto, creado por un grupo de desarrolladores dirigido por Gavin King a finales de 2001
  - Es el ORM más utilizado actualmente – http://www.hibernate.org
- **Motivación**
  - Simplificar la programación de la capa de persistencia de las aplicaciones: utilizar JDBC directamente es muy tedioso
- Permite al desarrollador utilizar objetos persistentes en lugar de manipular directamente los datos de la BD
  - Asocia clases persistentes con tablas de una BD relacional, haciendo transparente la API de JDBC y el lenguaje SQL
  - Soporta relaciones entre objetos y herencia
  - No oculta el tipo de BD (relacional), pero sí la BD concreta (Oracle, MySQL, PostgreSQL, etc.)

Introducción (y 2)

- API dual
  - Implementa la API de Persistencia de Java (JPA)
  - Proporciona una API propia, que será la **usada en los ejemplos del curso**, porque en aplicaciones reales puede no llegar con la API de JPA



- Lenguaje de Consultas:
  - HQL (Hibernate Query Language)
    - Lenguaje de consultas de búsqueda y borrados/actualizaciones en masa, con semántica orientada a objetos
    - Sintaxis parecida a SQL
    - Un subconjunto está estandarizado como JPA QL
  - API de Programación para QBC (Query By Criteria)
  - Si es necesario, permite lanzar consultas SQL

Mapping objeto/relacional (1)

- Para que Hibernate sepa cómo mapear las instancias de clases persistentes (llamadas "entidades") a la BD es necesario especificar metainformación como nombre de la tabla, nombres de las columnas a las que mapear los atributos, etc.
- Alternativas para especificar la metainformación de mapping
  - Usar ficheros XML
    - El código quedaría "limpio" de referencias a elementos de la BD
  - Usar Anotaciones sobre la clase persistente
    - Hibernate mantiene los nombres de anotaciones JPA para las comunes y define otras adicionales
    - El código tiene cableado nombres de tablas, columnas, etc.
    - Sin embargo, es una solución más sencilla, y en consecuencia, representa el **enfoque aconsejado**
  - Es posible especificar la información de mapping en anotaciones y ficheros de configuración simultáneamente para una misma entidad
    - La información de los ficheros de configuración tiene preferencia sobre las anotaciones => se puede sobre-escribir la información dada con las anotaciones y/o añadir más información sin necesidad de tocar código

Mapping objeto/relacional (y 2)

- Para realizar la persistencia, es posible:
  - Utilizar tablas existentes
    - Única opción cuando las tablas ya estaban creadas
    - Más control sobre las tablas e índices que se usarán
    - **Por motivos pedagógicos, en los ejemplos del curso se ha seguido este enfoque**
  - Generar las tablas a partir de la información disponible en las clases persistentes
    - En la mayor parte de los casos será preciso especificar metainformación adicional, como la longitud de las columnas de tipo cadena de caracteres, etc.

Anotación de Clases Persistentes (1)

- En MiniBank se han definido dos clases persistentes:
  - `es.udc.pojo.minibank.model.account.Account`
  - `es.udc.pojo.minibank.model.accountoperation.AccountOperation`
- Requisitos de una clase persistente
  - Se anota con **`@Entity`**
    - NOTA: las anotaciones comunes con la API de persistencia de Java están en **`javax.persistence`**
  - Debe tener un constructor sin argumentos público o protegido (puede tener otros constructores)
  - No debe ser **`final`**
  - Debe tener una clave primaria
  - Puede extender de otra clase
  - Puede ser abstracta
  - Puede tener relaciones de asociación con otras

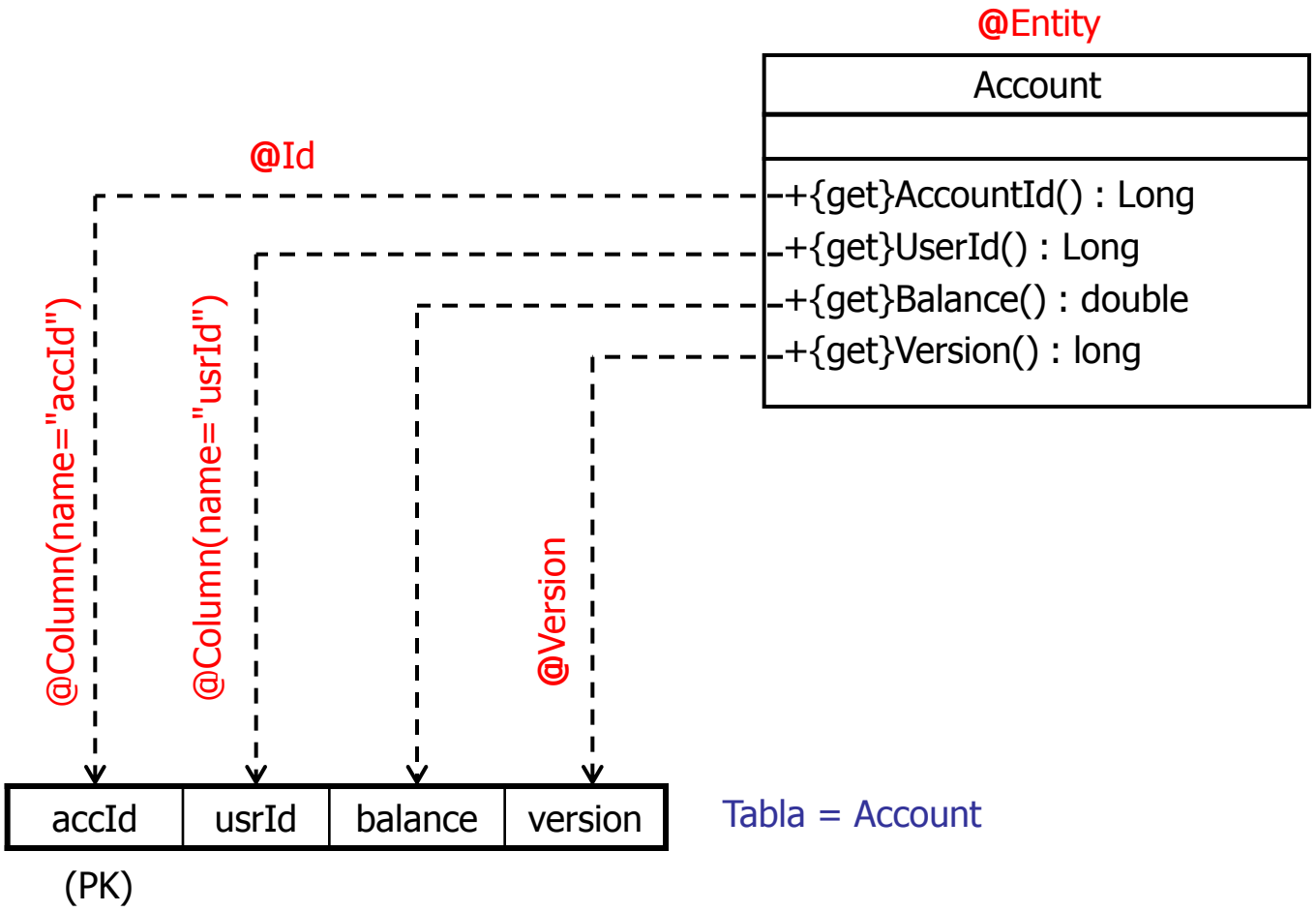
Anotación de Clases Persistentes (2)

- Requisitos de una clase persistente (cont)
  - Estado persistente
    - Si una clase se marca como persistente, por defecto se incluyen todas sus propiedades como persistentes
    - Se pueden excluir propiedades utilizando la anotación **`@Transient`** o definiendo la propiedad como **`transient`**
    - Típicamente disponen de métodos públicos **`get`** y **`set`** para acceder y modificar sus valores
    - La implementación de Hibernate accede al estado directamente a través de sus atributos (tipo de acceso: campo) o (exclusivo) mediante métodos **`get`**/**`set`** (tipo de acceso: propiedad)
      - La colocación de la anotación de clave primaria en los atributos o (exclusivo) métodos **`get`** determina el tipo de acceso
        - Con el "tipo de acceso propiedad", sólo se pueden colocar anotaciones en los métodos **`get`**

Anotación de Clases Persistentes (3)

- Requisitos de una clase persistente (cont)
  - Estado persistente (cont)
    - Los atributos o propiedades pueden ser (entre otros)
      - Tipos primitivos y sus contrapartidas objetuales: se mapean a una columna
      - **`java.lang.String`**, **`java.math.BigInteger`**, **`java.math.BigDecimal`**, **`java.util.Date`**, **`java.util.Calendar`**, **`java.sql.Date`**, **`java.sql.Time`**, **`java.sql.Timestamp`**, **`byte[]`**, **`Byte[]`**, **`char[]`** y **`Character[]`**: se mapean a una columna
      - Enumerados: se mapean a una columna
      - Otras entidades y colecciones de entidades: para mapear las relaciones con otras entidades se utilizan columnas y/o tablas adicionales dependiendo de la cardinalidad y direccionalidad de las relaciones
      - Otros
        - Tipos **`Serializable`** definidos por el usuario: normalmente se mapean a un BLOB
        - Permite anotar tipos definidos por el usuario para que sus atributos/propiedades se mapeen a columnas de la misma tabla que la entidad que la contiene

Anotación de Clases Persistentes (4)



Anotación de Clases Persistentes (5)

```
@Entity
public class Account {

    private Long accountId;
    private Long userId;
    private double balance;
    private long version;

    public Account() {}

    public Account(long userId, double balance) {
        /**
         * NOTE: "accountId" *must* be left as "null" since its value
         * is automatically generated.
         */
        this.userId = userId;
        this.balance = balance;

    }
```

Anotación de Clases Persistentes (6)

```
@Column(name="accId")
@SequenceGenerator(                 // It only takes effect for
     name="AccountIdGenerator", // databases providing identifier
     sequenceName="AccountSeq") // generators.
@Id
@GeneratedValue(strategy=GenerationType.AUTO,
                generator="AccountIdGenerator")
public Long getAccountId() {
    return accountId;
}

public void setAccountId(Long accountId) {
    this.accountId = accountId;
}

@Column(name="usrId")
public Long getUserId() {
    return userId;
}

public void setUserId(Long userId) {
    this.userId = userId;
}
```

