**Java**

**Annotation**

@Annotation

Esempio

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

@Repeatable(ToDoContainer.class)

public @interface Todo {

String dueDate();

String task() default "";}

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

@interface ToDoContainer { Todo[] value(); }

**Wildcard**

G<? extends tipo> accetta i tipi che estendono la classe tipo (utile per gli input)

G<? super tipo> accetta i tipi che vanno da Object fino a tipo non oltre (utile per gli output)

Stream metodi:

* **Min, max, filter**: trova massimo, trova minimo, filtra in base alla funzione specificata come parametro
* **Map, collect**: map crea una collection, di tipo map, e converte elemento per elemento nel tipo desiderato, lamda function
* **FlatMap**: rimpiazza ogni valore di uno stream, con una stream indipendente e infine crea un stream con i risultati. Esempio (estrarre i caratteri da un testo, converto tutte le stringhe in una stream di caratteri e con la flatMap crea una lista(stream) con tutti i caratteri di ogni parola. **Codice**: *words.stream().flatMap(word -> word.chars().mapToObj(c -> (char) c))*
* **AllMatch**: controlla se il predicato (lambda) è vero per tutti gli elementi della stream
* **anyMatch**: controlla se il predicato (lambda) è vero per nessuno gli elementi della stream
* **noneMatch**: controlla se non ci sono elementi he fanno match nel predicato.
* Reduce: consente di definire un’operazione di aggregazione (nella lambda) dei dati in modo da restituire un singolo risultato esempio: sommare gli elementi.
* **mapToInt**, **mapToLong**,…: mappano gli elementi con nel tipo primitivo
* **Collectors**.**summarizingDouble**(campo): crea statistiche sulla stream basate sul campo/elemento da analizzare
* **Collectors.groupingBy**(elemento da raggruppare, in base a cosa voglio raggruppare (Collectors.counting(),…)): raggruppa gli elementi in base alla lambda/parametri passate/i, nel primo parametro dire per cosa si vuole raggruppare, esempio valore, primo carattere ecc., se non specifico nulla nel secondo parametro, restituisce un set avente chiave l’elemento scelto da comparare e il valore è la lista di tutti gli elementi che hanno tale valore.
* **Collectors.partitioningBy():** crea una map(Boolean, valore) Boolean vale true o false in base alla condizione specificata in PartitioningBy()

**Optional**:

è un oggetto che contiene un qualsiasi oggetto. È di tipo generics dichiarazione: *Optional<T> tipo = Optional.empty();*//*optional* *vuoto* Optional.of(variabile) inserisce il valore di variabile nell’oggetto, se null lancia NullPointException.

* **Optional.ofNullable(var)** : S e var è null non resituisce un’accezione, ma crea un optional vuoto
* **isPresent()/isEmpty()**[da java 11]: restituisce true se l’optional è vuoto, false altrimenti. Esiste una versione nel quale è possibile specificare la condizione di esistenza.
* **get():** restituisce il valore contenuto se non è null, lancia *NoSuchElementException*
* **orElse():**permette di evitare le eccezioni, se il valore è null viene effettuata l’operazione definita (lambda), anche se il valore non è nullo restituisce il valore di defualt. **(da non usare)**
* **orElseGet():**come orElse(), restituisce un valore di default, solo se il dato è null. **(da non usare)**
* **orElseThrow():** se get restituisce null Lancia un eccezione.
* **ifPresentOrElse(p1, p2):** consente di specificare due predicati in base alla presenza o meno di un elemento, p1 caso in cui esista p2 caso null.**(da usare)**

**JBDC**

È un API Java standard che consente l’accesso a tipi tabulati (database). Esso include il supporto per ognuna delle seguenti funzionalità:

* Connessione al database
* Invio di query e aggiornare lo stato del database
* Recuperare e ricevere i risultati dal database

JBDC è: un API standard di Java, il client è portable e facile da configurare, il codice è indipendente da database su cui si opera, il database è identificato da un URL che usa l’interfaccia di connessione. Sono presenti anche dei DataSource che consentono la gestione di transazioni e le connessioni pooling.

URL di connessione: ***jdbc:[driver]:[database]*** (jbdc=protocollo, driver= tipo di db(mysal,h2,ssql,oracle…, database:nome db)

Esempio: jdbc:h2:~/test

**Creare una connessione**: Connection connection = DriverManager.getConnection(connectionURL, username, password);

**Creare uno statement**: Statement statement = connection.createStatement();

**Eseguire una query che ritorna un risultato:** ResultSet rs = statement.executeQuery(SQLQueryString);

**Statement**: fornisce metodi che consentono l’aggiornamento:

* executeQuery(sqlQuery) -> restituisce un ResultSet con i dati risultanti dalla query
* executeUpdate(sqlUpdateQuery)-> esegue query di aggiornamento (INSERT, UPDATE, DELETE, CREATE/DROP TABLE…)
* execute(sqlQuery)->può restituire più ResultSet o numero di aggiornamenti

