**Java EE Guide**

Francesco Abate

Sommario

[Capitolo 1: Introduzione a Java EE 7 4](#_Toc30756444)

[1.1: I livelli di Java EE 4](#_Toc30756445)

[1.2: Componenti 4](#_Toc30756446)

[1.3: Container 4](#_Toc30756447)

[1.4: Servizi 5](#_Toc30756448)

[1.5: Packaging 6](#_Toc30756449)

[1.6: Annotazioni 6](#_Toc30756450)

[Capitolo 2: Context and Dependency Injection (CDI) 7](#_Toc30756451)

[2.1: Panoramica sui bean 7](#_Toc30756452)

[2.2: Cos’è la Dependency Injection (DI) 7](#_Toc30756453)

[2.3: Ciclo di vita di un CDI bean 7](#_Toc30756454)

[2.4: Scope e contesto 8](#_Toc30756455)

[2.5: Introduzione agli interceptors 8](#_Toc30756456)

[2.6: Introduzione al Deployment Descriptor 8](#_Toc30756457)

[2.7: Introduzione alla pratica sui CDI bean 8](#_Toc30756458)

[2.8: Anatomia di un CDI bean 9](#_Toc30756459)

[2.9: Dependency Injection con i CDI bean 9](#_Toc30756460)

[2.9.1: Iniezione tramite @Inject 9](#_Toc30756461)

[2.9.2: Qualificatori - specificare l’implementazione 11](#_Toc30756462)

[2.9.3: Qualificatori con membri 13](#_Toc30756463)

[2.9.4: Producers - iniezione di tipi primitivi e POJO 14](#_Toc30756464)

[2.10: Contesto (scope) e annotazioni 15](#_Toc30756465)

[2.11: Interceptor 16](#_Toc30756466)

[2.11.1: Interceptor Binding 19](#_Toc30756467)

[2.12: Eventi 19](#_Toc30756468)

[2.13: Esercizio riepilogativo 21](#_Toc30756469)

[2.13.1: Scrivere il POJO Book 22](#_Toc30756470)

[2.13.2: Scrivere NumberGenerator, implementazioni e qualificatori 23](#_Toc30756471)

[2.13.3: Scrivere il gestore dei libri BookService 24](#_Toc30756472)

[2.13.4: Rendere iniettabile la classe Logger 24](#_Toc30756473)

[2.13.5: Creare l’interceptor che utilizza il Logger 25](#_Toc30756474)

[2.13.6: Applicare l’interceptor @LoggableInterceptor 25](#_Toc30756475)

[Capitolo 3: Java Persistence API (JPA) 26](#_Toc30756476)

[3.1: Introduzione alle entità 27](#_Toc30756477)

[3.2: Anatomia di un’entità 27](#_Toc30756478)

[3.3: Riguardo l’Object Relational Mapping (ORM) 28](#_Toc30756479)

[3.4: Introduzione alle query e all’Entity Manager 28](#_Toc30756480)

[3.5: Persistence Unit 30](#_Toc30756481)

[3.6: Ciclo di vita delle entità e le callbacks 31](#_Toc30756482)

[Capitolo 4: Gestione degli Oggetti Persistenti 31](#_Toc30756483)

[4.1: Entity Manager 31](#_Toc30756484)

[4.1.1: Ottenere un Entity Manager 32](#_Toc30756485)

[4.1.2: Entità managed, detached e relazione con l’appartenenza al database 33](#_Toc30756486)

[4.1.3: Persistence Context 33](#_Toc30756487)

[4.1.4: Rendere persistente un’entità tramite persist 34](#_Toc30756488)

[4.1.5: Ricavare un’entità dal database tramite find e getReference 34](#_Toc30756489)

[4.1.6: Rimuovere un’entità dal database tramite remove 35](#_Toc30756490)

[4.1.7: Sincronizzazione con il database tramite flush e refresh 35](#_Toc30756491)

[4.1.8: Contenimento di un’istanza tramite contains 36](#_Toc30756492)

[4.1.9: Rendere detach un’entità 36](#_Toc30756493)

[4.1.10: Aggiornare un’entità nel database 37](#_Toc30756494)

[4.2: Query 37](#_Toc30756495)

[4.3: Ciclo di vita delle entità 37](#_Toc30756496)

[Capitolo 5: Enterprise JavaBeans 37](#_Toc30756497)

[5.1: Introduzione agli EJB 37](#_Toc30756498)

[5.2: Scrivere gli EJB 37](#_Toc30756499)

# Capitolo 1: Introduzione a Java EE 7

Oggigiorno le aziende operano a livello globale, distribuendo i propri sistemi in tutto il mondo e permettendone il funzionamento ogni giorno ad ogni orario. Si cerca, quindi, di offrire un sistema distribuito, portabile, affidabile, sicuro ed efficiente spendendo il meno possibile, garantendo le caratteristiche appena elencate. Java Enterprise Edition cerca di rispondere a queste esigenze.

Java Enterprise Edition (Java EE) è un insieme di tecnologie integrate, con l’obiettivo di ridurre il costo e la complessità di sviluppare e gestire applicazioni basate su un’architettura multi-tier, dove tra i client (front end) e i dati (back end) vengono posti alcuni livelli. Java EE introduce la logica a componenti, quali sono unità di software categorizzabili come componenti client, web e business.

Java EE introduce una serie di nuove funzioni, tra le quali la Context and Dependency Injection (CDI), la quale permette, in poche parole, l’iniezione automatica di risorse. Vengono inoltre introdotti i containers, i quali provvedono determinati servizi ai componenti del sistema, come la gestione del ciclo di vita, dependency injection, concorrenza ed altro.

## 1.1: I livelli di Java EE

Abbiamo già preannunciato l’architettura di Java EE definendola multi-tier. I tier che introduce sono i seguenti:

* un tier di presentazione client (browser web o client applicativo);
* un tier HTTP server (Web server / Web container) che fornisce l’assemblaggio delle informazioni fornite dall’applicazione mediante determinate tecnologie come JSP, Servlet, ecc.;
* un tier server di logica di business (server applicazione / EJB container);
* un ultimo tier per l’accesso ai dati (Database container).

I tier servono vengono accomunati come unico tier, detto anche middle-layer o middle-tier.

## 1.2: Componenti

Java EE definisce quattro tipologie:

* Applets, sono applicazioni con GUI che vengono eseguite nel web browser;
* Applicazioni, sono programmi eseguiti su un client che sfruttano principalmente il middle-tier;
* Applicazioni Web, sono programmi fatti di servlet, filtri, listener, pagine JSP, ecc.; tali programmi vengono eseguiti in un web container e rispondono ad http requests da parte dei web client;
* Applicazioni Enterprise, sono programmi fatti di EJB (Enterprise Java Beans), Java Message Service, Java Transaction, ecc.; tali programmi vengono eseguiti in un EJB container e possono essere accessi sia localmente che in remoto tramite RMI.

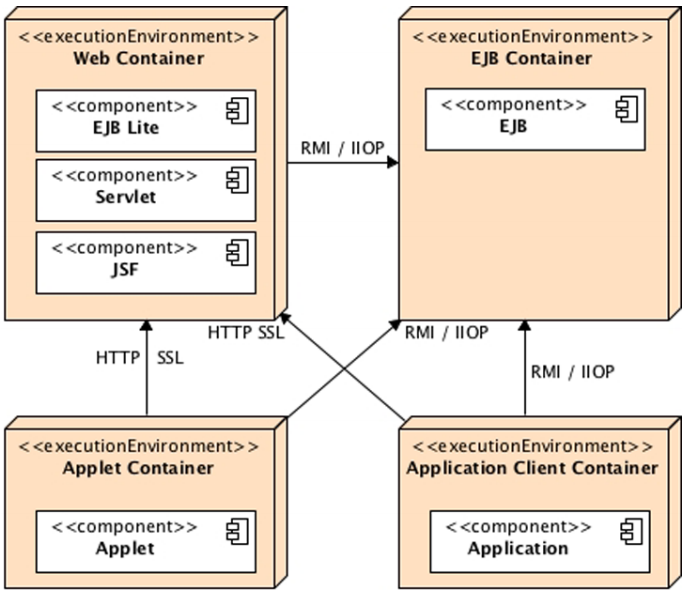
## 1.3: Container

L’infrastruttura di Java EE è partizionata in diversi domini denominati come container, i quali offrono servizi alle componenti supportate (sicurezza, accesso al database, transazioni, servizio di naming, resource injection) incrementando la modularità che il sistema offre. Ogni tipologia di container gestisce determinate tipologie di componenti.

Java EE definisce quattro tipi di container:

* Applet container, fornisce un ambiente in cui vengono delineati i limiti nel quale deve essere eseguito il codice;
* Application container, include molteplici classi Java, librerie ed altri file per gestire la sicurezza, il servizio di naming, ecc. Tale container comunica con il container EJB utilizzando RMI-IIOP, mentre con il web container utilizzando HTTP;
* Web container, adatto al managing e all’esecuzione di componenti web come servlets, JSPs, filtri, listeners, ecc. Tale container è responsabile riguardo l’istanziazione, l’inizializzazione e l’invocazione delle servlets, utilizzando HTTP;
* EJB container, adatto al managing e all’esecuzione degli Enterprise beans: crea nuove istanze di determinati EJBs, ne gestisce il loro ciclo di vita, provvede servizi e la possibilità di essere invocati asincronamente. Addirittura, in tale container non è il programmatore ad utilizzare la keyword new, siccome ciò viene effettuato dal container.

La seguente immagine mostra i diversi container che Java EE introduce: nei container vengono mostrati i componenti supportati, mentre le frecce che collegano i container mostrano i protocolli che i container utilizzano per comunicare tra loro.

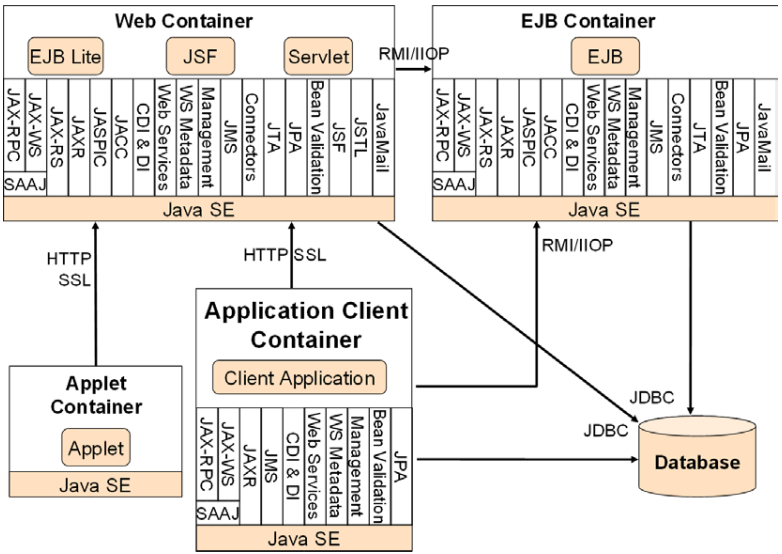


## 1.4: Servizi

I container provvedono determinati servizi alle componenti gestite, permettendo al programmatore di concentrarsi sul codice di business invece di risolvere problemi tecnici rilegati alle applicazioni enterprise. Vediamo ora i servizi offerti, in generale, dai container: ovviamente ogni container offre determinati servizi, quindi il fatto che vediamo tutti i servizi non implica il fatto che ogni container offra ogni servizio.