Anotación de Clases Persistentes (7)

```
public double getBalance() {
    return balance;
}

public void setBalance(double balance) {
    this.balance = balance;
}

@Version
public long getVersion() {
    return version;
}

public void setVersion(long version) {
    this.version = version;
}
```

Anotación de Clases Persistentes (8)

- **`@Entity`**
  - Especifica que es una clase persistente
- **`@Column`**
  - Permite especificar información sobre cómo mapear un atributo/propiedad a una columna de la BD
    - En el ejemplo anterior, la propiedad **`accountId`** se mapea a la columna **`accId`** (por defecto se asume que el nombre de la columna coincide con el nombre del atributo/propiedad)
- **`@Id`**
  - Especifica el atributo/propiedad clave
  - Sólo vale para claves simples
  - Hibernate tiene soporte para especificar claves compuestas, si fuese necesario
    - Normalmente sólo es necesario cuando la BD ya estaba creada y tenía claves compuestas
    - En general lo recomendable es usar siempre claves simples

Anotación de Clases Persistentes (9)

- Generación automática de claves primarias
  - Casi todas las BBDD disponen de mecanismos para generar identificadores numéricos automáticamente, que se pueden usar como claves
  - Algunas disponen de contadores (secuencias)
    - Se pueden leer e incrementar atómicamente
    - e.g.: Oracle, PostgreSQL
    - En estas BBDD es posible: (1) generar el identificador (mediante una consulta especial) y (2) a continuación insertar la fila con el identificador generado
  - Otras BBDD disponen de columnas contador
    - Cada vez que se inserta una fila, se le asigna automáticamente un valor a esa columna-contador (el siguiente)
    - e.g.: MySQL, PostgreSQL, SQL Server, Sybase
    - En estas BBDD es necesario: (1) insertar la fila (sin especificar un valor para la columna-contador) y (2) a continuación leer el valor que se le ha asignado a la columna-contador lanzando una consulta especial (usando la API de JDBC 3.0, es el propio driver el que se encarga de recuperar el valor asignado)

Anotación de Clases Persistentes (10)

- Generación automática de claves primarias (cont)
  - Ejemplo para PostgreSQL – secuencias
    - Creación de la secuencia

```
CREATE SEQUENCE AccountSeq;
```

    - Creación de la tabla

```
CREATE TABLE Account (
  accId BIGINT NOT NULL,
  usrId BIGINT NOT NULL,
  balance DOUBLE PRECISION NOT NULL,
  CONSTRAINT AccountPK PRIMARY KEY(accId),
  CONSTRAINT validBalance CHECK ( balance >= 0 )
);
```

    - (1) Generar el identificador

```
SELECT nextval('AccountSeq');
```

      -> Devuelve el identificador **<id>**
    - (2) Insertar la fila

```
INSERT INTO Account(accId, usrId, balance)
  VALUES (<id>,1,2.0)
```

Anotación de Clases Persistentes (11)

- Generación automática de claves primarias (cont)
  - Ejemplo para MySQL – columnas-contador
    - Creación de la tabla

```
CREATE TABLE Account (
  accId BIGINT NOT NULL AUTO_INCREMENT,
  usrId BIGINT NOT NULL,
  balance DOUBLE PRECISION NOT NULL,
  CONSTRAINT AccountPK PRIMARY KEY(accId),
  CONSTRAINT validBalance CHECK ( balance >= 0 )
) ENGINE = InnoDB;
```

      - En MySQL, **`ENGINE = InnoDB`** es necesario para permitir y tratar adecuadamente transacciones y restricciones de integridad referencial
    - (1) Insertar la fila

```
INSERT INTO Account(usrId, balance)
  VALUES (1,2.0)
```

    - (2) Leer el valor que se la ha asignado a la columna-contador
      - Ejecutar la siguiente consulta para obtener el identificador

```
SELECT LAST_INSERT_ID();
```

      - O usar la API de JDBC 3.0 que proporciona un método para recuperar el valor del identificador

Anotación de Clases Persistentes (12)

- En **`Account`** se han usado anotaciones especiales en la propiedad de la clave primaria para que
  - Los valores se generen mediante una secuencia (si la BD proporciona secuencias)
  - Se mapee a una columna contador (si la BD ofrece columnas contador)
- **`@GeneratedValue`**
  - **`strategy`**
    - Especifica la estrategia de generación de identificadores numéricos
    - Es de tipo **`GenerationType`** (enumerado). Entre otros, es posible especificar los siguientes valores
      - **`SEQUENCE`**: usar una secuencia
      - **`IDENTITY`**: mapear la clave a una columna contador
      - **`AUTO`**: Hibernate decide la estrategia en función de la BD usada
  - **`generator`**
    - Especifica el nombre del generador que se usará en el caso de la estrategia **`SEQUENCE`**

Anotación de Clases Persistentes (13)

- **`@SequenceGenerator`**
  - Especifica información de mapping para un generador que se puede usar con la estrategia **`SEQUENCE`**
  - **`name`**: nombre del generador
  - **`sequenceName`**: nombre de la secuencia
- Los atributos/propiedades anotadas tienen que ser de tipo entero (**`short`**, **`int`**, **`long`** o sus contrapartidas objetuales)

Anotación de Clases Persistentes (14)

- ¿Cómo especificar que utilice secuencias o columnas contador para la propiedad **`accountId`**?
  - En el ejemplo se ha especificado

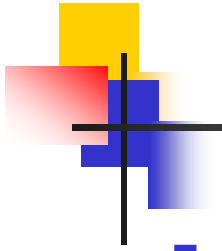
```
@GeneratedValue(strategy=GenerationType.AUTO,
                generator="AccountIdGenerator")
```

    - Hibernate utiliza la estrategia más apropiada a la BD usada
    - Por ejemplo, para MySQL utiliza **`IDENTITY`**, y para PostgreSQL utiliza **`SEQUENCE`** (y el generador especificado)
    - Esto evita tener que especificar explícitamente (en función de la BD)

```
@GeneratedValue(strategy=GenerationType.IDENTITY)
  o
@GeneratedValue(strategy=GenerationType.SEQUENCE,
                generator="AccountIdGenerator")
```

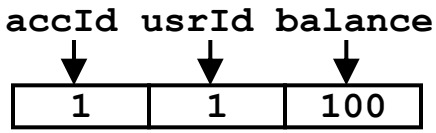
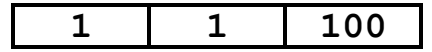
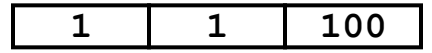
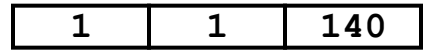
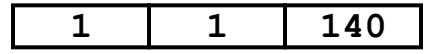
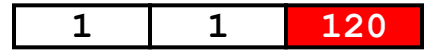
- El constructor con argumentos deja el atributo **`accountId`** sin inicializar
  - La implementación de los casos de uso invoca este constructor cuando tiene que crear una instancia de **`Account`**
  - No puede dar valor al atributo **`accountId`**: es Hibernate el que le asignará valor tras hacer el objeto persistente (invocando el método **`setAccountId`**)

Anotación de Clases Persistentes (15)



- Hibernate asume que por defecto no se utiliza un nivel de aislamiento en sus transacciones mayor que **`TRANSACTION_READ_COMMITED`**
  - No se evitan "non-repeatable reads" y "phantom reads"
    - Problema si las transacciones releen datos de la BD
- E.g.: Realización de dos ingresos concurrentes sobre la misma cuenta de un usuario (no consideramos operaciones bancarias para simplificar)
  - Transacción 1: añade **`40€`** a la cuenta **`1`** del usuario **`1`**
  - Transacción 2: añade **`20€`** a la cuenta **`1`** del usuario **`1`**

Anotación de Clases Persistentes (16)

Transacción 1	Transacción 2	Estado en BD
`begin`		
`SELECT balance` /* balance = 100 */		
/* Calcular nuevo balance en memoria, balance = **140** */	`begin`	
`UPDATE balance` /* balance = 140 en BD pero no comprometido */	`SELECT balance` /* balance = 100 */	
`commit` /* balance = 140 en BD y comprometido */	/* Calcular nuevo balance en memoria, balance = **120** */	
	`UPDATE balance` /* balance = 120 en BD pero no comprometido */	
	`commit` /* balance = 120 en BD y comprometido */	

Anotación de Clases Persistentes (17)

- Problema: hemos perdido los efectos de la primera de las transacciones
- Este problema es conocido con el nombre de **second lost update**
  - Es un caso particular del problema "non-repeatable reads”
- Es posible resolverlo sin recurrir a **`TRANSACTION_REPEATABLE_READ`** (o superior) usando **`TRANSACTION_READ_COMMITED`** y la estrategia **Optimistic Locking**
  - En general, es la estrategia recomendada por Hibernate, dado que maximiza la eficiencia (menor nivel de aislamiento => menos bloqueos en la BD) y es adecuada para la mayor parte de transacciones
  - Las transacciones que lanzan más de una vez una misma consulta, tienen que emplear un nivel de aislamiento mayor

Anotación de Clases Persistentes (18)

- Optimistic Locking
  - Hibernate permite utilizar Optimistic Locking en la ejecución de transacciones para resolver este problema
    - Se añade un atributo/propiedad **`version`** en las entidades modificables, anotándolas con **`@Version`**
    - Cada vez que se actualiza una instancia de una entidad, Hibernate comprueba si el atributo **`version`** coincide con su valor actual
      - En caso afirmativo, realiza la actualización y lo incrementa en una unidad
      - En otro caso, lanza una excepción de **`Runtime`** (**`org.hibernate.StaleObjectStateException`** o **`org.hibernate.StaleStateException`**)

