Corso di “Sviluppo di applicazioni Web”

Docente: Lorenzo Gallucci

- Hibernate: gerarchie di classi ed associazioni

Mapping in Hibernate

- Hibernate comprende un set di istruzioni di mapping, per mezzo di annotazioni, conformi allo standard JPA
- Sebbene una gran parte delle capacità di Hibernate siano sfruttabili per mezzo di annotazioni JPA, alcune particolarità del framework sono accessibili solo tramite annotazioni specifiche di Hibernate
  - Ad esempio, ottimizzazioni, come la dimensione di fetch, ecc.
- È possibile mescolare annotazioni JPA con annotazioni Hibernate
- Su quali classi si può operare?

Unità di persistenza (DTO)

- Hanno caratteristiche simili ai Javabean:
  - Deve essere esplicitamente definito un costruttore pubblico
  - Nessun metodo deve essere *final*
  - La classe non deve essere *final*
  - Per ogni proprietà sono richiesti I metodi get e set (“is” è accettabile se si lavora con proprietà booleane)
- Poichè il modello ad oggetti deve avere validità di per sè, è utile non esporre le Collection
  - Get di un campo collection dovrebbe restituire un oggetto immutabile (Collections.unmodifiable…())
  - Set va evitato
  - Possono essere inseriti dei “convenience method” per I comportamenti che si vogliono rendere disponibili
    - Ad es., un campo Collection<B> in A potrebbe disporre di un metodo addB() che, se presente, aggiorna un puntatore verso A in B

Unità di persistenza (DTO)

- Hibernate può accedere alle proprietà del DTO tramite
  - Invocazione dei metodi get/set
  - Accesso diretto ai campi del DTO
  - *Come può avvenire?*
- L'una o l'altra modalità di accesso sono attivate dall'annotazione @Id, presente sulle entità
  - Se @Id è specificato su un metodo get, a tutte le proprietà si accede tramite i relativi metodi get/set
  - Se @Id è specificato su un campo, a tutte le proprietà si accede tramite i campi che le rappresentano (saltando possibili validazioni inserite nei metodi set)
  - In Hibernate è possibile intervenire in questo automatismo tramite l'annotazione @AccessType

Unità di persistenza (DTO)

- Annotazioni sulle entità
- @Entity
  - Indica che la classe è una entità sottoposta a persistenza
- @Table
  - Può essere usato per indicare catalog, schema e name della tabella
  - @UniqueConstraint
- Su ogni campo/proprietà:
  - @Id
    - Indica che una proprietà rappresenta la chiave della tupla
    - Può essere usato @GeneratedValue per specificare una politica di generazione automatica della chiave
  - @Column
    - Consente di modificare il mapping generato per default su una proprietà
      - Alcune opzioni: insertable, updatable, nullable, name, ecc.
  - @Lob
    - Richiede una gestione di tipo CLOB/BLOB per una proprietà
  - @Temporal
    - Indica il tipo di dato temporale da associare alla proprietà