* JTA – Java Transaction API: questo servizio permette l’utilizzo delle transazioni, cioè serie di istruzioni le quali vengono eseguite tutte o nessuna;
* JPA – Java Persistence API: questo servizio permette di rendere persistenti determinati dati e permette l’esecuzione delle query tramite Java Persistente Query Language (JPQL), in modo da gestire oggetti posti in un database;
* JMS – Java Message Service: questo servizio permette alle componenti di comunicare tra loro asincronamente tramite messaggi;
* Naming: questo servizio permette di associare e trovare oggetti tramite operazione di lookup. Tale servizio viene utilizzato anche tramite iniezione, ma in maniera trasparente;
* JavaMail: questo servizio permette all’applicazione di mandare email;
* XML processing: questo servizio effettua la manipolazione di codice XML;
* JSON processing: questo servizio effettua la manipolazione di codice JSON;
* Servizi di sicurezza;
* Web services;
* Dependency Injection (DI): alcune risorse possono essere iniettate in determinati componenti (managed components);
* Deployment.

Nella seguente immagine vengono mostrati i servizi categorizzati in base ai container.



## 1.5: Packaging

Per esser deployati in un container, i componenti devono essere impacchettati in determinati archivi, il cui formato dipenderà dal container che gestisce quel determinato tipo di componente.

## 1.6: Annotazioni

È bene sapere che esistono due tipi di approccio alla programmazione: l’approccio imperativo e l’approccio dichiarativo: l’imperativo tende a specificare l’algoritmo (cosa si usa per risolvere) che risolve un determinato problema, mentre il dichiarativo tende a specificare come (come si risolve) deve essere risolto un determinato problema. In Java EE, l’approccio dichiarativo utilizza metadati, annotazioni e deployment descriptors. Vedremo in particolare le annotazioni, le quali seguono tale formato: @annotation.

# Capitolo 2: Context and Dependency Injection (CDI)

La prima versione di Java EE introduceva il concetto di inversion of control (IoC), il quale consisteva nel fatto che il container avesse predo controllo del codice di business e che avesse provvisto diversi servizi tecnici, come transazioni o gestione della sicurezza. Il fatto che il container prenda controllo sta nel fatto che gestisca egli stesso il ciclo di vita delle componenti, che gestisca la dependency injection e la configurazione delle componenti. In Java EE 6 è stata introdotta la Context and Dependency Injection (CDI), la quale permette ai managed bean (bean gestiti dal container) di essere iniettabili, intercettabili e, appunto, gestibili. CDI è stata creata con lo scopo di assicurare basso accoppiamento e forte tipizzazione (loose coupling, strong typing). In questo capitolo approfondiremo la dependency injection (DI), lo scoping, il basso accoppiamento e concetti che hanno a che fare con CDI.

## 2.1: Panoramica sui bean

Mentre Java SE possiede solamente i Java Bean, Java EE possiede gli Enterprise Java Bean (EJB). Gli EJB non sono gli unici componenti offerti da Java EE, difatti si hanno altri componenti come le servlet, web services, entità e managed bean. È bene sapere che i normali Java Bean offerti da Java SE sono detti POJO in Java EE (Plain Old Java Object) e seguono un determinato pattern che consiste in convenzioni per proprietà e costruttori. Anche altri componenti di Java EE seguono precisi pattern, ma con la differenza che vengono eseguiti in un container e usufruiscono di determinati servizi: tali componenti vengono detti managed bean.

I managed bean sono bean gestiti dal container che supportano solo determinati servizi, quali sono la resource injection, la gestione del ciclo di vita e l’intercezione. Tali bean possono essere visti come una generalizzazione delle diverse componenti di Java EE, ad esempio un EJB può esser visto come un managed bean con servizi aggiuntivi, come una servlet può esser vista come un managed bean con altri servizi aggiuntivi.

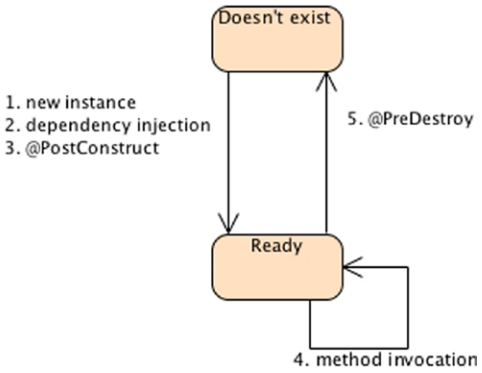
I CDI bean sono una categoria di managed bean con un ciclo di vita migliorato per oggetti con uno stato (stateful objects); sono legati ad un contesto ben definito; permettono la DI ben tipizzata, l’intercezione e l’uso delle annotazioni. Precisamente, tranne in alcune eccezioni, ogni classe Java con un costruttore di default ed eseguita in un container viene considerata come bean.

## 2.2: Cos’è la Dependency Injection (DI)

La Dependency Injection (DI) è un design pattern che permette di disaccoppiare componenti dipendenti tra loro. Si utilizza la DI in un ambiente gestito, quindi invece di cercare oggetti tramite servizi di lookup, il container inietta gli oggetti dipendenti in maniera del tutto automatica. Sostanzialmente, quindi, il container effettua in automatico il lookup dell’oggetto per poi iniettarlo. È bene sapere che è possibile utilizzare JNDI per cercare ed ottenere risorse gestite (vedremo JNDI più avanti, ma sostanzialmente è un servizio che permette di effettuare il lookup di oggetti), ma il container semplifica ciò nascondendo tali operazioni al programmatore. Ciò ha permetto ai programmatori di iniettare determinate risorse in determinati componenti.

## 2.3: Ciclo di vita di un CDI bean

Il ciclo di vita di un normalissimo POJO è semplicissimo: si crea l’istanza utilizzando la keyword “new” ed essa verrà eliminata quando il garbage collector lo riterrà giusto, in modo da liberare memoria. Per i CDI bean, la questione è decisamente diversa: se si vuole usare un CDI bean in un container, l’uso della keyword “new” è vietato. I CDI bean devono essere necessariamente iniettati, quindi il ciclo di vita di tali bean passerà nelle mani del container. A questo punto sorge spontanea una domanda: come si inizializza un bean se non si richiama il suo costruttore? Ebbene, il container fornisce delle funzioni chiamate “callback”, nelle quali è possibile effettuare determinate operazioni in determinati momenti. Riguardo il ciclo di vita di un CDI bean o di un managed bean, vengono eseguite due callback: @PostConstruct e @PreDestroy.

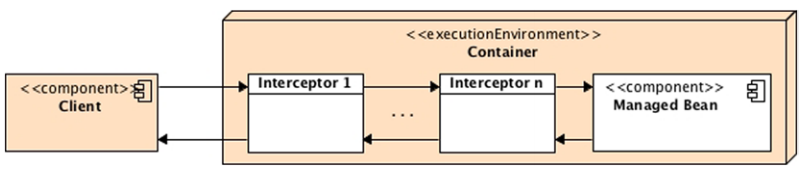


## 2.4: Scope e contesto

I CDI bean sono contestuali, quindi hanno uno scope ben definito: sostanzialmente, il bean opera entro determinati limiti quali possono essere una request, una sessione, l’intera applicazione, ecc. Il container gestisce tutti i bean posti nel proprio scope e, alla fine della sessione, li distrugge tutti.

## 2.5: Introduzione agli interceptors

Gli interceptors vengono utilizzati per interporre invocazioni a metodi di business: ciò permette di riutilizzare con estrema facilità eventuale codice ripetuto, magari, molteplici volte. Sostanzialmente, quindi, è possibile intercettare l’esecuzione di un managed bean per eseguire un interceptor.



## 2.6: Introduzione al Deployment Descriptor

La specifica di Java EE ha un deployment descriptor in XML opzionale: descrive come un componente, un modulo o un’applicazione dovrebbe essere configurata. Con CDI, il deployment descriptor viene chiamato beans.xml ed è obbligatorio il suo utilizzo: tale file non fa altro che attivare CDI.

## 2.7: Introduzione alla pratica sui CDI bean

Un CDI bean non è altro che una classe Java contenente logica di business. Come già detto, i CDI bean non possono essere istanziati tramite keyword “new”, quindi è necessario utilizzare l’iniezione. L’iniezione viene eseguita tramite apposita annotazione @Inject, ma vedremo meglio, tra poco, tutto ciò che è eseguibile su tali bean.

Partiamo, però, con l’anatomia di un CDI bean, in modo da sapere com’è fatto in precisione, per poi passare al come funziona l’iniezione.

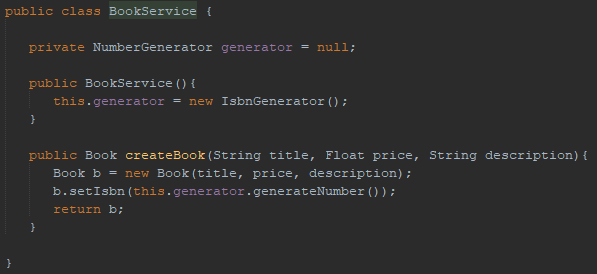
## 2.8: Anatomia di un CDI bean

Per essere trattata come CDI bean, una classe Java deve soddisfare le seguenti condizioni:

* Non deve essere una classe interna;
* Deve essere una classe concreta o annotata con @Decorator;
* Deve avere un costruttore di default senza alcun parametro (non deve essere per forza l’unico costruttore della classe).

## 2.9: Dependency Injection con i CDI bean

Vediamo come funziona la Dependency Injection nella pratica con i CDI bean. Poniamo di avere una classe BookService, la quale fa uso di un generatore di codici ISBN.



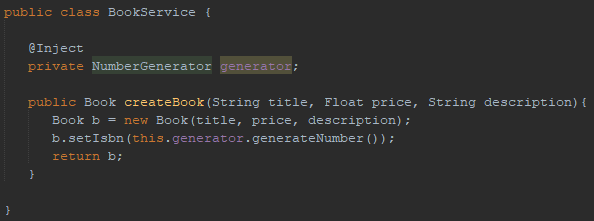
Come possiamo ben vedere, la classe BookService fa utilizzo di un generatore IsbnGenerator: in tal modo, l’accoppiamento è decisamente alto siccome la classe BookService dipende dall’utilizzo di IsbnGenerator. Ciò è un problema, siccome un eventuale cambiamento della classe IsbnGenerator dovrebbe essere riportato anche in BookService. Il problema si capirebbe meglio nel caso si voglia utilizzare un generatore di ISSN invece del generatore di ISBN: si dovrebbe cambiare la classe utilizzata in BookService a causa della dipendenza dovuta dall’alto accoppiamento. Una soluzione riguarda il passaggio di un generatore al costruttore che implementi un’interfaccia comune: se il generatore di ISBN ed il generatore di ISSN implementano entrambi NumberGenerator, il problema è risolto ma c’è comunque dipendenza a causa dell’interfaccia. A questo punto di ricorre all’iniezione.

### 2.9.1: Iniezione tramite @Inject

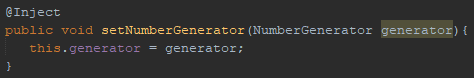
In un ambiente gestito non è necessario costruire dipendenze, ma è possibile lasciare tale compito al container evitando l’alto accoppiamento. In poche parole, la dependency injection è l’abilità di iniettare bean in un modo che assicuri forte tipizzazione. Con CDI è possibile iniettare qualsiasi risorsa tramite l’annotazione @Inject.

Nel seguente esempio, mostrato nella successiva pagina, il container inietta un oggetto che implementi NumberGenerator: la variabile iniettata viene denominata come injection point. L’annotazione @Inject, quindi, definisce un injection point che viene iniettato durante l’istanziazione del bean.

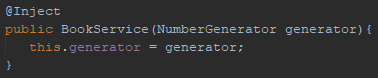
L’iniezione può avvenire in tre modi differenti: tramite proprietà, tramite metodo setter e tramite costruttore. Nell’esempio utilizzante la @Inject viene mostrata l’iniezione tramite proprietà, quindi nei successivi due esempi vengono mostrati in ordine l’iniezione tramite setter e l’iniezione tramite costruttore.



L’iniezione tramite metodo setter funziona nel seguente modo: viene annotato il metodo setter e viene iniettato il parametro passato come argomento. Sostanzialmente, quindi, il container provvederà all’iniezione di un NumberGenerator al metodo setter.



L’iniezione tramite costruttore, invece, permette di iniettare un bean come parametro del costruttore.