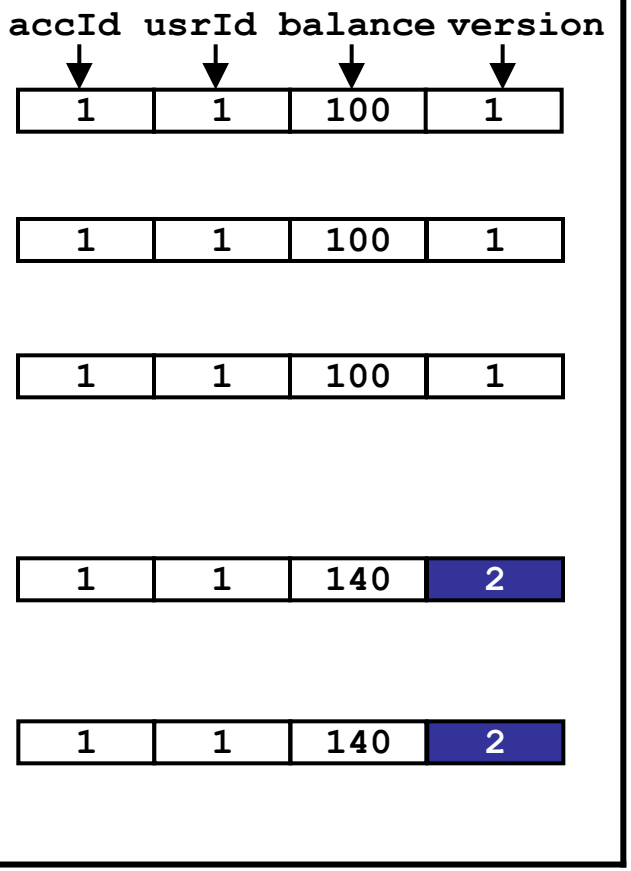
Anotación de Clases Persistentes (19)

- Optimistic Locking (cont)
  - El desarrollador no debe modificar el valor de este atributo/propiedad
  - Tipos de atributos/propiedades a las que es aplicable: **`short`**, **`int`**, **`long`** y sus contrapartidas objetuales, y **`java.sql.Timestamp`**
  - Hibernate implementa una actualización de la siguiente forma

```
UPDATE Account
   SET usrId = ?, balance = ?, version = version + 1
   WHERE accId = ? AND version = ?
```

    - Si devuelve 1 => se actualizó la fila (y en consecuencia, no hay conflictos con otra transacción)
      - **`version`** se incrementa en 1 (**`version = version + 1`**) en la fila y en la instancia de **`Account`** en memoria
    - Si devuelve 0 => No se actualizó la fila (otra transacción actualizó antes esa fila)
      - El valor de la columna **`version`** era superior al que tenía la instancia de **`Account`** en memoria
      - Se produciría la excepción de **`Runtime`**

Anotación de Clases Persistentes (20)

Transacción 1	Transacción 2	Estado en BD
`begin`		
`SELECT balance, version` /* balance = 100, version = 1 */		
/* Calcular nuevo balance en memoria, balance = 140 */	`begin`	

`UPDATE balance, version`
`WHERE version = 1`
/* balance = **140**, version = **2** en BD pero no comprometido */
	`SELECT balance, version`
	/* balance = 100, version = 1 */	

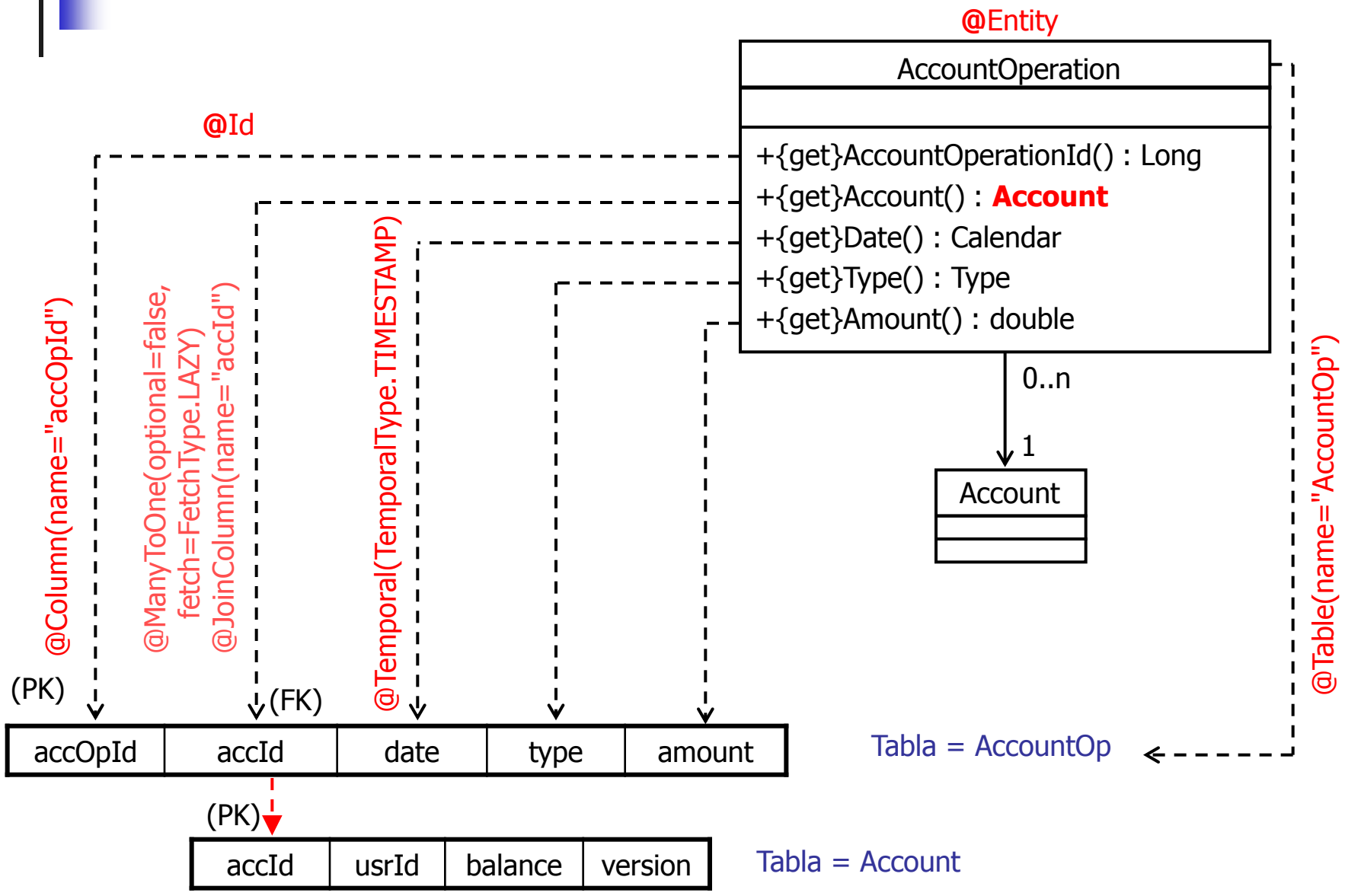
`commit` /* balance = 140, version =2 en BD y comprometido */	/* Calcular nuevo balance en memoria, balance = **120** */	

	`UPDATE balance, version`
	`WHERE version = 1`
	`->`
	`StaleObjectStateException`	

Anotación de Clases Persistentes (21)

- Optimistic Locking (cont)
  - La estrategia es optimista porque
    - No realiza un bloqueo especial sobre las filas que se leen durante la transacción
    - Retrasa al final de la transacción la comprobación de si hubo conflictos con otras transacciones
  - La estrategia Optimistic Locking también es útil para implementar **conversaciones**
    - Las conversaciones permiten englobar en una única transacción de 'usuario' varios casos de uso invocados por un mismo usuario para realizar una tarea concreta (más información en "Java Persistence with Hibernate")

Anotación de Clases Persistentes (22)



Anotación de Clases Persistentes (23)

```
@Entity
@org.hibernate.annotations.Entity(mutable=false)
@Table(name="AccountOp")
public class AccountOperation {

    public enum Type {ADD, WITHDRAW};

    private Long accountOperationId;
    private Account account;
    private Calendar date;
    private Type type;
    private double amount;

    public AccountOperation() {}

    public AccountOperation(Account account, Calendar date,
        Type type, double amount) {
       /**
        * NOTE: "accountOperationId" *must* be left as "null" since
        * its value is automatically generated.
        */
        this.account = account;
        this.date = date;
        this.type = type;
        this.amount = amount;
    }
```

Anotación de Clases Persistentes (24)

```
@Column(name="accOpId")
@SequenceGenerator(                          // It only takes effect
     name="AccountOperationIdGenerator",     // for databases provi-
                                             // ding intentifier
     sequenceName="AccountOpSeq")            // generators.
@Id
@GeneratedValue(strategy=GenerationType.AUTO,
                generator="AccountOperationIdGenerator")
public Long getAccountOperationId() {
    return accountOperationId;
}

public void setAccountOperationId (Long accountOperationId) {
    this.accountOperationId = accountOperationId;
}

@ManyToOne(optional=false, fetch=FetchType.LAZY)
@JoinColumn(name="accId")
public Account getAccount() {
    return account;
}

public void setAccount(Account account) {
    this.account = account;
}
```

Anotación de Clases Persistentes (25)

```
    @Temporal(TemporalType.TIMESTAMP)
    public Calendar getDate() {
        return date;
    }

    public void setDate(Calendar date) {
        this.date = date;
    }

    public Type getType() {
        return type;
    }

    public void setType(Type type) {
        this.type = type;
    }

    public double getAmount() {
        return amount;
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }
}
```

Anotación de Clases Persistentes (26)

- Utiliza anotaciones especiales para la generación del identificador numérico para la clave primaria, como **`Account`**
- **`AccountOperation`** es una entidad que nunca es modificada
  - **`@org.hibernate.annotations.Entity(mutable=false)`**
    - Esta anotación permite especificar esta situación a Hibernate, para aplicar optimizaciones de tiempo de ejecución
  - Por tanto, no define la propiedad (ni la anotación) **`version`**
- **`@Table`**
  - Por defecto, cada entidad se mapea a una tabla con el mismo nombre que la entidad
  - **`@Table`** permite especificar un nombre particular para la tabla, en este caso **`AccountOp`**
- **`@Temporal`**
  - Para los atributos/propiedades de tipo **`Calendar`** hay que especificar el mapping que se desea
    - **`TemporalType.DATE`**, **`TemporalType.TIME`** o **`TemporalType.TIMESTAMP`**

Anotación de Clases Persistentes (y 27)