**ResultSet**: è un’interfaccia che fornisce i metodi per l’ispezione dei risultati di una query, metodi:

* getXYZ(String nomeColonna): XYZ è il tipo della colonna, consente di ottenere i risultati di una determinata colonna di tipo XYZ, esempio:getString(“nome”)
* getXYZ(int indiceColonna): XYZ è il tipo, consente di prendere la colonna di tipo XYZ che è nella posizione richiesta,

**gli indici partono da 1**

* next(): muove il cursore al risultato successivo (riga)
* getMetaData(): restituisce i metadata della tabella (nome tabella, numero colonne, …)

gli Statement e i ResultSet DEVONO essere chiusi una volta terminato l’utilizzo con . close()

La connessione si crea una sola volta (oggetto globale), non sono thread safe, sarebbe meglio usare connessioni pool.

Gli statment è meglio non riutilizzarlo, crea ad ogni operazione.

Gli statment e i DataResult sono supportati dal try-with-resources, in modo da lasciar fare le operazioni di chiusura a java in automatico

Tutti le operazioni di query possono lanciare SQLException, quindi va sempre messo tutto in un try-catch/try-with-resources

**PreparedStament**: consente di creare statement (PreparedStatement estende statement) più rapidi e sicuri. Precompila in fase di creazione anche se i dati vengono modificati successivamente. Metodi:

* connection.preparedStatement(query): crea un prepared statement, se vi sono dei dati di input usare ? , essi saranno numerati

esempio: PreparedStatement statement = connection.prepareStatement("SELECT \* FROM MOVIES where TITLE = ?");

* statement.setXYZ(int indice, XYZ dato): setta il parametro ? alla posizione dell’indice e mette dato che è di tipo XYZ,

esempio: statement.setString(1, "Avengers");

I DataSource sono usati per software di tipo Enterprise. Sono una alternativa al DriverMenager, supportano connessioni pooling e transactions distribuite.

**Hibernate**

Hibernate è un implementazione di un sistema ORM (Object Relational Mapping) che consente la traduzione di un applicazione OOP in database (la traduzione non è semplice come operazione).

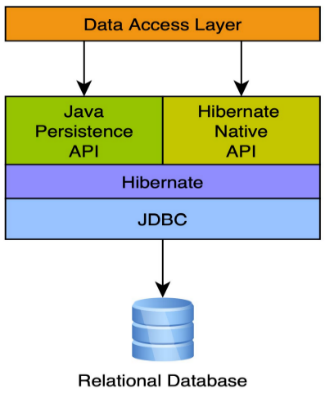
In un sistema ORM si possono distinguere 3 elementi:

* Le Classi sono le entità, quindi le tabelle del database
* Gli oggetti sono le istanze, quindi le righe della tabella
* Gli attributi della classe e le relazioni sono gli attributi, quindi le colonne

Da Java 5 sono state inserite le JPA (Java Persistence API) che consentono appunto la traduzione delle applicazioni ad oggetti in database relazionali.

Le JPA consentono l’uso dei POJOs (Plain Old Java Objects) che consentono la persistenza senza richiedere l’implementazione nella classe di funzionalità aggiuntive, il mappaggio avviene tramite file XML (persistence.xml) e annotation standard (@entity, @column…).

Le JPA forniscono un EntityMenager API, un Api runtime che gestisce le entità e consente di effettuare query tramite JPQL.

Hibernate internamente sfrutta JBDC.

La funzionalità principale di Hiberante è quella di mappare le classi in tabelle relazionali e mappare i tipi dei dati in tipi SQL. Hiberante fornisce anche funzionalità che facilitano la lettura dei risultati del query.

Regole per creare un DB con Hiberante:

* Le classi non devono avere parametri nel costruttore
* Usare un identificativo numerico come chiave primaria, o comunque va messo un attributo che identifica in modo univoco l’istanza.
* Le liste, i set e i generics sono supportati

Per l’uso di hibernate si può usare maven per gestire le dipendenze, nel pom.xml si specificano le dipendenze.

In src>main>java>resources>META-INF>persistence.xml si specificano i dati relative alle entità e al database

Peristence.xml è il persistence file. La persistence Unit è la classe nel quale è presente l’EntityMenager.

<class>NomeClasseEntità</class> tutte le entità vanno mappate in questo modo nel file xml.