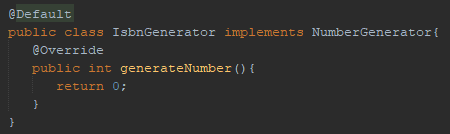


Nei modi appena visti, viene iniettato un oggetto che rappresenta un’implementazione di NumberGenerator: precisamente, però, quale delle tante? Ebbene, è possibile iniettare solo se presente un’unica implementazione dell’interfaccia. L’iniezione, in tal caso, viene anche detta default injection (iniezione di default).

È bene precisare, prima di continuare, il fatto che si possono avere più implementazioni della stessa interfaccia: cambierà il modo con cui viene effettuata l’iniezione (lo vedremo tra poco), ma tra le molteplici implementazioni è possibile specificarne una di default.

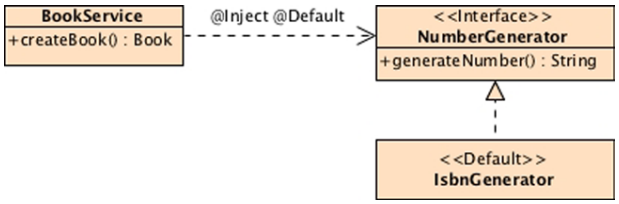
Ricapitolando: se esiste un’unica implementazione dell’interfaccia, allora quell’unica implementazione viene detta di default; se esistono molteplici implementazioni dell’interfaccia, una di queste può essere specificata come default. Ovviamente, nel caso si voglia iniettare un’implementazione di default, è necessario specificarlo tramite apposita annotazione. Mostriamo, di seguito, la specifica dell’implementazione di default e come viene effettuata l’iniezione di default.

Specifica dell’implementazione di default:



Iniezione dell’implementazione di default:



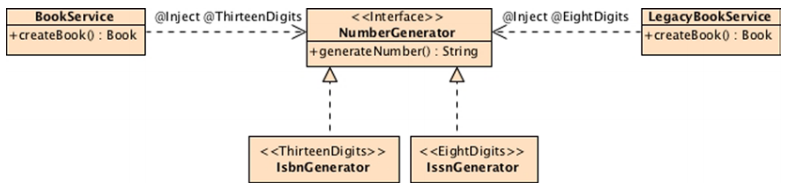


### 2.9.2: Qualificatori - specificare l’implementazione

Il controllo della presenza di un’implementazione iniettabile viene fatto a tempo di inizializzazione, quindi viene controllato ogni injection point esistente. Ciò significa che nel caso non esista alcuna implementazione dell’interfaccia, il container ci informerà riguardo la mancanza di essa. Nel caso, invece, esista un’unica implementazione, sarà possibile effettuare l’iniezione tramite l’annotazione @Default.

Discutiamo, ora, nel caso esistano molteplici implementazioni della stessa interfaccia: il container ci informerebbe riguardo la dipendenza ambigua siccome non saprebbe quale implementazione dell’interfaccia scegliere. È possibile specificare quale implementazione si preferisce utilizzare tramite i qualificatori, i quali precisano l’iniezione effettuata.

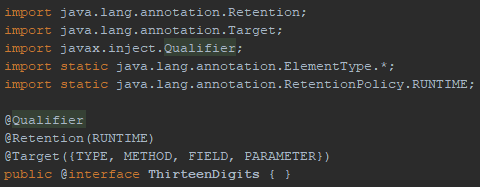
I qualificatori sono annotazioni che non fanno altro che mantenere la tipizzazione forte. Il seguente schema spiega, in maniera generale, come funzionano i qualificatori:



Il qualificatore è un’annotazione definita dal programmatore e specifica un tipo associato ad un’implementazione. È utile, quindi, quando si hanno molteplici implementazioni della stessa interfaccia.

Si supponga di avere due implementazioni dell’interfaccia NumberGenerator: IsbnGenerator e IssnGenerator. Quando si effettua l’iniezione, il container non sa quale delle due implementazioni iniettare. Si specificano, quindi, tanti qualificatori quante sono le implementazioni di quell’interfaccia.

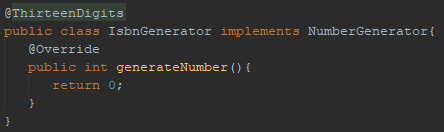
Nel seguente modo viene specificato un qualificatore denominato come ThirteenDigits.



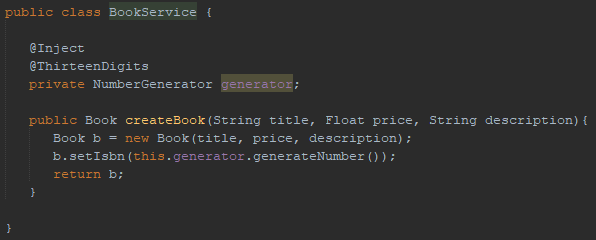
Si specifica, ovviamente, nello stesso modo anche EightDigits.

Il qualificatore ThirteenDigits, alla scelta dell’implementazione da iniettare, suggerirà l’utilizzo di IsbnGenerator, mentre il qualificatore EightDigits suggerirà l’utilizzo di IssnGenerator.

A tal punto, quindi, si specifica che le implementazioni faranno riferimento a tali qualificatori semplicemente utilizzandoli come annotazioni alle classi.



Al momento dell’iniezione, quindi, va specificato quale implementazione iniettare.



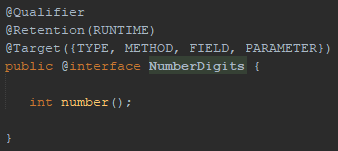
Ecco perché si dice che CDI utilizza forte tipizzazione: è possibile rinominare l’implementazione a proprio piacere, cambiarla o farci quello che si vuole, ma l’injection point non cambierà, assicurando basso accoppiamento. L’utilizzo di molteplici qualificatori, però, rende il codice più complesso e meno leggibile: si ricorre, quindi, ai qualificatori con membri.

È importante, comunque, specificare che una classe può essere comunque contraddistinta da molteplici qualificatori e non solamente da uno. Se una classe è contraddistinta da molteplici qualificatori, allora per essere iniettata dovranno essere utilizzati altrettanti qualificatori.

### 2.9.3: Qualificatori con membri

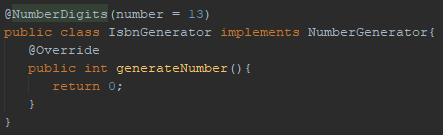
Ogni volta in cui c’è da scegliere una tra le molteplici implementazioni di un’interfaccia, è necessario l’utilizzo dei qualificatori. Come visto nel capitolo precedente, nel caso si vogliano distinguere molteplici implementazioni sarebbe necessario creare annotazioni extra (ad esempio @TwoDigits, @EightDigits, @TenDigits, @ThirteenDigits), rendendo il codice decisamente meno leggibile. È possibile evitare la creazione di un gran numero di annotazioni tramite i membri, i quali sono parametri che permettono di specificare quale implementazione utilizzare.

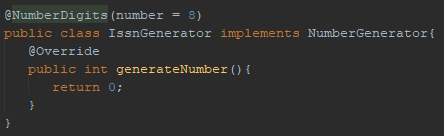
La seguente interfaccia NumberDigits permette tramite un membro denominato number di specificare quante cifre utilizza quell’implementazione, quindi le consente di contraddistinguersi dalle altre tramite un semplice parametro.



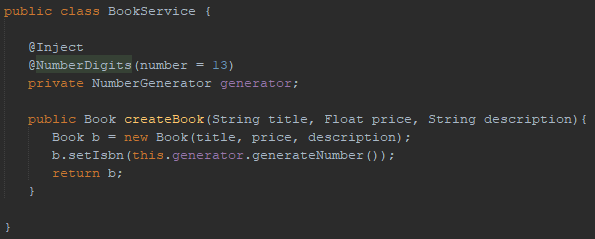
È di semplice utilizzo ed evita che si creino molteplici qualificatori inutilmente, quando poi è possibile utilizzare lo stesso per effettuare una maggiore contraddistinzione tramite dei semplici parametri. In questo esempio viene mostrato l’utilizzo di un solo parametro, ma è bene sapere che è possibile utilizzare quanti parametri si vogliano.

Vediamo ora come viene utilizzata l’annotazione sulla classe per categorizzarla con tale qualificatore.





La differenza circa il normale utilizzo è veramente minima, se non semplificata.

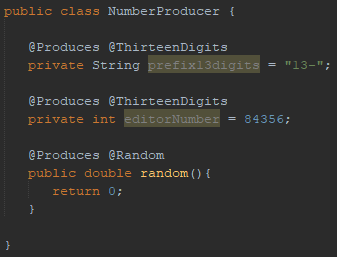


### 2.9.4: Producers - iniezione di tipi primitivi e POJO

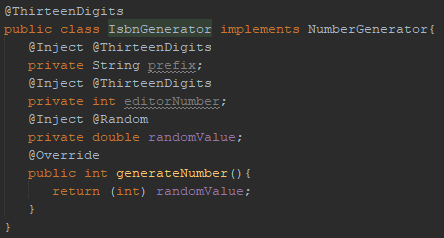
Nei capitoli precedenti abbiamo visto come iniettare CDI bean in semplici variabili. È, inoltre, possibile iniettare tipi primitivi e POJO: ciò è reso possibile dai producers. Se non fosse per i producers, non sarebbe possibile iniettare classi come Date e String, perché nel package in cui sono situate non è presente il deployment descriptor beans.xml: se beans.xml non è presente, allora CDI non sarà attivo e gli oggetti non saranno trattati come beans; di conseguenza, non saranno nemmeno iniettabili. L’unico modo per iniettare tali oggetti è tramite l’utilizzo dei producers.

Viene stabilita, innanzitutto, una classe che produca risorse associate ad un qualificatore. Nel seguente esempio, la classe NumberProducer dichiara due variabili private ed un metodo contrassegnati dai qualificatori (come solito non è obbligatorio il loro utilizzo). I qualificatori utilizzati per “differenziare” tali variabili o metodi devono essere vuoti, quindi devono essere interfacce che non forniscano metodi. Sono qualificatori vuoti, atti solamente a differenziare la risorsa. In tal caso, se si inietta una stringa utilizzando l’annotazione @ThirteenDigits, si inietterà l’apposita risorsa stringa contrassegnata con tale annotazione nel momento della dichiarazione, quindi in questo caso verrà iniettato prefix13digits.

Notiamo che la stessa annotazione può essere utilizzata per la dichiarazione di molteplici risorse, quindi quale verrà iniettata? Ebbene, si inietterà la risorsa che corrisponderà al tipo richiesto: se si richiede l’iniezione di una stringa con annotazione @ThirteenDigits, allora si otterrà la stringa dichiarata con tale annotazione. Come detto, però, l’iniezione di tipi primitivi è resa possibile tramite producers, quindi si utilizza anche l’annotazione @Produces, oltre all’annotazione per distinguere la risorsa.



Vediamo ora la classe IsbnGenerator che, a differenza dei precedenti esempi, conterrà variabili di istanza che verranno iniettate al momento dell’istanziazione: tali variabili sono tipi primitivi, quindi vengono iniettati grazie ai producers. Ovviamente, al momento dell’iniezione, per capire a quale stringa/intero/ecc si fa riferimento, si specifica il qualificatore che annota l’oggetto da iniettare al momento della sua dichiarazione.