- Las enumeraciones (en este caso **`Type`**) se almacenan en BD como **`String`** o como un ordinal, dependiendo del tipo de la columna de la BD con la que se mapea
  - Se puede utilizar la anotación **`@Enumerated`** para forzar una de las estrategias:
  - **`@Enumerated(EnumType.ORDINAL)`**
    - Utiliza la posición de la opción en la enumeración
  - **`@Enumerated(EnumType.STRING)`**
    - Utiliza el nombre de la constante en la enumeración
- En nuestro caso, como se mapea con una columna numérica en BD, se utiliza la estrategia **`ORDINAL`**

Anotando Relaciones (1)

- Tipos de relaciones
  - Uno-a-Uno, Uno-a-Muchos/Muchos-a-Uno, Muchos-a-Muchos
    - Uno significa 0..1 o 1; Muchos significa 0..N o N
- Atributos/propiedades que representan relaciones
  - Tipos: clases persistentes o colecciones de clases persistentes (utilizando las interfaces **`Collection`**, **`List`**, **`Set`** y **`Map`**)
  - Se utilizan anotaciones para especificar cómo mapear las relaciones a columnas/tablas
- Direccionalidad
  - Unidireccionales o Bidireccionales
- **Lado propietario** ("*owning side*") y **lado inverso** ("*inverse side*") en una relación
  - El lado propietario es la entidad cuya tabla asociada tiene la clave foránea que mantiene la relación
  - Una relación unidireccional sólo tiene lado propietario
    - El que permite navegar hacia la otra entidad
  - En una relación bidireccional
    - Si es Uno-a-Muchos o Muchos-a-Uno, el lado propietario es el lado Muchos
    - Si es Uno-a-Uno, la entidad cuya tabla contiene la clave foránea es el lado propietario
    - Si es Muchos-a-Muchos, cualquier lado puede ser el lado propietario

Anotando Relaciones – 1:N (2)

- Entre **`Account`** y **`AccountOperation`** existe una relación unidireccional, Muchos-a-Uno, que es necesario mapear a BD
  - Desde **`AccountOperation`** se puede recuperar el objeto **`Account`** al que pertenece la operación
  - En cambio desde **`Account`** no se pueden recuperar los objetos **`AccountOperation`** asociados

```
Account (1) <-- AccountOperation (0..N)
```

- Como ya se ha comentado en el tema 3.2
  - No se ha modelado la relación **`Account`** --> **`AccountOperation`** porque es posible que una cuenta tenga muchas operaciones, y no es factible recuperarlas todas juntas
  - Para recuperar las operaciones de una cuenta se utiliza el patrón *page by page iterator*

Anotando Relaciones – 1:N (3)

- Relaciones Uno-a-Muchos / Muchos-a-Uno
  - Se utiliza **`@OneToMany`** (lado Uno) o **`@ManyToOne`** (lado Muchos) sobre los atributos/propiedades que definen la relación
  - Si la relación es unidireccional, sólo se anota el lado que permite navegar hacia el otro
    - En el caso **`Account <-- AccountOperation`**, la anotación se realiza sobre el lado Muchos (**`AccountOperation`**) sobre el método **`getAccount`**

```
@ManyToOne(optional=false)
@JoinColumn(name="accId")
```

  - **`optional=false`** (por defecto **`true`**)
    - Especifica que una operación siempre está asociada a una cuenta
  - Se utiliza **`@JoinColumn`** sobre el atributo/propiedad que mantiene la relación, para especificar la columna que actúa como clave foránea
    - En el caso de **`AccountOperation`** es **`accId`**

Anotando Relaciones – 1:N (4)

- Relaciones Uno-a-Muchos / Muchos-a-Uno
  - Si la relación tratada fuese bidireccional

    `Account (1) <--> AccountOperation (0..N)`

    en **`Account`** habría que añadir

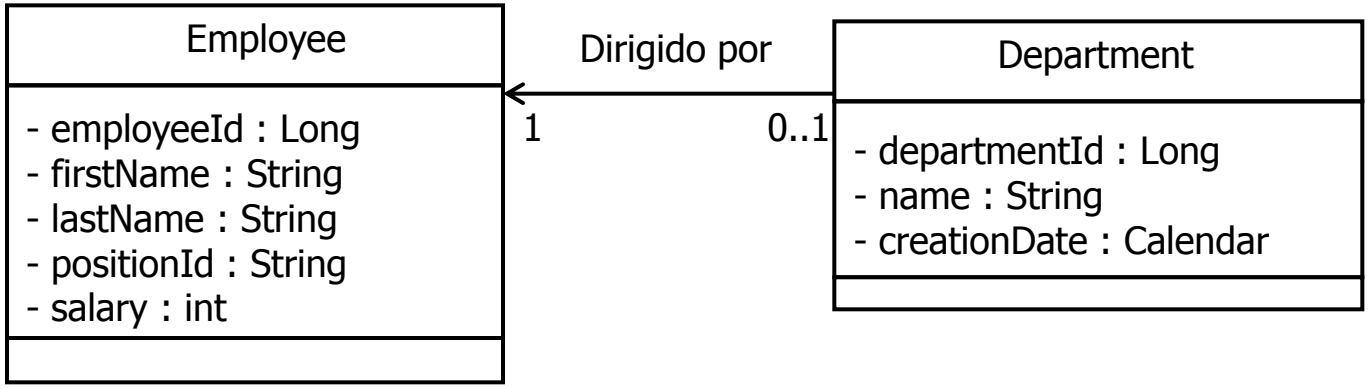
```
public class Account {

    // ...
    private Set<AccountOperation> accountOperations;

    @OneToMany(mappedBy = "account")
    public Set<AccountOperation> getAccountOperations() {
        return accountOperations;
    }

    public void setAccountOperations(
            Set<AccountOperation> accountOperations) {
        this.accountOperations = accountOperations;
    }
    // ...
}
```

Anotando Relaciones – 1:N (5)

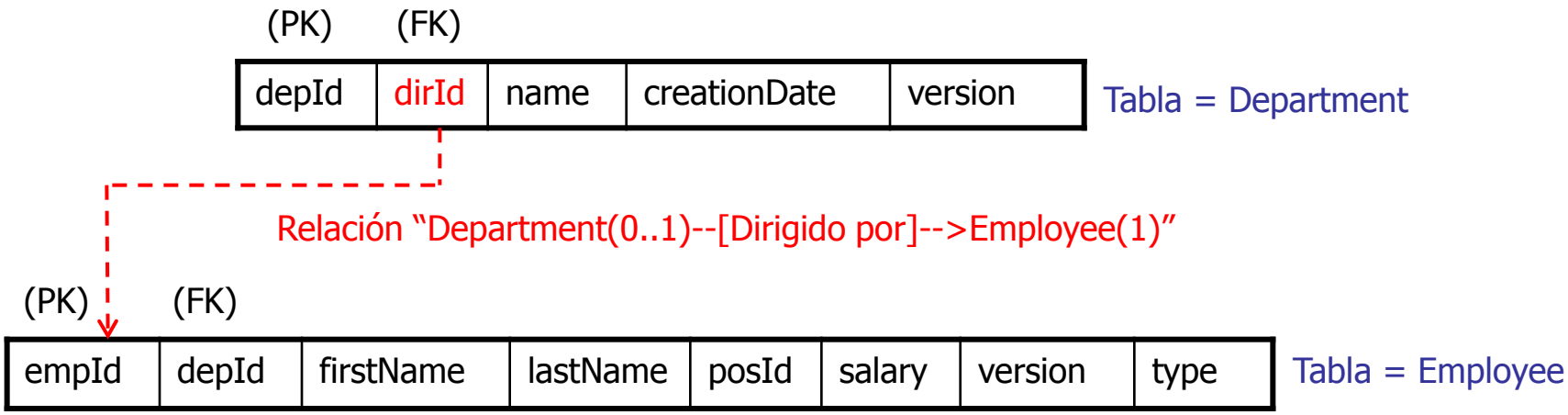
- En las relaciones bidireccionales, el lado inverso tiene que usar el elemento **`mappedBy`** en **`@OneToOne`**, **`@OneToMany`** y **`@ManyToMany`**
  - No se puede usar en **`@ManyToOne`** porque en una relación bidireccional el lado Muchos siempre es el lado propietario
  - **`mappedBy`** especifica el nombre del atributo/propiedad del otro lado (lado propietario) de la relación
  - En el ejemplo, el elemento **`mappedBy`** está diciendo que la propiedad **`account`** de **`AccountOperation`** conjuntamente con la propiedad **`accountOperations`** de **`Account`** forman una relación bidireccional

Anotando Relaciones – 1:1 (6)

- Clases Persistentes (ejemplo completo apartado 3.6)



- Tablas



Anotando Relaciones – 1:1 (7)

- **Department(0..1)--[Dirigido por]--> Employee(1)**
  - En **Department**

```
@Entity
public class Department {

    // ...

    @OneToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="dirId")
    public Employee getDirector() {
        return director;
    }

    public void setDirector(Employee director) {
        this.director = director;
    }

    // ...

}
```

Anotando Relaciones – 1:1 (y 8)

- Relaciones Uno-a-Uno
  - Se utiliza **`@OneToOne`** sobre los atributos/propiedades que definen la relación
    - En el ejemplo, dado que la relación es unidireccional, sólo se aplica sobre el método **`getDirector`** de la entidad **`Department`** (en otro caso, se aplicaría en ambas entidades)
  - Se utiliza **`@JoinColumn`** sobre el atributo/propiedad que define la relación en el lado propietario
    - En el ejemplo, dado que la relación es unidireccional, el lado propietario es **`Department`**
    - Especifica que la columna **`dirId`** actúa como clave foránea para mantener la relación
    - Se puede usar el elemento **`referencedColumnName`** para especificar el nombre de la columna a la que hace referencia la clave foránea
      - Por defecto se asume que es la clave primaria de la tabla de la otra entidad

Gestión de Relaciones (1)

- Cuando una entidad tiene una relación con una o varias entidades, es importante por temas de eficiencia conocer cómo gestiona Hibernate la recuperación de las instancias relacionadas
- Hibernate por defecto utiliza una estrategia **`LAZY`** para relaciones, excepto en los casos Muchos-a-Uno y Uno-a-Uno (que utiliza una estrategia **`EAGER`** por compatibilidad con JPA)
  - **`LAZY`** – Aplaza la recuperación de la entidad hasta que es necesaria
  - **`EAGER`** – Cuando se recupera un objeto persistente, se recuperan también todos los objetos persistentes que presenten este tipo de asociación