Gerarchie di classi

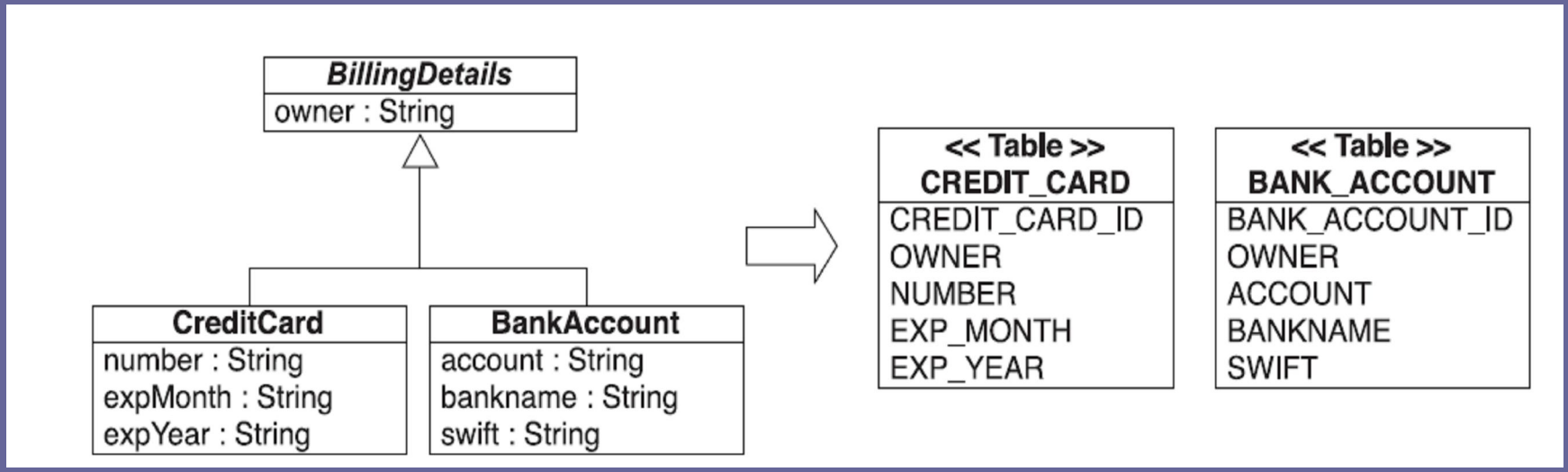
- Possono essere gestite con 4 approcci differenti:
  - Table per concrete class con mapping del polimorfismo
    - Non si asserisce nulla riguardo il mapping dell'ereditarietà
    - Ogni classe diventa una tabella a sè, le proprietà sono ripetute come attributi in ogni tabella, se indicate come persistenti
  - Table per concrete class
    - Polimorfismo ed ereditarietà non hanno nessun mapping nello schema SQL
  - Table per class hierarchy
    - Si usa uno schema SQL denormalizzato, che "riassume" tutta la gerarchia di classi in un'unica tabella
    - Una colonna funge da "discriminator" → quale delle classi di una gerarchia mappa la tupla?
  - Table per subclass
    - La relazione di ereditarietà *is-a* viene mappata come un *has-a*
    - La classe principale dispone di una tabella, la quale riferisce, per mezzo di relazioni di chiave esterna, altre tabelle che contengono gli attributi specifici delle sottoclassi

Gerarchie di classi

- La modalità di mapping si specifica con l'annotazione @Inheritance
  - È sempre necessario l'id nella superclasse
- InheritanceType.TABLE_PER_CLASS → "table per concrete class"
- InheritanceType.SINGLE_TABLE → "table-per-class-hierarchy"
  - @DiscriminatorColumn indica l'attributo "discriminator" (nella superclasse)
  - @DiscriminatorValue va usato, nelle sottoclassi, per attribuire un valore al discriminator che funga da contrassegno
  - Hibernate supporta anche la possibilità di desumere il discriminator per mezzo di una formula SQL
    - Utile per il mapping di schemi *legacy*
  - Si noti che, con questo approccio, i campi devono essere tutti *nullable*
- InheritanceType.JOINED → "table per subclass"
  - È anche possibile ottenere il comportamento equivalente solo per una sottoclasse, marcando quest'ultima con @SecondaryTable

Table per concrete class con mapping del polimorfismo

•Non si asserisce nulla riguardo il mapping dell'ereditarietà
•Ogni classe diventa una tabella a sè, le proprietà sono ripetute come attributi in ogni tabella, se indicate come persistenti



•Problemi: non supporta bene le associazioni polimorfe. Non basta una chiave esterna nella superclasse ma occorre in tutte le sottoclassi.
•Anche le query polimorfe sono un problema: richiede più queries sulle sottoclassi

```
@MappedSuperclass
public abstract class BillingDetails implements Serializable, Comparable {
    @Column(name = "OWNER", nullable = false)
    private String owner;
.....
@Entity
@AttributeOverride(name = "owner", column =
@Column(name = "CC_OWNER", nullable = false)
)
public class CreditCard extends BillingDetails {
@Id @GeneratedValue
@Column(name = "CREDIT_CARD_ID")
private Long id = null;
@Column(name = "NUMBER", nullable = false)
private String number;
```