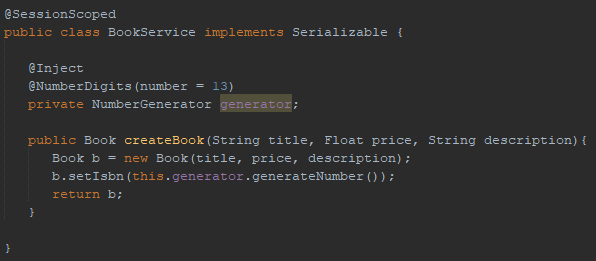


## 2.10: Contesto (scope) e annotazioni

Ogni CDI Bean ha un ciclo di vita rilegato ad un determinato contesto. In CDI, un bean è legato e rimane in tale contesto finchè non viene distrutto dal container. Non esiste alcun modo per rimuovere manualmente un bean dal contesto: può esser rimosso solamente dal container. CDI definisce alcuni contesti (detti anche scope), i quali vediamo di seguito:

* Application scope (@ApplicationScoped): si estende per tutta la durata dell’applicazione. Il bean da utilizzare viene creato una sola volta durante l’intera applicazione e viene eliminato dal container quando l’applicazione si conclude. Questo scope è utile per le classi “helper”, cioè per oggetti che conservano dati condivisi dall’intera applicazione;
* Session scope (@SessionScoped): si estende tra diverse richieste http o tra diversi metodi utilizzati per la sessione di un singolo utente. Il bean da utilizzare viene creato per la sessione HTTP e viene eliminato quando la sessione scade. Utile, appunto, per oggetti che necessitano di essere nella sessione, come le credenziali di login;
* Request scope (@RequestScoped): corrisponde ad una singola richiesta HTTP o ad una singola invocazione di un metodo. Il bean da utilizzare viene creato per la durata dell’invocazione del metodo e viene eliminato quando l’invocazione si conclude. Viene utilizzato quando i bean hanno necessità di esistere per la durata di una request HTTP;
* Conversation Scope (@ConversationScoped): si estende tra diverse invocazioni di metodi, precisamente da una determinata invocazione ad un’altra (vengono dette starting ed ending point);
* Dependent pseudo-scope (@Dependent): un bean viene creato ogni volta che viene iniettato e il suo riferimento viene rimosso quando il target dell’iniezione viene rimosso. Questo è lo scope di default per CDI.

Vediamo un semplice esempio nel quale viene applicata l’annotazione per il contesto.



Va data, però, particolare attenzione al Conversation Scope, siccome viene utilizzato tramite uno starting point e un ending point: si inietta un oggetto Conversation tramite semplice @Inject, il quale verrà utilizzato per eseguire i metodi definiti nell’intervallo dei due punti, i quali sono definiti tramite appositi metodi begin() ed end() di tale oggetto iniettato.

## 2.11: Interceptor

Gli interceptor permettono di aggiungere funzionalità ai nostri CDI bean: quando un client invoca un metodo su un CDI bean, su un EJB, ecc, il container intercetta la chiamata al metodo ed esegue l’interceptor, quindi le funzionalità aggiuntive, prima dell’esecuzione dell’invocazione.

Esistono quattro tipi di interceptor:

* Constructor-level interceptor (@AroundConstruct): interceptor eseguito all’utilizzo del costruttore della classe target;
* Method-level interceptor (@AroundInvoke): interceptor eseguito all’utilizzo di un metodo della classe target;
* Timeout method interceptor (@AroundTimeout): interceptor che si interpongono tra metodi di timeout;
* Life-cycle callback interceptor (@PostConstruct e @PreDestroy): interceptor che si interpongono tra gli eventi del ciclo di vita del bean target.

È bene sapere che gli interceptor hanno un proprio scope, quindi vengono a loro volta suddivisi in tre categorie circa il loro utilizzo:

* Target class interceptor: l’interceptor è disponibile e viene eseguito solo nella classe in cui è dichiarato, quindi solamente nella classe target (quindi lo scope è locale per la classe in cui l’interceptor è dichiarato);
* Class interceptor: l’interceptor viene isolato in una classe di supporto e viene utilizzato da altre classi specificando la classe di supporto in apposita annotazione (quindi lo scope è locale ed esterno alla classe in cui l’interceptor è dichiarato);
* Life-Cycle interceptor: riguarda i callback del ciclo di vita dei bean. Il callback viene visto come interceptor (effettivamente un callback è un interceptor) e viene richiamato nel momento adatto nel ciclo di vita del bean. Tali interceptor sono, però, isolati in una classe di supporto.

Se non fosse ancora chiaro, gli interceptor non sono altro che metodi eseguibili in determinati situazioni o eventi: come visto, possono essere eseguiti prima dell’esecuzione del costruttore, prima dell’invocazione di un metodo, durante il ciclo di vita, ecc.

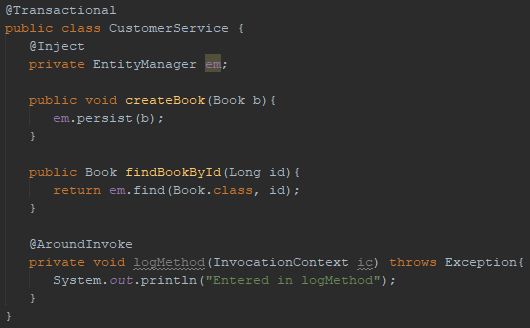
I metodi candidati ad essere interceptor devono rispettare le seguenti regole:

* Il metodo non deve essere static o final;
* Il metodo deve avere un parametro InvocationContext e deve ritornare Object e non una precisa classe (è possibile non ritornare nulla, rendendo il metodo void);
* Il metodo deve poter lanciare una checked exception.

L’oggetto InvocationContext permette agli interceptor di controllare il comportamento della catena di invocazioni: se molteplici interceptor sono concatenati tra loro tramite un’invocazione, viene passata la stessa istanza di InvocationContext, la quale permette di gestire determinati dati tra i diversi interceptor.

**Esempio di target class interceptor**

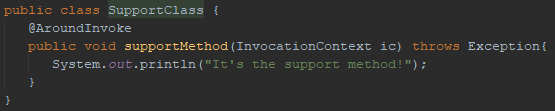
Vediamo ora un esempio di target class interceptor: viene specificato un method-level interceptor, il quale verrà eseguito prima di ogni esecuzione di ogni chiamata a metodo appartenente alla classe in cui tale interceptor è dichiarato.



**Esempio di class interceptor**

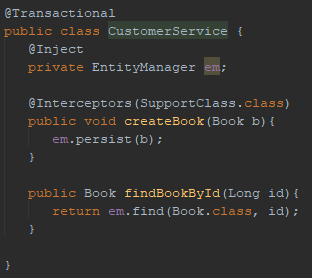
Vediamo ora un esempio di class interceptor: viene creata una classe contenente quanti interceptor si vogliano (in tal caso ne conterrà uno solo), i quali verranno utilizzati da classi esterne. In tale esempio dichiariamo, quindi, una classe contenente un interceptor che viene eseguito all’invocazione di un determinato metodo di una classe esterna.

Qui vediamo la dichiarazione della classe contenente il metodo interceptor:

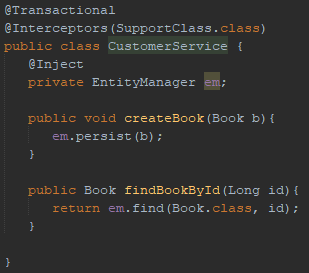


Per utilizzare l’interceptor dichiarato nella classe esterna, bisogna utilizzare l’annotazione @Interceptors(…), la quale specifica la classe dalla quale andare a ricavare gli interceptor.

Il seguente esempio mostra come è possibile richiamare gli interceptor su un solo e specifico metodo, facendo in modo che l’annotazione faccia riferimento ad un unico metodo.



Il seguente esempio, invece, mostra come è possibile richiamare gli interceptor su qualsiasi metodo della classe:



È possibile escludere l’esecuzione dell’interceptor da determinati metodi tramite l’utilizzo dell’annotazione @ExcludeClassInterceptors.

Altra nota riguarda l’utilizzo di molteplici class interceptor: è possibile includere molteplici classi tramite l’annotazione @Interceptors(…), utilizzando la seguente sintassi:

@Interceptors({C1.class, C2.class, …})

L’utilizzo di tale sintassi permette di definire anche una determinata priorità per gli interceptor, definendo quale venga eseguito per primo e quale dopo.

**Interceptor nel Deployment Descriptor**

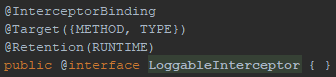
Gli interceptor sono disabilitati di default e necessitano di essere abilitati utilizzando il deployment descriptor beans.xml.



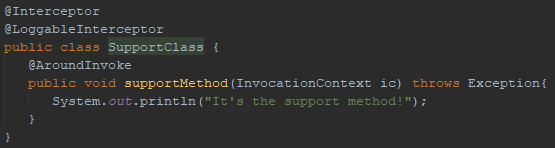
### 2.11.1: Interceptor Binding

Nel capitolo precedente abbiamo visto come funzionano gli interceptor, specificando la classe in cui sono dichiarati per poterli usare: ciò non offre basso accoppiamento, siccome si necessita di specificare una determinata classe. Di conseguenza, quindi, CDI offre l’interceptor binding, il quale introduce l’utilizzo di annotazioni definite dal programmatore che facciano riferimento alla classe contenente gli interceptor, in modo da offrire basso accoppiamento.

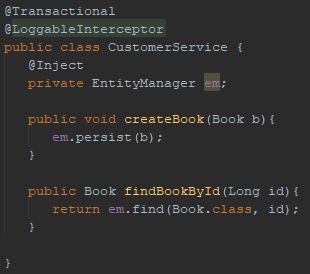
L’annotazione personalizzata interceptor binding viene creata nel seguente modo:



Ottenuta quindi l’annotazione personalizzata, la si applica alla classe contenente gli interceptor preceduta dall’annotazione @Interceptor.



Fatto ciò, si applica l’interceptor binding (precisamente, solamente l’annotazione personalizzata) ai metodi o alla classe i cui metodi verranno preceduti dall’esecuzione dell’interceptor.



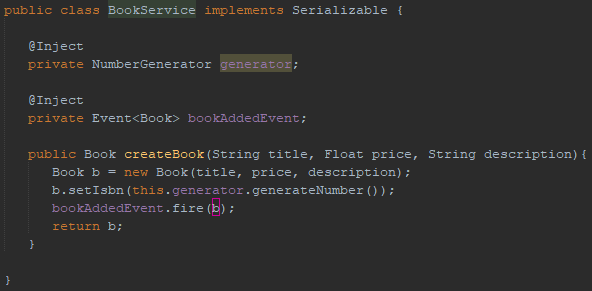
Notiamo, però, che tale definizione non regola quale interceptor viene eseguito prima dell’altro, quindi non definisce in qualche modo una priorità tra interceptor: è possibile, però, definirla manualmente tramite apposita annotazione @Priority(p), dove p è un intero che indica la priorità. Gli interceptor con priorità p minore vengono eseguiti prima.

## 2.12: Eventi

Gli eventi sono un qualcosa di astratto, definiscono quando succede qualcosa e sono gestibili: seguono il pattern Observer della Gang of Four. I bean possono definire eventi, lanciarli (viene detto “fire”) e gestirli.

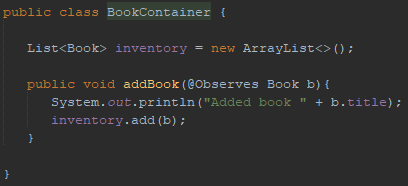
Nell’ambito degli eventi abbiamo due attori: l’event producer definisce e lancia eventi, l’observer osserva il comportamento degli eventi e li gestisce.

Nel seguente esempio definiamo un event producer, quindi una classe in grado di definire e lanciare eventi.



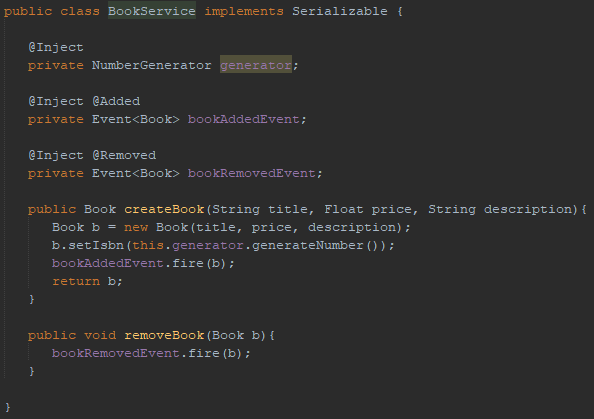
La gestione dell’evento lanciato passa, quindi, all’observer, il quale è un semplice bean con metodi riguardanti la gestione degli eventi: ognuno di questi metodi richiede uno specifico parametro annotato con @Observes ed eventuali qualificatori. Ciò fa in modo che il metodo venga notificato di un evento, controllando se il tipo di oggetto contenuto dall’evento coincida col parametro annotato.