Gestión de Relaciones (2)

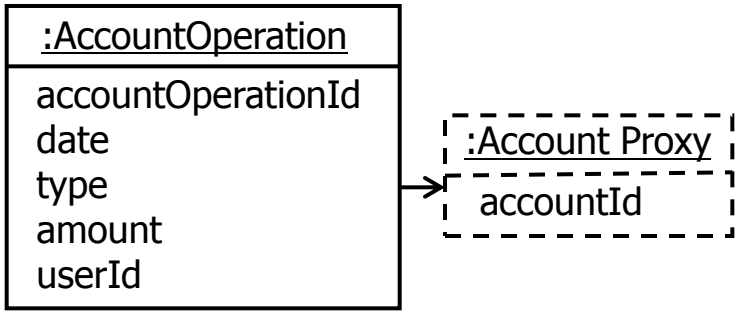
- ¿Qué ocurre cuando se invoca al método **`get`** que permite atravesar una relación **`LAZY`**?
  - Si la relación es Uno/Muchos-a-Uno
    - Hibernate devuelve un proxy del objeto relacionado que contiene la clave primaria de la entidad
    - Los proxies los genera Hibernate dinámicamente en tiempo 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
    - Si se invoca cualquier método sobre el proxy, excepto **`get<<Clave>>`**, Hibernate inicializa el proxy (es decir, recupera el estado de BD)
  - Si la relación es Uno/Muchos-a-Muchos
    - Hibernate devuelve una colección que implementa la interfaz usada (**`List`**, **`Set`**, etc.) para modelar la relación
    - La colección se inicializa (es decir, recupera las instancias relacionadas de BD) cuando se invoca alguna operación de la colección (**`size`**, **`contains`**, iterar, etc.)

Gestión de Relaciones (y 3)

- E.g. Muchos-a-Uno: **`(Account <-- AccountOperation)`**
  - Por defecto, cuando Hibernate obtiene una operación, recupera automáticamente su cuenta asociada - **`EAGER`**
  - Sin embargo, se ha utilizado el modificador **`fetch`** en la anotación de la relación para especificar que la estrategia a seguir para recuperar las instancias relacionadas sea **`LAZY`**

    ```
    @ManyToOne(optional=false, fetch=FetchType.LAZY)
    ```

    - La documentación de Hibernate aconseja utilizar siempre LAZY para todas las relaciones, para evitar que cada vez que se carga un objeto persistente en memoria se tengan que recuperar los relacionados, lo que podría desencadenar demasiadas consultas o consultas con muchos JOINs



Gestión de la Persistencia (1)

- **`org.hibernate.Session`**
  - Constituye la principal abstracción de la API de Hibernate
  - Incluye soporte para las operaciones CRUD básicas sobre entidades
  - Permite construir búsquedas complejas mediante el objeto **`Query`**
  - Permite gestionar transacciones mediante el objeto **`Transaction`**
  - Representa un gateway contra la BD
    - Mantiene internamente una conexión contra la BD
    - Contiene un mapa con las entidades que obtiene/almacena/modifica y funciona a modo de caché de objetos persistentes
    - Sincroniza de forma automática el estado de las entidades persistentes con el de la BD
      - Intenta retrasar la sincronización todo lo posible

Gestión de la Persistencia (y 2)

- Operaciones CRUD
  - Insertar un objeto persistente o actualizar uno existente
    - `saveOrUpdate(Object entity)`
  - Eliminar un objeto persistente
    - `delete(Object entity)`
  - Recuperar un objeto persistente a partir de su clase y su clave primaria
    - `get(Class entityClass, Serializable primaryKey)`
      - Devuelve `null` si la instancia no existe
- **`saveOrUpdate`** y **`get`** asocian la instancia persistente con el objeto **`Session`**
- **`delete`** elimina la asociación de la instancia (que deja de ser persistente) con el objeto **`Session`**

Transacción Típica con Hibernate (1)

- El objeto **`Session`** se obtiene a partir de un objeto **`SessionFactory`**, invocando el método **`openSession`**
- Un objeto **`SessionFactory`** representa una configuración particular de un conjunto de metadatos de mapping objeto/relacional

```
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();

    // Utilizar la Session para saveOrUpdate/get/delete/...

    tx.commit();
} catch (Exception e) {
    if (tx != null) {
        tx.rollback();
        throw e;
    }
} finally {
    session.close();
}

// Al finalizar la aplicación ...
HibernateUtil.shutdown();
```

Transacción Típica con Hibernate (y 2)

- Cuando se crea el objeto **`Session`**, se le asigna la conexión de BD que va a utilizar
- Una vez obtenido el objeto **`Session`**, se crea una nueva unidad de trabajo (**`Transaction`**) utilizando el método **`beginTransaction`**
- Dentro del contexto de la transacción creada, se pueden invocar los métodos de gestión de persistencia proporcionados por el objeto **`Session`**, para recuperar, añadir, eliminar o modificar el estado de instancias de clases persistentes. También se pueden realizar consultas
- Si las operaciones de persistencia no han producido ninguna excepción, se invoca el método **`commit`** de la unidad de trabajo para confirmar los cambios realizados. En caso contrario, se realiza un **`rollback`** para deshacer los cambios producidos
- Sobre un mismo objeto **`Session`** pueden crearse varias unidades de trabajo
- Finalmente se cierra el objeto **`Session`** invocando su método **`close`**

Ejemplos de Gestión de Persistencia

- Los siguientes ejemplos utilizan la plantilla definida en la transparencia anterior
- **es.udc.pojo.minibank.test.experiments.hibernate.***
  - **CreateAccount**
    - Crea una nueva cuenta para el usuario con identificador 1 y balance inicial 200

```
Account account = new Account(1, 200);
session.saveOrUpdate(account);
```

  - **FindAccount <accountId>**
    - Encuentra una cuenta a partir de su identificador, que recibe como parámetro

```
Account account =
   (Account) session.get(Account.class, accountId);
```

  - **RemoveAccount <accountId>**
    - Elimina una cuenta a partir de su identificador, que recibe como parámetro

```
Account account = (Account)
    session.get(Account.class, accountId);
if (account != null) {
     session.delete(account)
}
```

Obtención de la `SessionFactory` (1)

```
public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            sessionFactory = new AnnotationConfiguration().
                configure("hibernate-tutorial-hibernate-config-test.xml").
                buildSessionFactory();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public static void shutdown() {
        getSessionFactory().close();
    }

}
```

Obtención de la `SessionFactory` (y 2)

- La forma habitual de inicializar Hibernate es creando un objeto **`SessionFactory`** a partir de un objeto **`AnnotationConfiguration`**
- Un objeto **`AnnotationConfiguration`** es una representación objetual del fichero de configuración de Hibernate, que utiliza las anotaciones definidas en las clases persistentes para obtener los mappings objeto/relacional
- Cuando se invoca el método **`configure`** sin argumentos de **`AnnotationConfiguration`**, Hibernate busca un fichero con nombre **`hibernate.cfg.xml`** en el CLASSPATH de la aplicación
  - Existe otra versión de **`configure`** que permite especificar otro nombre de fichero
- La inicialización del objeto **`SessionFactory`** es un proceso costoso, que suele hacerse sólo una vez al comienzo de la aplicación
- Normalmente se utiliza una clase, **`HibernateUtil`**, para implementar la instancia única de este objeto
  - En un bloque estático se inicializa la instancia de **`SessionFactory`** que utilizará la aplicación
  - El método **`getSessionFactory`** devuelve siempre la misma instancia de la clase **`SessionFactory`**
  - El método **`shutdown`** cierra la instancia única de **`SessionFactory`** y libera todos sus recursos (información de mapping, pool de conexiones, etc.)

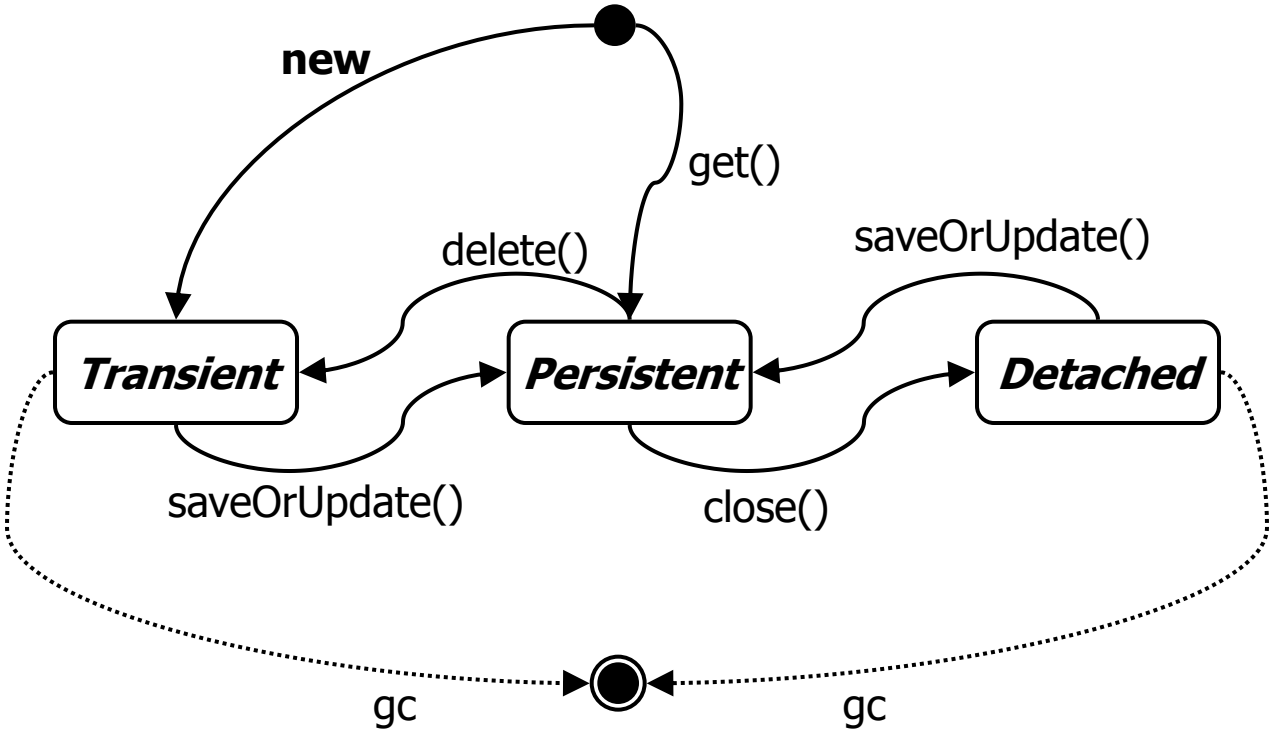
Fichero de Configuración (1)

```
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property
      name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property
      name="hibernate.connection.url">jdbc:mysql://localhost/pojotest</property>
    <property name="hibernate.connection.username">pojo</property>
    <property name="hibernate.connection.password">pojo</property>
    <!-- JDBC connection pool (use the built-in) -->
    <property name="hibernate.connection.pool_size">1</property>
    <!-- SQL dialect -->
    <property
      name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

    <!-- IMPORTANT: for efficiency, in a real deployment the following
         debug properties must be commented or removed. -->
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.format_sql">true</property>
    <property name="hibernate.use_sql_comments">true</property>

    <!-- List of persistent classes -->
    <mapping class="es.udc.pojo.minibank.model.account.Account" />
    <mapping
       class="es.udc.pojo.minibank.model.accountoperation.AccountOperation" />
  </session-factory>
</hibernate-configuration>
```