Table per concrete class con unione

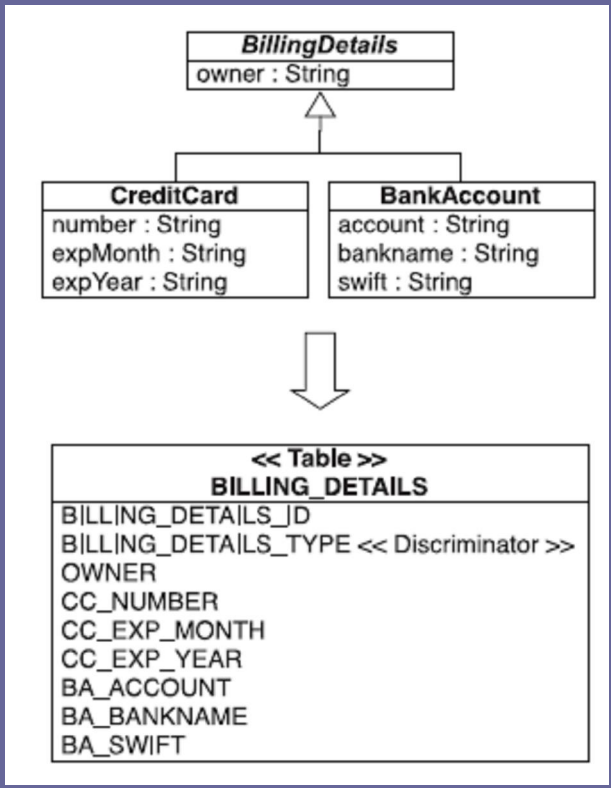
- Condivisione delle proprietà della superclasse, no duplicazione
- Associazioni possibili a livello di superclasse
- Hibernate risolve le query polimorfe usando Union

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class BillingDetails {
@Id @GeneratedValue
private Long id = null;
@Column(name = "OWNER", nullable = false)
private String owner;
...
}

@Entity
@Table(name = "CREDIT_CARD")
public class CreditCard extends BillingDetails {
@Column(name = "NUMBER", nullable = false)
private String number;
...
```

Table per class hierachy

•Si usa uno schema SQL denormalizzato, che “riassume” tutta la gerarchia di classi in un’unica tabella
•Una colonna funge da “discriminator” → quale delle classi di una gerarchia mappa la tupla?



Strategia semplice e performante

Ottima rappresentazione del polimorfismo

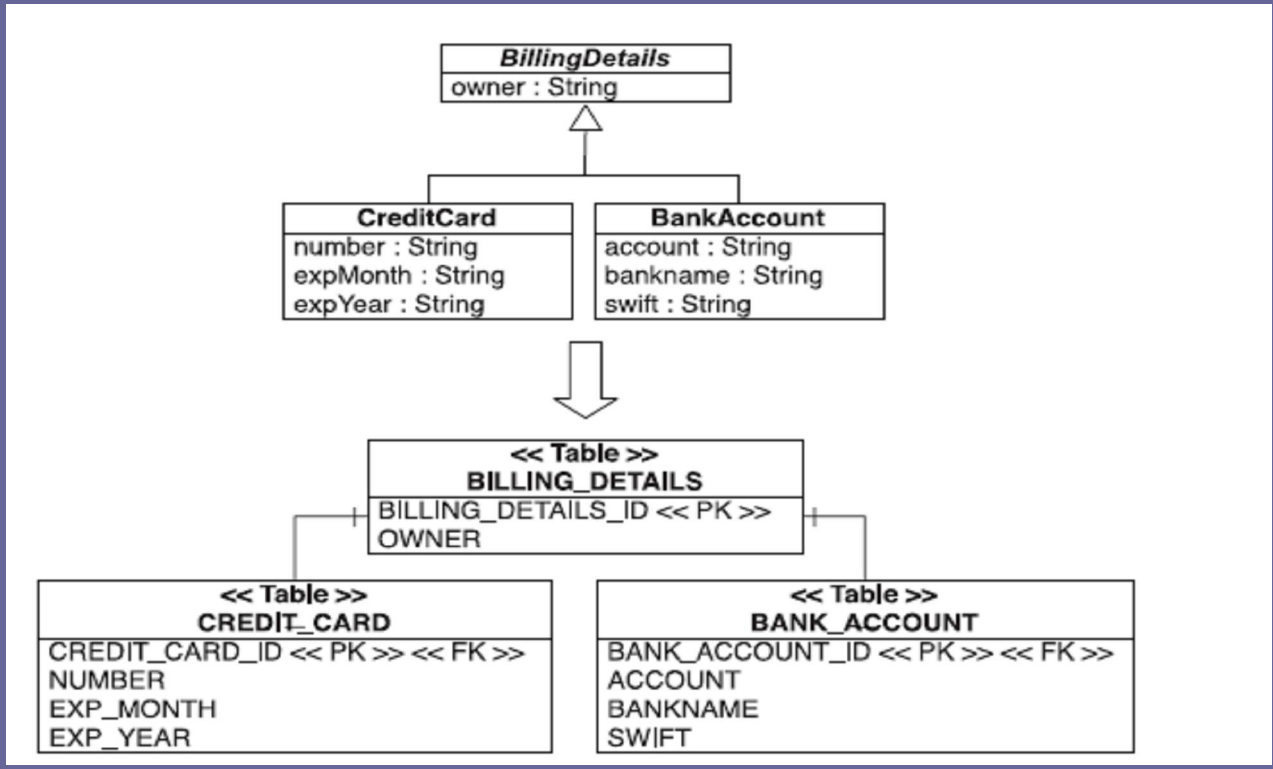
Problemi: colonne devono essere nullable, denormalizzazione dello schema

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
name = "BILLING_DETAILS_TYPE",
discriminatorType = DiscriminatorType.STRING)
public abstract class BillingDetails {
@Id @GeneratedValue
@Column(name = "BILLING_DETAILS_ID")
private Long id = null;
@Column(name = "OWNER", nullable = false)
private String owner;
...}

@Entity
@DiscriminatorValue("CC")
public class CreditCard extends BillingDetails {
@Column(name = "CC_NUMBER")
private String number;
...
}
```