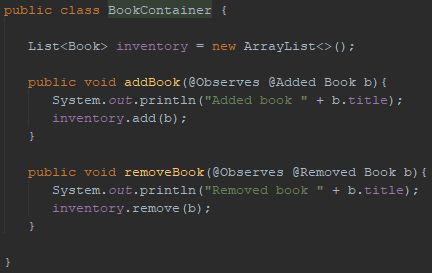
Vediamo un esempio di observer:



L’observer BookContainer contiene un metodo di gestione degli eventi addBook, riconoscibile dal fatto che abbia come parametro un oggetto annotato con @Observes.

È possibile applicare qualificatori agli eventi, in modo da poter distinguere gli eventi da altri eventi dello stesso tipo. Nel seguente esempio, grazie ai qualificatori, dichiariamo un evento lanciabile quando un libro viene creato ed un evento lanciabile quando un libro viene eliminato. Il qualificatore, in realtà, non serve tanto al lancio degli eventi, bensì serve alla loro gestione siccome, altrimenti, con la semplice annotazione @Observes si gestirebbe qualsiasi evento contenente un oggetto Book. Ciò implica che l’utilizzo dei qualificatori non renderà più unico il controllo del tipo. Dopo l’esempio riguardante la dichiarazione degli eventi e il loro lancio, vediamo anche l’esempio in cui essi vengono gestiti tramite qualificatori.

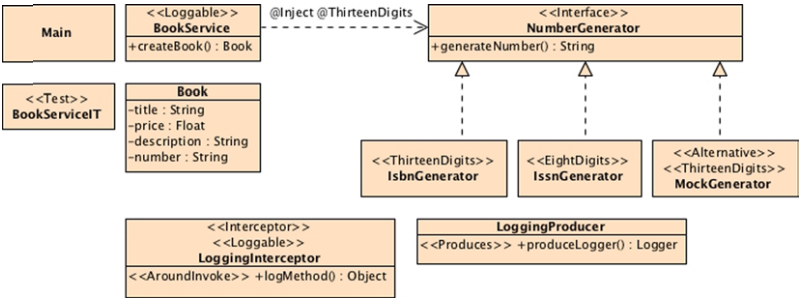




## 2.13: Esercizio riepilogativo

Abbiamo visto tanti concetti insieme affiancati da esempi con codice, come lo scrivere i bean, producers, l’utilizzo dell’iniezione, qualificatori ed interceptor binding.

Vediamo ora un grande esempio in cui raccogliamo tutti i concetti visti finora. Prima, però, è bene avere una panoramica di quel che faremo tramite un semplice class diagram.

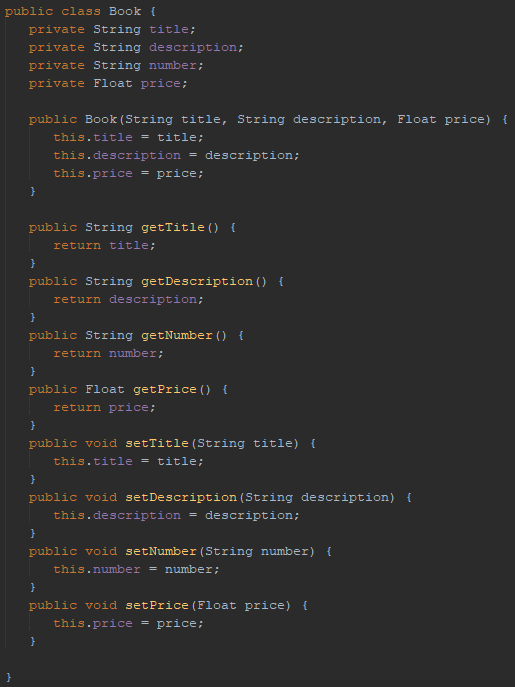


* La classe BookService contiene i metodi per la gestione dei libri, in tal caso solo riguardanti l’aggiunta;
* L’interfaccia NumberGenerator possiede due implementazioni, le quali permetteranno di generare codici ISBN e ISSN. La terza implementazione è di testing, ma non ci interessa. Le due implementazioni sono realizzate utilizzando i qualificatori;
* La classe LoggingProducer farà uso di producers per rendere iniettabile il logger, il quale terrà conto di ogni operazione che si effettuerà nel sistema;
* La classe interceptor binding LoggingInterceptor, distinto dall’interfaccia InterceptorLogger (annotazione personalizzata necessaria per l’interceptor binding), permetterà al logger di registrare le attività nel sistema in maniera del tutto automatica.

L’esempio che vedremo non sarà totalmente completo siccome non vedremo la classe Main, siccome non abbiamo ancora confidenza con i container.

### 2.13.1: Scrivere il POJO Book

Il nostro sistema utilizzerà la classe BookService per gestire libri, quindi necessitiamo di una classe Book che rappresenterà i nostri libri. Tale classe descriverà un normalissimo libro fornendo le sue caratteristiche senza far uso di annotazioni.



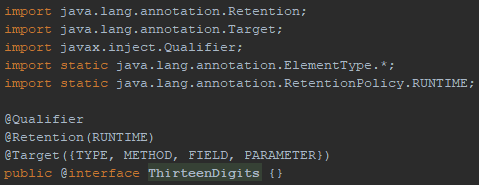
### 2.13.2: Scrivere NumberGenerator, implementazioni e qualificatori

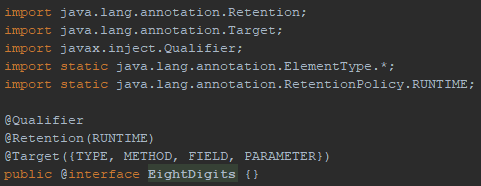
Sappiamo che ogni libro possiede un seriale che lo distingue dagli altri: tale seriale viene creato dal BookService tramite apposito oggetto generatore. Tale generatore di seriali viene iniettato grazie ad @Inject e un qualificatore, il quale specifica quale generatore iniettare. Sappiamo, quindi, di dover creare l’interfaccia NumberGenerator e le sue implementazioni con tanto di qualificatori.

Iniziamo dall’interfaccia NumberGenerator.



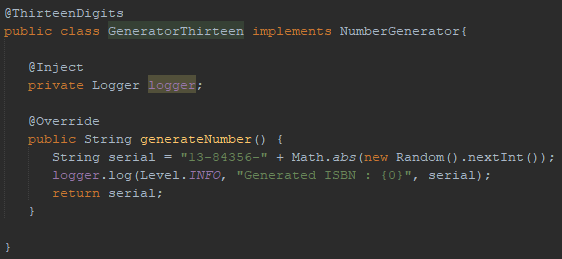
Vediamo i qualificatori che distingueranno le implementazioni.



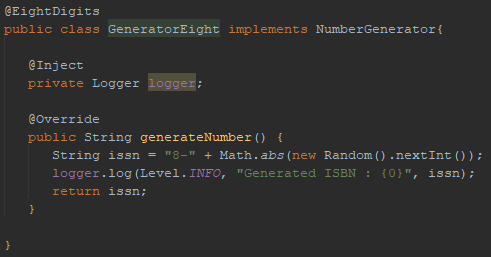


Vediamo le implementazioni dell’interfaccia NumberGenerator, le quali saranno collegate ai qualificatori che abbiamo appena stabilito.

Di seguito, vediamo l’implementazione di NumberGenerator qualificata dall’annotazione @ThirteenDigits.

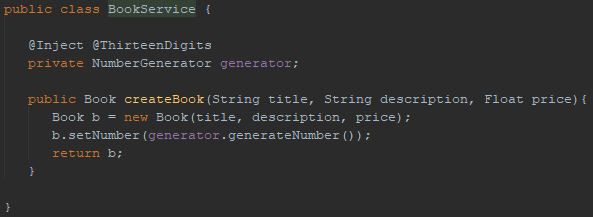


Di seguito, invece, vediamo l’implementazione di NumberGenerator qualificata dall’annotazione @EightDigits.



### 2.13.3: Scrivere il gestore dei libri BookService

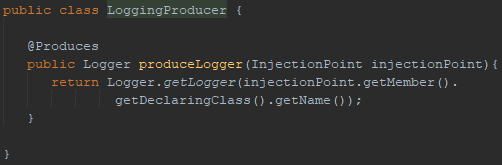
Avendo creato il POJO Book ed il NumberGenerator, il quale permette di creare identificativi per i libri, possiamo creare BookService, il quale è il gestore che permetterà l’effettiva creazione dei libri. Si vuole che la creazione del libro comporti la generazione di un seriale, il quale viene creato dal NumberGenerator.



In tal caso viene iniettata l’implementazione annotata con @ThirteenDigits, quindi nel caso si volesse utilizzare GeneratorEight al posto di GeneratorThirteen basterebbe cambiare l’annotazione con cui si inietta il NumberGenerator.

### 2.13.4: Rendere iniettabile la classe Logger

Nell’applicazione utilizziamo la classe Logger per stampare in output informazioni riguardo le istruzioni che si stanno eseguendo. La classe Logger, però, non è iniettabile siccome appartiene alla JDK e si necessita dei producers per renderla tale. Il producer è dichiarato nella classe LoggingProducer.



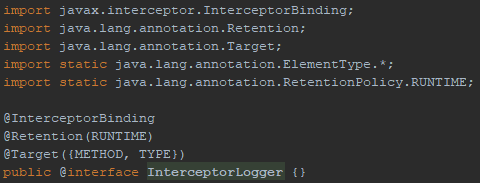
In tal modo, la classe Logger è resa iniettabile grazie ad un metodo annotato con @Produces.

Quando verrà richiesta l’iniezione di un Logger, verrà eseguito il metodo produceLogger che restituirà il logger da utilizzare.

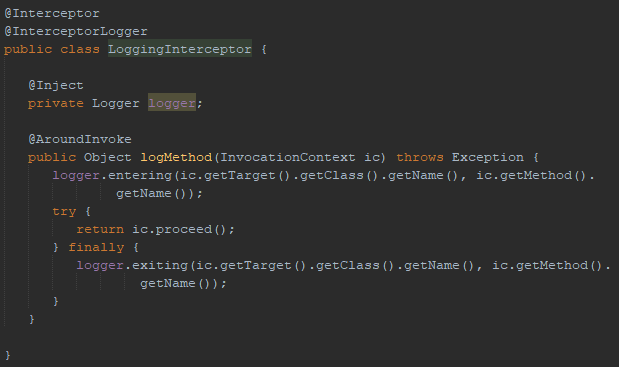
### 2.13.5: Creare l’interceptor che utilizza il Logger

Abbiamo visto l’iniezione di un Logger in molteplici classi, resa ora possibile grazie al producers. Probabilmente abbiamo fatto caso al fatto che tali Logger vengono iniettati ma non utilizzati: come mai? Ebbene, il Logger interverrà da solo all’utilizzo di determinati metodi, stampando in output informazioni riguardo il metodo che si sta utilizzando. Il fatto che il Logger intervenga da solo è reso possibile dall’interceptor.

Si vuole, quindi, creare una classe interceptor binding che contenga i metodi invocati automaticamente dall’esecuzione di determinati metodi: ciò implica che i metodi contenuti verranno annotati con @AroundInvoke, siccome è l’annotazione che stabilisce che quel determinato metodo verrà eseguito prima dell’esecuzione di un altro metodo (vedi l’apposito capitolo). Sappiamo, però, che la classe interceptor binding necessita di un’annotazione personalizzata, quindi ne vediamo la sua dichiarazione.