Fichero de Configuración (y 2)

- Fichero de configuración de Hibernate – **`hibernate.cfg.xml`**
  - Configuración de Conexión a BD (data source)
    - **hibernate.connection.driver_class, hibernate.connection.url, hibernate.connection.username, hibernate.connection.password**
  - Dialecto hibernate
    - **hibernate.dialect**
      - Especifica qué variante de SQL tiene que generar para comunicarse con la BD
      - Incluye dialectos para multitud de BBDD
  - Configuración pool de conexiones (opcional)
    - **hibernate.connection.pool_size**
      - Soporte para integrar cualquier pool de conexiones; incluye soporte nativo para C3P0
  - Propiedades para habilitar *logging* (deshabilitado en un entorno en producción)
    - **hibernate.show_sql**
      - Habilita mostrar todas las sentencias SQL ejecutadas por Hibernate por consola
    - **hibernate.format_sql**
      - Hace que la salida sea más legible, pero ocupa más espacio en pantalla
    - **hibernate.use_sql_comments**
      - Hibernate añade comentarios a todas las sentencias SQL generadas para explicar su origen
  - Mapping de entidades
    - Lista de clases anotadas o de ficheros XML definiendo mappings

Ciclo de Vida de una Entidad (1)



Ciclo de Vida de una Entidad (y 2)

- Un objeto es **persistent** si está asociado a una **`Session`** y tiene correspondencia en BD
  - Un objeto en este estado puede formar parte en transacciones
  - Si durante la transacción se realiza algún cambio sobre él (invocación de un método **`set`**), Hibernate actualiza automáticamente su estado en BD antes de terminar la transacción
- Un objeto es **detached** si no está asociada a una **`Session`** pero tiene correspondencia en BD
- Un objeto es **transient** si no está asociado a una **`Session`** y no tiene correspondencia en BD

Lenguaje de Consultas (1)

- La **`Session`** también proporciona acceso a la API de consultas de Hibernate
  - **`session.createQuery(hqlQuery)`**
    - HQL (Hibernate Query Language)
    - Permite realizar consultas similares a SQL, en términos de clases persistentes y sus propiedades
  - **`session.createCriteria(entityClass)`**
    - API de Programación para QBC (Query By Criteria)
    - QBC es una alternativa a HQL, que al igual que ésta permite lanzar consultas, pero a diferencia de ella, no es un lenguaje de consultas, sino una API que permite expresar consultas en términos de objetos (existen objetos para hacer JOINs, restricciones, proyecciones, etc.)
  - **`session.createSQLQuery(sqlQuery)`**
    - SQL (Structured Query Language)
    - Sólo debe de utilizarse cuando las otras opciones no son válidas (e.g., cuando se necesite utilizar una característica propia del SQL nativo de la BD)

Lenguaje de Consultas – HQL (2)

- Hibernate Query Language (HQL)
  - Sintaxis parecida a SQL (para que sea fácil de aprender)
  - No usa nombres de tablas ni columnas, sino referencias a objetos y propiedades
  - E.g.: Encontrar las cuentas del usuario con identificador 1

    ```
    SELECT a FROM Account a WHERE a.userId = 1
    ```

    - **`Account`** es el nombre de la entidad
    - **`userId`** es el nombre de una propiedad de la entidad
    - La implementación de Hibernate
      - Averigua el nombre de la tabla y las columnas correspondientes a las propiedades haciendo uso de las anotaciones insertadas en la clase **`Account`**
      - Traduce la consulta HQL a una consulta SQL
      - La ejecuta vía JDBC
      - Recupera las filas y devuelve una lista de objetos **`Account`**

Lenguaje de Consultas – HQL (3)

- Ejemplos de utilización de la API de consultas HQL
  - **`findByUserId`**: Encontrar las cuentas de un usuario

```
getSession().createQuery(
    "SELECT a FROM Account a WHERE " +
    "a.userId = :userId ORDER BY a.accountId").
    setParameter("userId", userId).
    setFirstResult(startIndex).
    setMaxResults(count).list();
```

  - **`findAccountOperationsByDate`**: Encontrar las operaciones bancarias para una cuenta, entre dos fechas

```
getSession().createQuery(
    "SELECT o FROM AccountOperation o WHERE " +
    "o.account.accountId = :accountId AND " +
    "o.date >= :startDate AND o.date <= :endDate ORDER BY o.date").
    setParameter("accountId", accountId).
    setCalendar("startDate", startDate).
    setCalendar("endDate", endDate).
    setFirstResult(startIndex).
    setMaxResults(count).
    list();
```

Lenguaje de Consultas – HQL (y 4)

- El método **`createQuery`** del objeto **`Session`** devuelve un objeto **`Query`**
  - Las cadenas **`:XXX`** son "parámetros nombrados" y se les puede dar valor con los métodos **`setParameter`**
    - Devuelven el propio objeto **`Query`** otra vez, para permitir escribir de manera "compacta" la construcción de la consulta y su ejecución
    - En el caso de fechas, hay que usar la variante que recibe un parámetro que especifica si el valor pasado es un **`DATE`**, **`TIME`** o **`TIMESTAMP`** (alternativamente es posible usar el método **`setCalendar`**)
  - Page-by-Page Iterator
    - **`setMaxResults`** permite especificar el número máximo de resultados
    - **`setFirstResult`** permite especificar el índice del primer resultado (de 0 en adelante)
    - Ambos métodos devuelven otra vez el objeto **`Query`**
  - **`list`** permite ejecutar una consulta de lectura y devuelve una lista con los resultados
  - Se puede utilizar **`uniqueResult`** cuando se sabe que sólo habrá un resultado o ninguno (**`null`** en este caso).
    - Lanza la excepción **`NonUniqueResultException`** si la consulta devuelve más de un resultado

Implementación de DAOs con Hibernate (1)

- Un DAO define una interfaz para las operaciones de persistencia (métodos CRUD y de búsqueda) relacionadas con una clase persistente particular
  - Existen métodos comunes a todas las entidades
- Utilizando Generics (características introducida en Java SE 5.0), se puede diseñar un DAO genérico con las *operaciones comunes* para todas las clases persistentes
  - **`save`**
  - **`find`**
  - **`exists`**
  - **`remove`**
- Cada entidad persistente tendrá su propio DAO, que extenderá el genérico para añadir operaciones propias
  - Normalmente añadirá operaciones de búsqueda utilizando diferentes criterios

Implementación de DAOs con Hibernate (2)

- No se ha definido el método **`update`** en el DAO porque Hibernate actualiza los objetos **persistent dirty** automáticamente
- El DAO Genérico asume un ORM con estado, es decir, uno que dispone de una sesión en memoria con los objetos accedidos por el caso de uso (esos objetos son los objetos en estado persistent)
- El ORM sabe si los objetos de la sesión están modificados, si tienen que ser insertados o eliminados, de manera que cuando se hace el **`commit`** (o se fuerza un **`flush`** explícitamente), se lanzan las correspondientes sentencias de actualización/inserción o eliminación
- No es factible definir un DAO completamente genérico
  - Éste es genérico pero asumiendo un ORM con estado
  - En cualquier caso, una vez definido, facilita el desarrollo de los casos de uso (y esa es su principal virtud)