Table per subclass

La relazione di ereditarietà *is-a* viene mappata come un *has-a*
La classe principale dispone di una tabella, la quale riferisce, per mezzo di relazioni di chiave esterna, altre tabelle che contengono gli attributi specifici delle sottoclassi



Ogni classe/sottoclasse (anche astratta) che ha proprietà persistenti ha la sua tabella. (Non c'è duplicazione nelle sottoclassi)

Ogni proprietà di una sottoclasse è memorizzta insieme ad una primary key che è anche chiave esterna nella tabella della superclasse

- Vantaggi nella normalizzazione dei dati

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetails {
@Id @GeneratedValue
@Column(name = "BILLING_DETAILS_ID")
private Long id = null;
...}
```

```
@Entity
public class BankAccount {
...
}
```

In subclasses, you don’t need to specify the join column if the primary key column of the subclass table has (or is supposed to have) the same name as the primary key column of the superclass table

BankAccount has no identifier property; it automatically inherits the BILLING_

- DETAILS_ID property and column from the superclass, and Hibernate knows how to join the tables together if you want to retrieve instances of BankAccount. Of
- course, you can specify the column name explicitly:
- @Entity
- @PrimaryKeyJoinColumn(name = “Bank_account_ID")

Mapping delle relazioni

- *“Managed relationships”*
  - *Hibernate* non gestisce, lato Java, le associazioni permanenti
  - In caso di problemi di comprensione delle associazioni con Hibernate, chiediamoci:
    - *Cosa farei se non stessi usando Hibernate?*
  - Infatti, Hibernate non cambia la semantica di Java che già conosciamo

Mapping delle relazioni

- Collections
  - Set -> HashSet
  - SortedSet -> TreeSet
  - List -> ArrayList
  - Collection (Bag) -> ArrayList
  - Map -> HashMap
  - SortedMap -> TreeMap

Mapping delle relazioni

- Relazione monodirezionale
- Si usa @ManyToOne(targetEntity=…class)
  - Marca una proprietà come foreign key verso una entità
    - La classe dell'entità obiettivo può essere omessa
    - In tal caso, fa fede il tipo della proprietà desunto tramite reflection
  - @JoinColumn(name="…", nullable=false/true)
    - Può essere usato per specificare che colonna generare per la foreign key

Mapping delle relazioni

- Per rendere la relazione bidirezionale, è sufficiente:
  - Inserire una proprietà collection con opportuna semantica nell'entità obiettivo
  - Marcarla con @OneToMany(mappedBy="field")
    - Il nome del campo deve corrispondere a quello usato nella classe di partenza

Mapping delle relazioni

- Persistenza transitiva:
  - La richiesta di salvare un oggetto sul DB può far partire, a catena, altre INSERT/UPDATE sugli oggetti collegati
    - Usare cascade = {CascadeType.PERSIST, CascadeType.MERGE}
  - Per propagare la richiesta di eliminazione a tutti gli oggetti dipendenti, si può usare cascade = {…, CascadeType.REMOVE, …}
    - È necessario assicurarsi di essere gli unici proprietari dell'oggetto
    - Oggetti condivisi dovranno essere necessariamente eliminati ad uno ad uno, con una sequenza di operazioni in Java