Creata l’annotazione personalizzata, vediamo l’interceptor binding vero e proprio.

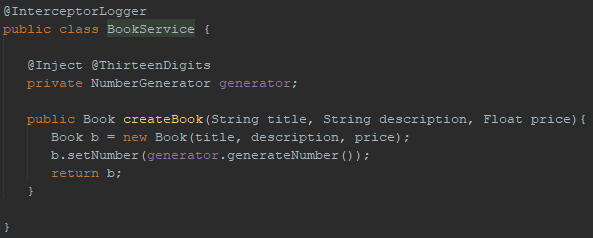


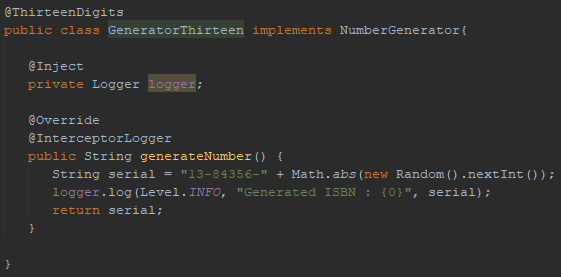
Da questo momento, quindi, sarà possibile utilizzare l’annotazione @LoggableInterceptor per far in modo che l’invocazione di un metodo venga preceduta dall’esecuzione del metodo annotato con @AroundInvoke.

### 2.13.6: Applicare l’interceptor @LoggableInterceptor

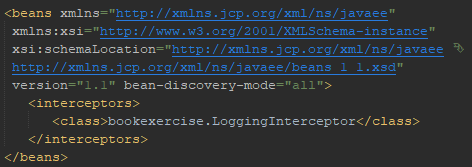
Facendo un punto della situazione, abbiamo classi che ottengono tramite iniezione il Logger ma non che lo utilizzano: per permettere ciò, è necessario annotare i metodi con @LoggableInterceptor, in modo da far sapere che la loro invocazione sarà preceduta dall’esecuzione dell’interceptor.

Refactorizziamo, quindi, le classi in cui si voglia che il Logger faccia il suo compito, aggiungendo l’annotazione per eseguire l’interceptor.





Non bisogna, però, dimenticare di abilitare l’interceptor creato. Vediamo il file beans.xml.



# Capitolo 3: Java Persistence API (JPA)

Sappiamo bene che le applicazioni sono fatte di logica di business (logica di elaborazione che rende operativa un’applicazione), interazioni con altri sistemi, interfacce e dati. Soffermandoci sui dati, sappiamo che per la loro manipolazione è necessario che i dati siano conservati in un database, nel quale possono essere anche ripresi ed analizzati. I database, quindi, conservano dati, fungendo da punto centrale alle applicazioni, e processano dati.

Nei database relazionali i dati sono conservati in tabelle composte da righe e colonne e sono distinti tra loro da chiavi primarie (primary key, PK), mentre sono collegati tra loro tramite chiavi esterne (foreign key, FK).

Gli oggetti Java non rientrano in database relazionali siccome si dovrebbero manipolare istanze di classi e non dati. Certo, sarebbe possibile estrarre i dati dalle istanze, ma questo è un altro discorso. Il problema è che, quindi, i database relazionali non possono trattare determinate cose come istanze di oggetti, interfacce, classi astratte, annotazioni, metodi, attributi, ecc.

Quel che abbiamo visto, però, non implica il fatto che non esista alcun modo per rendere tali dati persistenti. Innanzitutto, cosa si intende per dati persistenti? Dati persistenti sono dati conservabili permanentemente su memorie magnetiche, memorie flash, ecc. Detto ciò, introduciamo l’Object Relational Mapping (ORM), fornito da JPA, il quale interfaccia il mondo dei database al mondo degli oggetti. In realtà di ORM ne vedremo solamente il suo funzionamento in maniera estremamente generale: ci concentreremo in particolare sull’utilizzo delle query e sulla Persistence Unit.

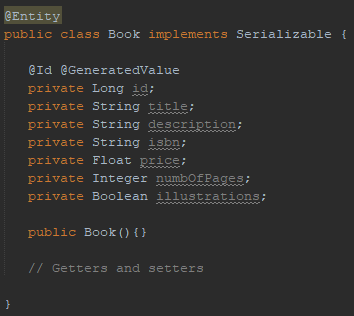
## 3.1: Introduzione alle entità

Quando si parla di oggetti in relazione ai database, sarebbe più corretto utilizzare il termine “entità” anziché “oggetto”. Gli oggetti sono istanze che vivono temporaneamente in memoria, mentre le entità sono istanze che possono essere rese persistenti nel database. È possibile rendere persistente un’entità, è possibile rimuoverla dal database o effettuare una query su di essa utilizzando il linguaggio JPQL (Java Persistence Query Language).

Un’entità, sostanzialmente, è un normalissimo POJO: quindi dovrà essere dichiarata, istanziata e utilizzata come qualsiasi altra classe Java. Ovviamente l’entità, come qualsiasi altra classe Java, avrà un suo stato e le sue caratteristiche saranno manipolabili tramite getter e setter.

## 3.2: Anatomia di un’entità

Per essere entità, una classe deve essere annotata con @Entity, la quale annotazione permette al gestore della persistenza di riconoscere tale classe come classe persistente e non come semplice POJO. Inoltre, un’entità deve possedere un identificativo, il quale distingue l’entità dalle altre: l’identificativo è specificabile tramite l’annotazione @Id e raffigura la primary key dell’entità.



In sostanza, quindi, un’entità, per essere tale, deve seguire le seguenti regole:

* La classe dell’entità deve essere annotata da @Entity;
* L’entità deve possedere un identificativo, dichiarabile con @Id;
* L’entità deve possedere un costruttore senza alcun argomento, quindi un costruttore vuoto. Ciò non implica il fatto che non possano esserci altri costruttori oltre a quello vuoto;
* L’entità non può essere un’enumerazione o un’interfaccia;
* La classe dell’entità non deve essere final, come nessun suo metodo o variabile;
* Se l’istanza dell’entità viene passata per valore come oggetto detached (vedremo più avanti cosa significa), la classe dell’entità deve implementare Serializable.

## 3.3: Riguardo l’Object Relational Mapping (ORM)

Il principio di ORM è quello di delegare a tools o framework esterni (nel nostro caso, JPA) il compito di creare una corrispondenza tra oggetti e database. Di norma, è possibile estrarre i dati dagli oggetti per poi salvarli nelle tabelle dei database relazionali, ma nel caso si vogliano salvare le entità al posto delle tabelle? Come fa JPA a mappare gli oggetti nel database? Tramite annotazioni e descrittori XML (XML descriptors). Senza alcuna annotazione, l’entità verrebbe trattata come un normale POJO e non sarebbe adatta ad un database: per cambiare tale comportamento, si annota la classe con @Entity, rendendola persistente e, quindi, adatta ad un database.

Come possiamo ben vedere, una semplice annotazione stravolge il comportamento che avrebbe il gestore della persistenza verso tale oggetto. Altro esempio è l’annotazione @Id: con tale annotazione, un attributo viene reso chiave primaria dell’entità.

Tali cambiamenti raffigurano l’approccio “configuration-by-exception”, cioè vengono fatti dei cambiamenti alle regole di mapping di default, quali sono:

* Il nome dell’entità rappresenta il nome della tabella nel database relazionale. Se si vuole mappare l’entità in un’altra tabella, è necessario utilizzare l’annotazione @Table;
* I nomi degli attributi rappresentano i nomi delle colonne. Se si vuole il comportamento di default, basta utilizzare l’annotazione @Column.

Esistono ovviamente tante altre regole, ma ne abbiamo viste solo un paio in modo da renderci conto di come un’annotazione possa cambiare le regole di mapping di default.

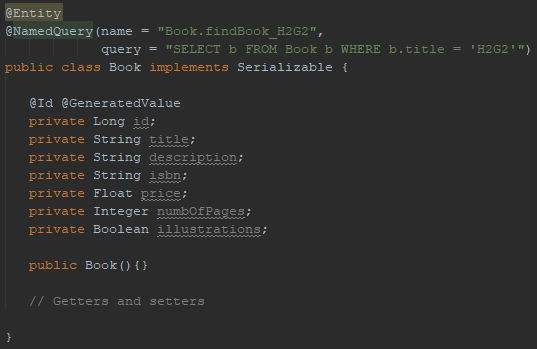
## 3.4: Introduzione alle query e all’Entity Manager

JPA permette di mappare le entità in database e di effettuare query su entità e le loro relazioni secondo diversi criteri, trattando tutto ciò con un approccio object-oriented, senza avere a che fare con righe e colonne. Tali query utilizzano JPQL, il quale è molto simile al normale SQL, con la differenza che la sua sintassi tratta oggetti e, quindi, utilizza la notazione con il punto ( . ) per trattare attributi.

SELECT b FROM Book WHERE b.title = “H2G2”

Come ben vediamo non si trattano più le colonne delle tabelle, bensì si ricavano gli oggetti e si trattano gli attributi.

Le query JPQL, principalmente, possono essere create dinamicamente a runtime o staticamente a tempo di compilazione. Il seguente esempio mostra la query statica, cioè la named query. Gli altri tipi di query li vedremo più avanti siccome questa è solamente un’introduzione.

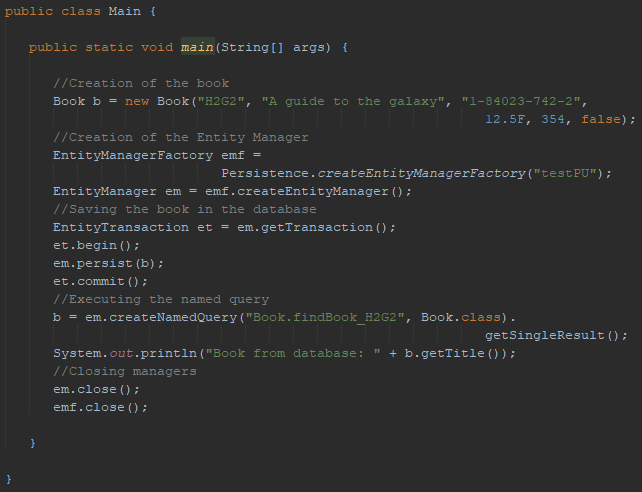


Il responsabile della gestione delle entità è l’Entity Manager. Il suo ruolo è quello di gestire le entità secondo le operazioni CRUD (Create, Read, Update, Delete) utilizzando JPQL. È definibile come un’interfaccia da utilizzare per la gestione delle entità, la cui implementazione è fornita dal gestore della persistenza. In poche parole, l’Entity Manager rende persistenti le entità, le aggiorna e le rimuove dal database; inoltre, può effettuare query sulle entità utilizzando JPQL.

L’Entity Manager è ottenibile tramite un factory, come vediamo nel seguente codice.



Essendo le query eseguibili solamente dall’Entity Manager, si necessita forzatamente di esso quando si parla di persistenza. Vediamo ora un esempio completo per poi commentarlo.