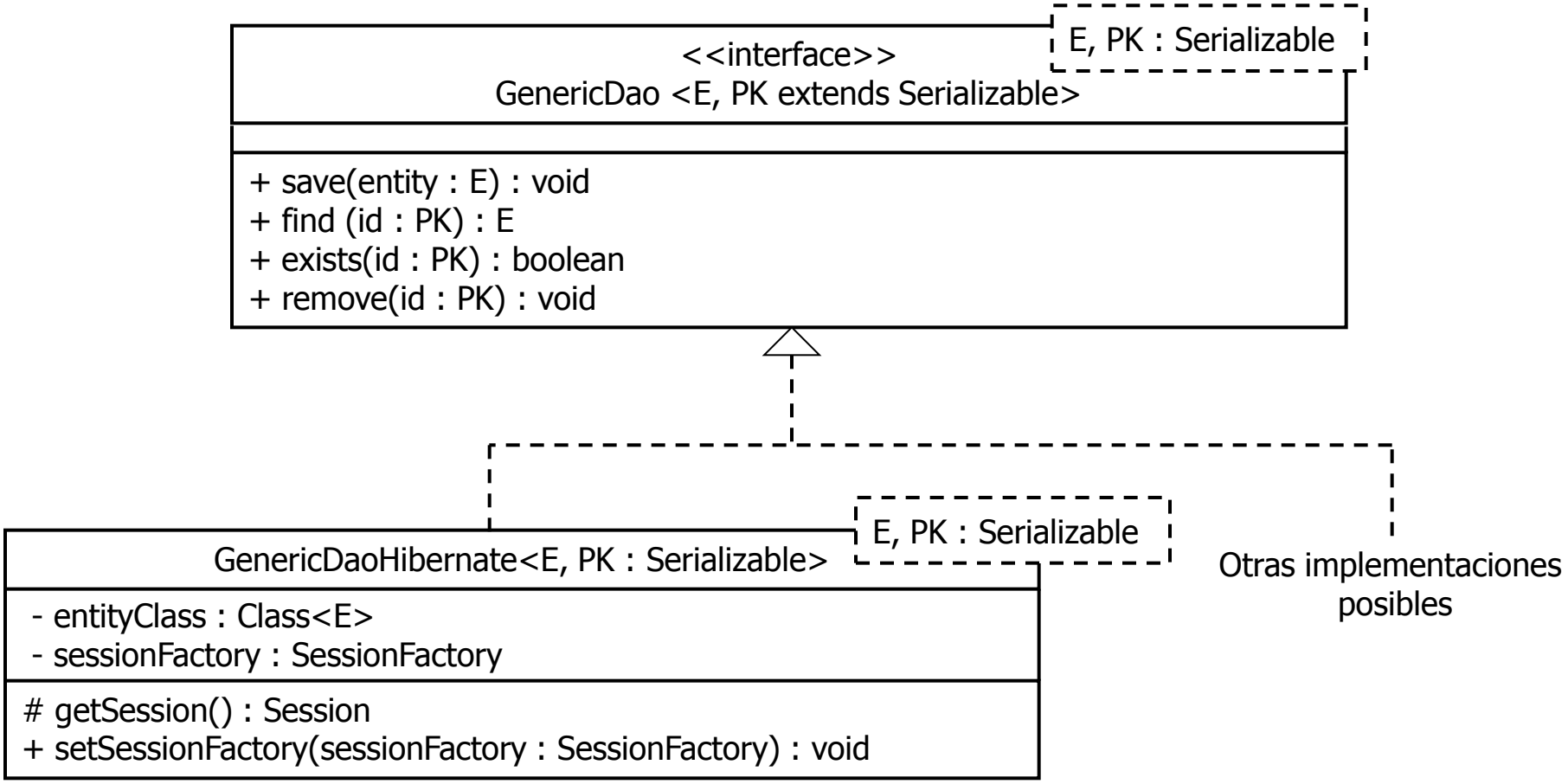
DAO Genérico – Interfaz (3)

- El DAO genérico se encuentra en el módulo **`pojo-modelutil`**
- La interfaz parametrizada del DAO genérico recibe 2 argumentos
  - **`E`**, es la clase persistente para la que se implementará el DAO
  - **`PK`**, define el tipo del identificador de la clase persistente. El identificador debe ser **`Serializable`**
- Los métodos están definidos en base a esos parámetros y no están acoplados a ninguna tecnología de persistencia

```
public interface GenericDao <E, PK extends Serializable>{

    void save(E entity);

    E find(PK id) throws InstanceNotFoundException;

    boolean exists(PK id);

    void remove(PK id) throws InstanceNotFoundException;

}
```

GenericDaoHibernate (4)

- Implementación del DAO genérico con Hibernate



GenericDaoHibernate (5)

```
public class GenericDaoHibernate<E, PK extends Serializable>
        implements GenericDao<E, PK> {

    private SessionFactory sessionFactory;
    private Class<E> entityClass;

    @SuppressWarnings("unchecked")
    public GenericDaoHibernate() {
        this.entityClass = (Class<E>) ((ParameterizedType) getClass().
            getGenericSuperclass()).getActualTypeArguments()[0];
    }

   public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    protected Session getSession() {
        return sessionFactory.getCurrentSession();
    }

    public void save(E entity) {
        getSession().saveOrUpdate(entity);
    }
```

GenericDaoHibernate (6)

```
    @SuppressWarnings("unchecked")
    public E find(PK id) throws InstanceNotFoundException {
        E entity = (E) getSession().get(entityClass, id);
        if (entity == null) {
            throw new InstanceNotFoundException(id, entityClass.getName());
        }
        return entity;

    }

    public boolean exists(PK id) {
        return getSession().createCriteria(entityClass).
            add(Restrictions.idEq(id)).
            setProjection(Projections.id()).
            uniqueResult() != null;
    }

    @SuppressWarnings("unchecked")
    public void remove(PK id) throws InstanceNotFoundException {
        getSession().delete(find(id));
    }

}
```

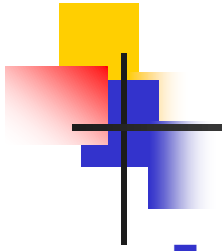
GenericDaoHibernate (7)

- Para implementar la persistencia utilizando Hibernate, el DAO necesita
  - Un objeto **`Session`**. El método **`setSessionFactory`** permite proporcionar la factoría a partir de la cual se obtendrá la sesión actual (a través del método **`getCurrentSession`**)
    - En el apartado 3.4 se utilizará inyección de dependencias para establecer el valor de la propiedad **`sessionFactory`**
  - Conocer la clase persistente que es gestionada por el DAO. En el constructor del DAO se utiliza *Java reflection* para encontrar la clase del argumento genérico **`E`** y almacenarla en la propiedad **`entityClass`**
    - Una clase java parametrizada (**`ParameterizedType`**) dispone de métodos para obtener un array con los tipos de sus argumentos (**`getActualTypeArguments`**). Aquí interesa el tipo del primero de los argumentos (**`0`**)
- NOTA: Es necesario suprimir algunas advertencias del compilador respecto a conversiones "unchecked", debido a que las interfaces de Hibernate son compatibles con Java SE 1.4

GenericDaoHibernate (8)

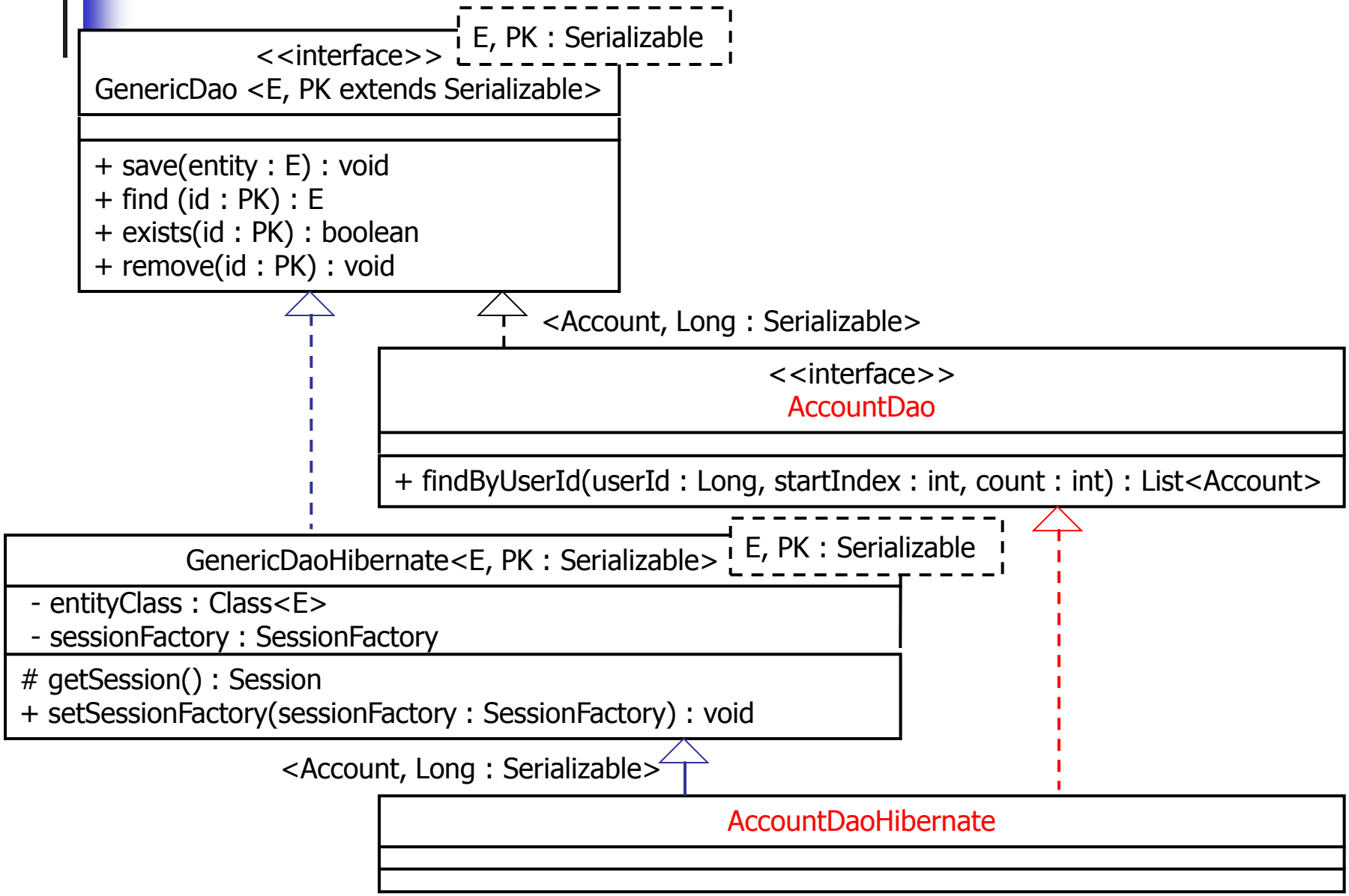
- La implementación de los métodos **`save`**, **`find`** y **`remove`** delega directamente en los métodos de la API de hibernate **`saveOrUpdate`**, **`get`** y **`delete`**
- La implementación del método **`exists`** podría delegar en el método **`get`** de Hibernate, pero como se comentó en el apartado 3.2, por eficiencia se utiliza una **`Query`** especial que sólo recupera la clave primaria del objeto persistente
  - HQL no es suficiente
    - Necesita conocer el nombre de la clave primaria
  - Criteria permite crear consultas de forma programática. En este caso se utiliza para:
    - Añadir una condición por clave primaria (**`Restrictions.idEq(id)`**)
    - Proyectar el campo clave primaria (**`Projections.id()`**)

{Account, AccountOperation}Dao (9)



- Los DAOs específicos para una clase persistente
  - Extienden el DAO genérico proporcionando como argumentos el tipo de clase persistente (**`Account`** o **`AccountOperation`**) y de su clave primaria (**`Long`**)
  - Proporcionan métodos adicionales
    - **`AccountDao`**
      - **`findByUserId`**
    - **`AccountOperationDao`**
      - **`getNumberOfOperations`**
      - **`findByDate`**
      - **`removeByAccountId`**