Annotation di Hiberante:

* @Entity: va fuori dalla classe e indica che quella classe è un entità
* @Id

@GeneratedValue (generator = "increment")

@GenericGenerator (name = "increment", strategy = "increment"): va posto sopra un campo numerico che sarà la chiave primaria id

* @Column: va posto sopra tutti i campi che si vuole mappare come attributo
* @Table(name=”…”): va fuori dalla classe, consente di dare un nome diverso da quello della classe alla tabella
* @Transient: il dato non va mappato
* @OrderBy(“campo asc/desc”): consente l’ordinamento degli elementi di una list in base al parametro e al criterio

**EntityMenagerFactory**

Consente la creazione di EntityMenager che è l’elemento principale per gestire gli oggetti mappati. Creazione EntityMenager:

EntityManagerFactory factory =Persistence.createEntityManagerFactory("persistenceUnitClassName");//crea un EMF

EntityManager entityManager = factory.createEntityManager(); //crea l’EM

Le operazioni di inserimento, aggiornamento, query vanno fatte tramite le transaction. Esempio di transaction di inserimento dati:

public class Example {

public static void main(String[] args) {

EntityManagerFactory factory = Persistence.createEntityManagerFactory("ch.supsi.persistence.example");

EntityManager entityManager = factory.createEntityManager();

entityManager.getTransaction().begin();

Author author = new Author();//crea istanza

author.setFirstName("Bruce");//inserisci I dati

author.setLastName("Wayne");

author.setBirthYear(1900);

entityManager.persist(author);//inserisci i dati, ma non in modo definitivo, **persist si usa solo per le nuove entità**

// ...

entityManager.getTransaction().commit();//conferma l’inserimento con commit, con roolback si annulla l’inserimento

entityManager.close();

}

}

Se una transation fallisce, va rieseguita nuovamente. Se viene completata con successo sarebbe meglio chiudere l’EntityMenager e ricrearne uno nuovo in modo da evitare artefatti di caching per esempio.

Ricerca per id si usa find(nome classe, id da cercare) ritornerà un oggetto del tipo del nome classe specificato Esempio:

Author author = entityManager.find(Author.class, 1);

**Filed contro Property**

* Field-based-access: la JPA usa la relection per recuperare le informazioni sui campi e si aspetta che ci siano anche le annotation nella dichiarazione dei campi
* Property-based-access: la JPA sfrutta le annotation dei metodi getter e setter, per accedere al campo userà i metodi getter e setter

Field annotation è la soluzione migliore per diverse ragioni (lettura codice, flessibilità, omissione dei metodi getter e setter,…)

**Le associazioni in Hibernate**

In Hibernate come i database esistono 4 tipi associazione:

* One-to-One
* Many-to-One
* One-to-Many
* Many-to-Many

**One-to-One**

Si usa per le associazioni 1:1 tra due tabelle si usa l’annotation @OneToOne e @JoinColumn(name=”nome fk” referencedColumnName=”id”)

Queste due annotation vanno poste nel campo che fa riferimento ad un'altra tabella, referenceColumnName contiene il nome del campo chiave primaria.

Esistono relazioni unidirezionali e bidirezionali, quest’ultime sono relazioni nelle quali non mettiamo il JoinColumn, ma mettiamo a OneToOne l’attributo mappedBy, questo avviene nell’altra entità

Nelle Bidirezionali 1-1 abbiamo quindi:

T1: @OneToOne e @JoinColumn(name=”nome fk” referencedColumnName=”id”) e in

T2: @OneToOne(mappedby=”nome campo di riferimento”),

Le bidirezionali consentono di raggiungere l’elemento di partenza e quello di arrivo da entrambe le direzioni A->B =B->A nelle unidirezionali invece è possibile solo fare A->B, si prediligono le bidirezionali, sono più semplici e agevoli

**One-to-Many**

* Lato N: @OneToMany(mappedBy=”nomeCampoCheRelazionaLato1”)
* Lato 1: @ManyToOne; JoinColumn(name=”nomeFk”):

Crea una tabella di mezzo

È simile al One-To-Many, ma non usa la tabella di mezzo

**Many-to-Many**

Viene creata una tabella di mezzo; configurazione relazione esempio, le due parti sono solitamente delle list(il lato è indifferente)

@ManyToMany

@JoinTable(name = "books\_authors",

joinColumns = @JoinColumn(name = "author\_id"),

inverseJoinColumns = @JoinColumn(name = "book\_id"))

private Set<Book> books;

nell’altro lato della relazione va messo solo @ManyToMany(mappedBy=”books”)

**Fare query**

Esiste un oggetto Query e tramite EntityMenagerObj.createQuery(JPQL query) si crea quet’ultimo. Vi sono metodi come getResultList(), getSingleResult() che restituiscono i risultato(aiuto IDE). È possibile creare delle namedquery, quindi query già pronte richiamabili con il solo nome attraverso l’annotation @NamedQueries({NamedQuery(name=”…", query=”…),….}), si possono usare parametri placeholder scrivendo :nome e poi aggiungere il parametro in fase di richiesta. Esempi per richiamare una named query

List<Book> books1 = entityManager.createNamedQuery("getAllBooks").getResultList();

List<Book> books2 = entityManager.createNamedQuery("getBookByTitle").setParameter("title", "Super").getResultList();

JPQL è simile a SQL, la logica però di unione dati ad esempio ci pensa hibernate a prendere i dati esterni, il join serve ad avere dati di più oggetti(casi particolari non dovrebbero presentarsi) esempio:

SELECT a FROM AUTHOR a; oppure SELECT b FROM BOOK b WHERE b.title=:titleName; **bisogna sempre definire la variabile (a,b,..)**