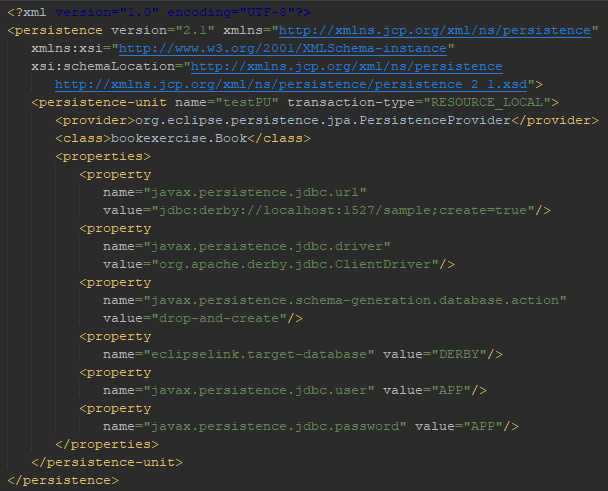


Lo scopo dell’esempio è quello di mostrare come un libro viene salvato e ripreso dal database. Avendo un oggetto Book (il quale è un’entità, quindi annotato con @Entity), lo si vuole salvare nel database e, dopo il salvataggio, riprenderlo: per fare ciò sappiamo che è necessario l’Entity Manager, il quale è ricavabile da apposito factory. Ottenuto l’Entity Manager, si apre una transazione in cui si rende persistente l’oggetto (vedremo le transazioni tra qualche capitolo): tramite il metodo persist, si salva l’oggetto nel database, quindi viene reso persistente (in realtà non è proprio così, approfondiremo tra qualche paragrafo). Fatto ciò si vuole riprendere l’oggetto dal database, quindi si esegue la named query: la si crea tramite apposito metodo e la si esegue facendosi restituire il risultato. Fatto ciò, chiudiamo l’Entity Manager ed il suo factory.

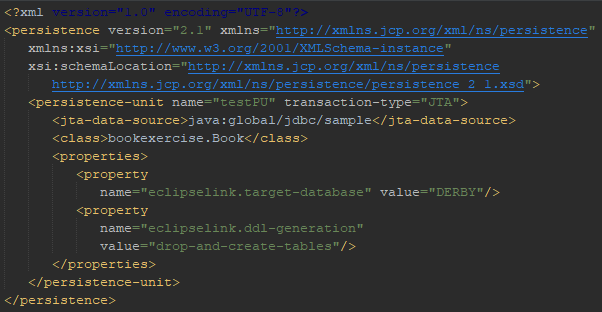
## 3.5: Persistence Unit

Abbiamo finora discusso in generale il funzionamento delle query e dell’entity manager, quindi stiamo iniziando a toccare con mano il database. Abbiamo visto un esempio che mostra come si interagisce con il database, ma non abbiamo visto quale driver viene utilizzato per la comunicazione, come viene effettuata la connessione, ecc. L’esempio completo del precedente paragrafo pecca di numerose informazioni fornite dalla Persistence Unit.

La Persistence Unit (PU) indica all’Entity Manager il tipo di database da utilizzare e i parametri di connessione, definiti nel file persistence.xml posto nella cartella META-INF.



Prima di continuare, è bene fare un piccolo ed importante commento su un paio di cose. È bene sapere che l’ambiente in cui si lavora può essere application-managed e container-managed (vedremo più avanti cosa significa): se application-managed, vanno trascritte nel persistence.xml tutte le informazioni riguardanti il collegamento al database, proprio come nell’esempio appena visto; se container-managed, il discorso cambia siccome non si specificano più le informazioni riguardanti il collegamento, bensì si specifica il data source (il quale serve, appunto, per collegarsi al database). Vediamo anche un esempio di persistence.xml per ambienti container-managed.



## 3.6: Accenni sul ciclo di vita delle entità e le callbacks

Come già anticipato, le entità non sono altro che POJO gestiti dall’Entity Manager, il quale permette di assegnare identificativi alle entità e permette al database di salvarne il loro stato. Nel caso le entità non siano gestite dall’Entity Manager, allora saranno considerate come normalissimi oggetti Java, quindi saranno POJO, e non avranno nulla a che fare con il database.

Quando si crea un’istanza di una determinata classe con l’operatore new, essa esisterà in memoria e JPA non saprà nulla riguardo la sua esistenza. Dal momento in cui l’Entity Manager inizia a gestire l’istanza creata, essa verrà mappata nel database e ne verrà sincronizzato il suo stato.

Un’entità può essere resa dall’Entity Manager persistente, aggiornata, rimossa e caricata: tali operazioni corrispondono alle operazioni database di inserimento, aggiornamento, rimozione e selezionamento.

Ogni operazione è caratterizzata da due eventi “pre” e “post” (tranne per il caricamento, il quale possiede solamente l’evento “post”): l’evento “pre” si manifesta prima dell’operazione; l’evento “post” si manifesta dopo l’operazione. L’evento manifestante può essere intercettato dall’Entity Manager in modo da poter invocare metodi di business annotati da determinate annotazioni, in base al tipo di operazione che si sta eseguendo: ad esempio, per la resa persistente vengono utilizzate le annotazioni @PrePersist e @PostPersist.

Maggiori dettagli nel capitolo 4.2.

# Capitolo 4: Gestione degli Oggetti Persistenti

In JPA, il servizio che manipola le entità è l’Entity Manager, il quale crea, trova, rimuove e sincronizza oggetti con il database. Inoltre, l’Entity Manager esegue molteplici tipi di query JPQL, in modo da poter operare con completa libertà sulle entità. JPQL è il linguaggio definito in JPA per effettuare le query su entità conservate nei database. La sintassi di tale linguaggio ricorda decisamente SQL, ma con un approccio Object Oriented siccome vengono trattati gli oggetti. Le query non fanno altro che gestire le entità secondo le operazioni CRUD (create, read, update, delete).

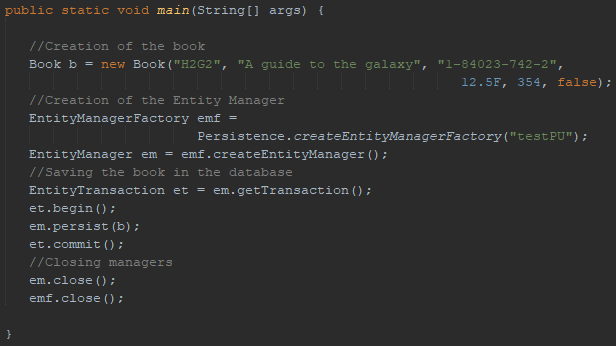
## 4.1: Entity Manager

L’Entity Manager è il pezzo centrale di JPA: gestisce il ciclo di vita delle entità ed esegue le query, trattando le entità nel database. Tale componente di JPA è responsabile riguardo la creazione e la rimozione di entità persistenti (ricordiamo che un’entità è detta persistente se si trova nel database) e riguardo la loro ricerca in base alla loro chiave primaria (ID). Quando un Entity Manager gestisce un’entità, quindi ne ottiene un riferimento ad essa, quest’ultima viene detta “managed”; se un’entità non risulta managed, viene detta “detached”. La differenza tra un’entità managed ed una detached consiste nella sincronizzazione con il database, la quale avviene solo per le entità managed.

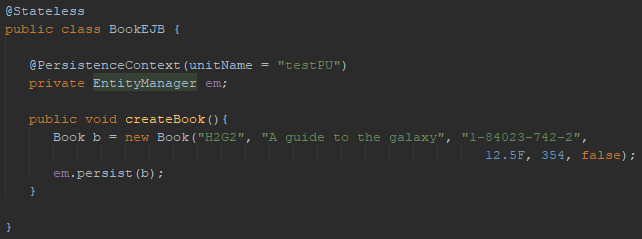
L’Entity Manager è un’interfaccia implementata da un persistence provider, il quale genera ed esegue statements in SQL. Non ne vedremo tutti i metodi, bensì vedremo direttamente i suoi utilizzi e i metodi che usa in determinate occasioni.

### 4.1.1: Ottenere un Entity Manager

L’Entity Manager è un’interfaccia utilizzata per interagire con le entità, ma per essere utilizzata deve essere ottenuta dall’applicazione: il modo in cui viene ottenuta dipende dal tipo di ambiente, il quale può essere gestito dal container o dall’applicazione (container-managed environment, application-managed environment). Ad esempio, nell’ambiente gestito dal container non c’è il bisogno di utilizzo dei metodi commit e rollback, a differenza dell’ambiente gestito dall’applicazione. In sostanza, il tipo di ambiente impatta sulla gestione dell’istanza dell’Entity Manager e sul ciclo di vita. Nel seguente esempio vediamo come il programmatore è responsabile riguardo la creazione e la chiusura dell’Entity Manager (ciò rientra nella gestione del ciclo di vita di quest’ultimo).



La creazione di un Entity Manager in un ambiente application-managed è relativamente semplice, ma nel container-managed è un pochino diverso, lo vediamo subito in un esempio:



L’ottenimento dell’Entity Manager in ambienti container-managed consiste nell’utilizzo dell’annotazione @PersistenceContext o tramite JNDI lookup (lo vedremo più avanti). Inoltre, come già anticipato, nel container-managed non ci sarà il bisogno di creare o chiudere l’Entity Manager, siccome il suo ciclo di vita è gestito dal container. L’annotazione @PersistenceContext richiede come parametro il nome della Persistence Unit.

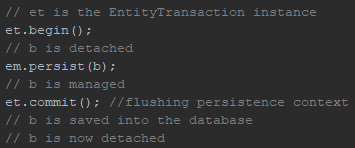
### 4.1.2: Entità managed, detached e relazione con l’appartenenza al database

Prima di approfondire per bene l’Entity Manager, è necessario comprendere a fondo il concetto di entità managed e detached. È bene premettere che, a differenza di come si sia capito finora, il fatto che un’entità sia managed o detached non c’entra con la sua appartenenza al database. Sia chiaro sin da ora: il fatto che un’entità appartenga al database non implica il fatto che sia managed o detached.

Un’entità è detta managed se gestita dall’Entity Manager, quindi se ne possiede un riferimento ad essa, nel senso che in una transazione (o in generale) sta operando su di essa. Un’entità è detta detached se non gestita dall’Entity Manager.

Ora come ora non conosciamo i metodi e non abbiamo idea di cosa sia una transazione, ma ci basta sapere un paio di cose che approfondiremo successivamente: il metodo persist, al contrario di come detto precedentemente per semplificare le cose, non salva l’oggetto nel database, bensì lo rende managed; le entità managed sono raccolte in un insieme detto persistence context; le entità managed vengono salvate nel database solamente quando il persistence context viene svuotato; se un’entità non appartiene al persistence context, essa allora sarà detached.

Vediamo un esempio:



A questo punto si presenta un gran bel dubbio: come viene salvata l’entità b nel database? Ebbene, se la transazione va a buon fine effettuerà un flush automatico del persistence context, liberandolo di ogni entità e salvandole nel database. Nel caso la transazione non vada a buon fine, viene effettuato un rollback, quindi si ritorna allo stato precedente all’inizio della transazione.

Comprendiamo, finalmente, che un’entità può essere managed o detached a prescindere dal fatto che si trovi o meno nel database.

### 4.1.3: Persistence Context

Il persistence context (contesto di persistenza) può essere visto allo stesso tempo sia come un livello posto tra Entity Manager e database, sia come un insieme di entità gestite dove non compaiono doppioni per identificativi: ad esempio, se esiste un libro con ID pari a 12 nel persistence context, non ne potrà esistere un altro con lo stesso ID.

Il persistence context contiene tutte le entità che risultano managed, quindi gestite dall’Entity Manager: i cambiamenti di stato effettuati su tali entità verranno riportati automaticamente nel database.

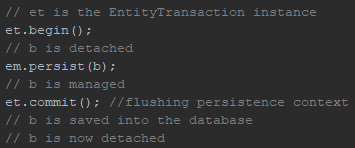
L’Entity Manager, all’invocazione di metodi, consulta ed eventualmente aggiorna il persistence context: ad esempio, se viene chiamato il metodo persist, l’entità passata come argomento dovrà essere aggiunta al persistence context se non esista già al suo interno. Tale controllo viene effettuato, ovviamente, in base all’ID dell’entità.

Il persistence context può essere visto come una memoria cache che contiene tutte le entità in attesa di essere inserite nel database: sostanzialmente, le entità sono conservate in tale spazio di memoria per la durata di una transazione (tra non molto vedremo cosa sia, ma sostanzialmente è un insieme di operazioni). Conclusa la transazione, viene effettuato un flush sul persistence context e tutte le sue entità vengono riposte nel database.