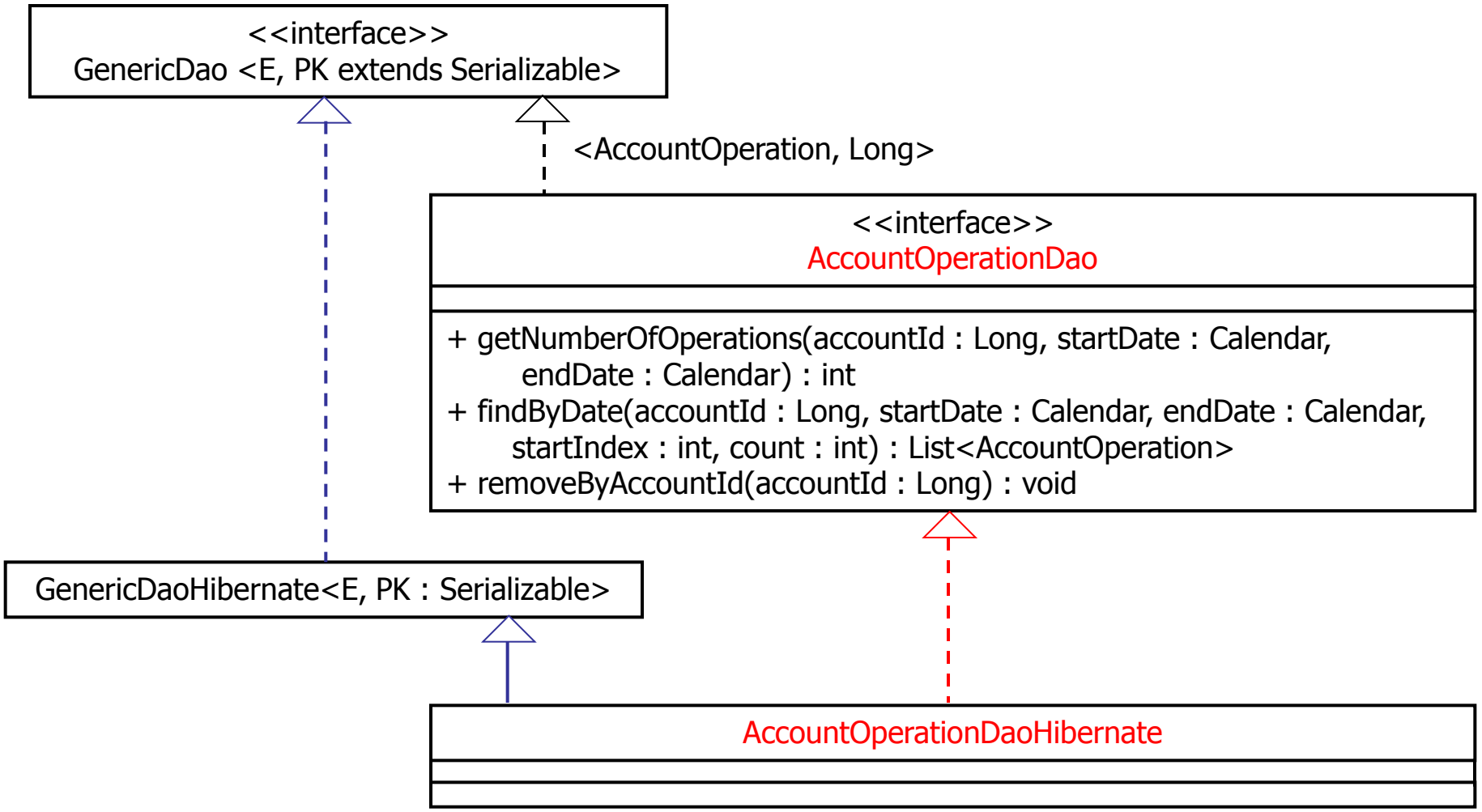
AccountDao (10)



AccountDao (11)

```
public interface AccountDao extends GenericDao<Account, Long> {

    public List<Account> findByUserId(Long userId, int startIndex,
        int count);

}


public class AccountDaoHibernate
   extends GenericDaoHibernate<Account, Long>
   implements AccountDao {

   @SuppressWarnings("unchecked")
    public List<Account> findByUserId(Long userId, int startIndex,
        int count) {

        return getSession().createQuery(
         "SELECT a FROM Account a WHERE a.userId =:userId " +
           "ORDER BY a.accountId").
         setParameter("userId", userId).
         setFirstResult(startIndex).
         setMaxResults(count).list();

    }

}
```

AccountOperationDao (12)



AccountOperationDao (13)

```
public interface AccountOperationDao
    extends GenericDao<AccountOperation, Long> {

    public int getNumberOfOperations(Long accountId,
        Calendar startDate, Calendar endDate);

    public List<AccountOperation> findByDate(Long accountId,
        Calendar startDate, Calendar endDate, int startIndex,
        int count);

    public void removeByAccountId(Long accountId);

}
```

AccountOperationDao (14)

```
public class AccountOperationDaoHibernate extends
    GenericDaoHibernate<AccountOperation, Long>
    implements AccountOperationDao {

    public int getNumberOfOperations(Long accountId,
        Calendar startDate, Calendar endDate) {

        long numberOfOperations = (Long) getSession().createQuery(
           "SELECT COUNT(o) FROM AccountOperation o WHERE "
           + "o.account.accountId = :accountId AND "
           + "o.date >= :startDate AND o.date <= :endDate")
           .setParameter("accountId", accountId)
           .setCalendar("startDate",startDate)
           .setCalendar("endDate", endDate)
           .uniqueResult();

        return (int) numberOfOperations;
}
```

AccountOperationDao (y 15)

```
    @SuppressWarnings("unchecked")
    public List<AccountOperation> findByDate(Long accountId,
        Calendar startDate, Calendar endDate, int startIndex,
        int count) {

        return getSession()
          .createQuery(
          "SELECT o FROM AccountOperation o WHERE "
          + "o.account.accountId = :accountId AND "
          + "o.date >= :startDate AND o.date <= :endDate ORDER BY o.date")
          .setParameter("accountId", accountId)
          .setCalendar("startDate", startDate)
          .setCalendar("endDate", endDate)
          .setFirstResult(startIndex)
          .setMaxResults(count)
          .list();
    }
    ...
}
```

- El método **`removeByAccountId`** se explica en el apartado 3.6

Alcance de la `Session` (1)

- **`WithdrawFromAccount`** (en **`es.udc.pojo.minibank.test.experiments.hibernate`**) implementa el caso de uso de retirar dinero de una cuenta utilizando DAOs

```
AccountDaoHibernate accountDaoHibernate = new AccountDaoHibernate();
accountDaoHibernate.setSessionFactory(HibernateUtil.getSessionFactory());
AccountDao accountDao = accountDaoHibernate;

AccountOperationDaoHibernate accountOperationDaoHibernate =
    new AccountOperationDaoHibernate();
accountOperationDaoHibernate.
    setSessionFactory(HibernateUtil.getSessionFactory());
AccountOperationDao accountOperationDao = accountOperationDaoHibernate;

Transaction tx = HibernateUtil.getSessionFactory().
    getCurrentSession().beginTransaction();

try {
    /* Find account. */
    Account account = accountDao.find(accountId);

    /* Modify balance. */
    double currentBalance = account.getBalance();
```

Alcance de la `Session` (2)

```
        if (currentBalance < amount) {
            throw new InsufficientBalanceException(
                accountId, currentBalance, amount);
        }
        account.setBalance(currentBalance - amount);

        /* Register account operation. */
        accountOperationDao.create(new AccountOperation(account,
            Calendar.getInstance(), AccountOperation.Type.WITHDRAW,
            amount));
        tx.commit();

    } catch (RuntimeException e) {
        e.printStackTrace();
        tx.rollback();
        throw e;
    } catch (InstanceNotFoundException e) {
        e.printStackTrace();
        tx.commit();
    } catch (InsufficientBalanceException e) {
        e.printStackTrace();
        tx.commit();
    } finally {
        HibernateUtil.getSessionFactory().getCurrentSession().close();
    }
    HibernateUtil.shutdown();
```

Alcance de la `Session` (y 3)

- **`WithdrawFromAccount`** (cont)
  - Utiliza dos DAOs, **`accountDao`** y **`accountOperationDao`**
  - Antes de utilizar los DAOs, es necesario configurarlos estableciéndoles el objeto **`SessionFactory`**, a través del método **`setSessionFactory`**
  - Para que el caso de uso sea transaccional, las invocaciones a los métodos del DAO tienen que realizarse dentro de la misma unidad de trabajo: misma instancia de **`Transaction`** en la misma instancia de la **`Session`**
  - El método **`getSession`** del DAO genérico invoca el método **`getCurrentSession`** de **`SessionFactory`** que permite obtener la **`Session`** del contexto de ejecución actual (*Contextual Session*)
    - La **`Session`** se crea la primera vez que se invoca este método dentro de un ámbito definido; el resto de veces devuelve el objeto previamente creado
  - ¿Cuál es el ámbito de la sesión actual? Configurable
    - En nuestro ejemplo se está devolviendo el mismo objeto **`Session`** a todas las invocaciones a **`getCurrentSession`** realizadas sobre el mismo thread
      - Hemos añadido la siguiente propiedad al fichero de configuración de hibernate

```
<!-- Enable Hibernate's automatic session context management -->
<property name="current_session_context_class">thread</property>
```

Conclusiones

- La API Hibernate simplifica la gestión de la persistencia
- Hemos encapsulado la lógica de persistencia utilizando el patrón DAO
- Hemos desarrollado un DAO genérico que reutilizaremos para implementar todos los DAOs
- En el apartado 3.4
  - Se utilizará inyección de dependencias (Spring) para proporcionar un objeto `SessionFactory` a los DAOs e inyectar los propios DAOs en los servicios
  - La gestión de transacciones y sesiones será transparente a los DAOs, por eso no se ha considerado en su implementación
  - El DAO genérico y las diferentes implementaciones utilizan una clase utilidad para traducir las excepciones que devuelve Hibernate en excepciones propias de Spring (independientes del framework de persistencia utilizado)