L’annotazione @PersistenceContext viene utilizzata per ottenere un Entity Manager in ambienti container-managed, specificando la Persistence Unit da utilizzare, definita nel file persistence.xml posto nella cartella META-INF. Abbiamo già visto tale file in un capitolo ad esso dedicato, quindi non ci soffermeremo a vederlo. Piuttosto, ci soffermiamo sul tag class: nel persistence.xml devono essere specificate tutte le entità che possono essere gestite nel persistence context. Inoltre, se si parla di ambienti container-managed, deve essere specificato anche il tipo di transazione come JTA (transaction-type=“JTA”).

### 4.1.4: Rendere persistente un’entità tramite persist

Rendere persistente un’entità significa crearne un’istanza, settarle gli attributi e chiamare il metodo persist dell’Entity Manager. Tramite tale metodo, l’entità entrerà a far parte del persistence context, quindi verrà considerata managed. Attenzione: l’utilizzo di tale metodo non implica il fatto che l’entità venga salvata nel database: ciò accade nel caso venga effettuato un flush sul persistence context.



In tal caso, il metodo persist viene eseguito nel mezzo di una transazione: l’aggiunta al database sarà effettuata solo al commit della transazione siccome genera un flush sul persistence context.

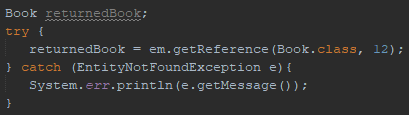
### 4.1.5: Ricavare un’entità dal database tramite find e getReference

Per ricavare un’entità dal database tramite il suo ID, stabilito tramite l’annotazione @Id, è possibile utilizzare due metodi differenti.

Il primo metodo consiste nell’utilizzo del metodo find dell’Entity Manager, il quale accetta due parametri: la classe dell’entità e l’identificativo da cercare.

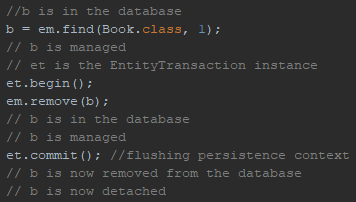


Il secondo metodo consiste nell’utilizzo del metodo getReference dell’Entity Manager, il quale funziona come il metodo find con la differenza che ritorna un riferimento all’oggetto, senza ricavarne i suoi dati, fungendo da proxy.



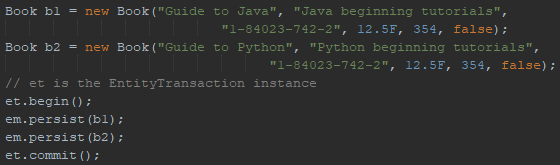
### 4.1.6: Rimuovere un’entità dal database tramite remove

Un’entità può essere rimossa dal database tramite il metodo remove dell’Entity Manager. L’utilizzo di tale metodo al di fuori di una transazione rende l’entità passata come argomento detached; se si sta lavorando in una transazione, l’entità rimane managed finchè non viene effettuato il commit, siccome è lì che verrà registrata l’eliminazione e l’operazione di detach. Ovviamente l’entità viene rimossa dal database, quindi sarà comunque accessibile al di fuori di quest’ultimo.

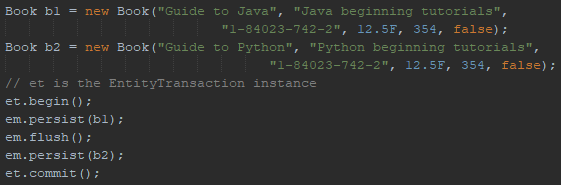


### 4.1.7: Sincronizzazione con il database tramite flush e refresh

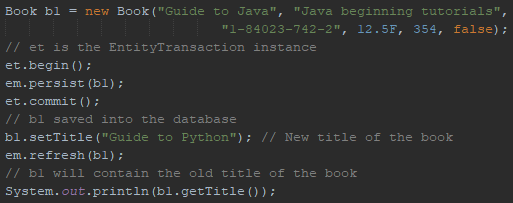
Fino ad ora abbiamo visto della sincronizzazione con il database solamente il salvataggio dell’entità tramite flush automatico o la rimozione. In realtà, ci sono altri metodi che riguardano la sincronizzazione delle entità con il database e in questo paragrafo ne vedremo un paio. Ribadendo nel caso non si fosse capito, la commit di una transazione non fa altro che eseguire un flush di ciò che è conservato nel persistence context, salvando le entità presenti al suo interno. Nel seguente esempio vengono effettuate due persist, quindi si vogliono salvare due entità: tali entità vengono inserite nel persistence context ma non nel database. Tramite la commit si effettua un flush sul persistence context, quindi le due entità verranno registrate nel database.



Volendo, per qualsiasi ragione, è possibile esplicitare il flush tramite un apposito metodo dell’Entity Manager chiamato flush. Utilizzando questo metodo, il persistence provider sarà esplicitamente forzato a flushare i dati nel database, ma ciò non committerà la transazione.



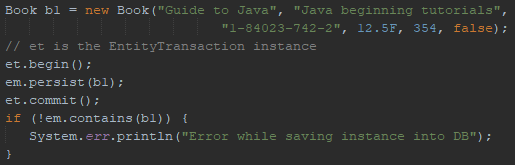
Quindi, se la flush registra i cambiamenti dalla normale esecuzione verso il database, allora ci sarà un metodo che registrerà i cambiamenti dal database alla normale esecuzione: stiamo parlando di refresh, altro metodo dell’Entity Manager, il quale sovrascrive l’attuale stato dell’entità con i dati presenti nel database.



Nota: la flush automatica delle transazione rende detached le entità; la flush forzata le rimane managed. Leggi il capitolo 4.1.9 per comprendere al meglio.

### 4.1.8: Contenimento di un’istanza tramite contains

Le entità, come ben sappiamo, possono essere gestite o meno dall’Entity Manager. L’Entity Manager fornisce un metodo, contains, il quale permette di controllare se una determinata entità risulta essere tra quelle gestite.

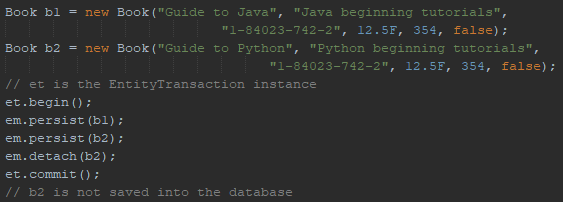


### 4.1.9: Rendere detach un’entità

È possibile manipolare lo stato delle entità portandolo da managed a detached tramite due metodi: clear permette di svuotare il persistence context rendendo ogni entità detached (ciò implica non vengano salvate nel database); detach permette di rendere un’entità detached.

Un’entità può essere resa detached anche in un altro modo, oltre all’utilizzo dei metodi clear e detach: quando si utilizzano le transazioni e si usa rendere le entità persistenti dentro esse (transaction scoped persistence context), il commit svuota il persistence context salvando le entità nel database e rendendole detached. Nel caso ci abbiate fatto caso, negli esempi riportati, le entità diventano detached subito dopo il commit. Ciò non avviene con il flush forzato, tramite esecuzione dell’apposito metodo.

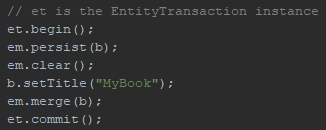
Nel seguente esempio viene riportato l’utilizzo del metodo detach.



### 4.1.10: Aggiornare un’entità nel database

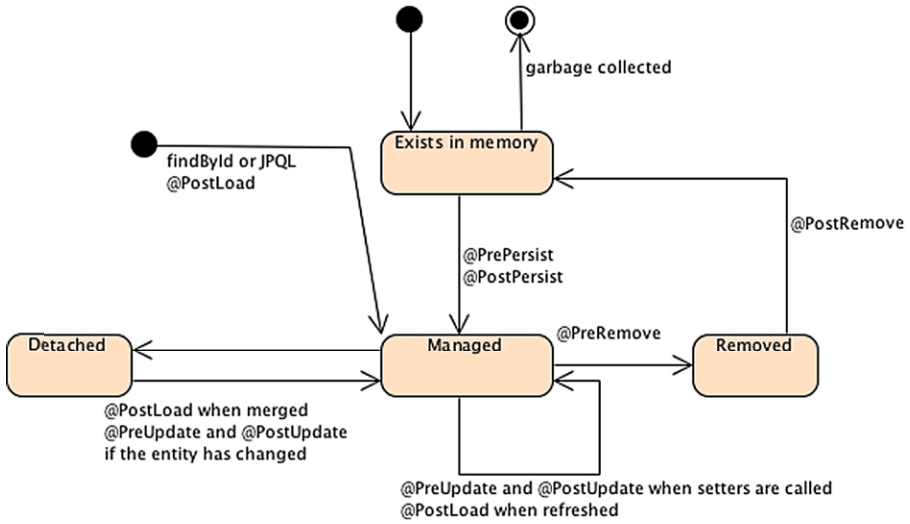
L’aggiornamento dell’entità nel database è decisamente più semplice di quel che si aspetta. Come già anticipato precedentemente, se l’entità è managed allora gli aggiornamenti al database saranno automatici. Il problema si presenta se si ha a che fare con un’entità detached: in tal caso, viene utilizzato il metodo merge.

Il metodo merge si utilizza solamente per le entità che non sono più associate al persistence context, quindi solo per le entità detached di cui si ha la necessità di aggiornarla nel database, e serve ad apportare le modifiche apportate sull’entità nel database.



## 4.2: Ciclo di vita delle entità

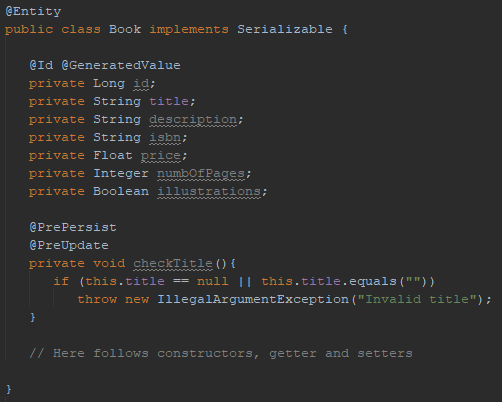
Di seguito viene mostrato uno schema che raffigura gli stati delle entità in base alle esecuzioni dei metodi dell’Entity Manager. Nello schema non si accennano la transazioni: è bene sapere che per la gestione delle entità non sono necessarie le transazioni. Il loro utilizzo dipende dal tipo di ambiente che si sta utilizzando. Riprendendo il capitolo 4.1.1, notiamo che nell’ambiente container-managed non vengono usate le transazioni per effettuare la commit, quindi cambiano i modi di liberare il persistence context. Sostanzialmente, in generale, vale lo schema riportato; se si parla di transazioni, è bene sapere che il commit libera il persistence context, rendendo tutte le sue entità detached.



### 4.2.1: Callbacks

Il ciclo di vita delle entità, quindi, si dirama in quattro categorie: persistenza, aggiornamento, rimozione e caricamento, le quali corrispondono alle operazioni di database di inserimento, aggiornamento, rimozione e selezione. Ogni diramazione del ciclo di vita ha degli eventi “pre” e “post” che possono essere intercettati dall’Entity Manager per invocare metodi di business. I metodi di business che si desidera invocare in base agli eventi, devono essere annotati da determinate annotazioni, le quali dipendono dal tipo di evento che si sta verificando. Nello schema soprastante, vengono mostrate tutte le annotazioni utilizzabili in base all’evento verificato. I metodi callback vanno posti nella classe su cui si sta operando.

Vediamo un esempio di callback:



## 4.3: Query

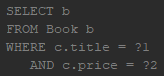
Come già anticipato, l’Entity Manager è in grado di eseguire query scritte in JPQL: ne esistono ben 5 tipi, ognuna con un funzionamento diverso: nei prossimi paragrafi vedremo le più rilevanti.

Per eseguire query di selezione, quindi che utilizzino SELECT, è possibile scegliere tra due metodi in base al result che si vuole ottenere: il metodo getResultList esegue la query e ritorna una lista di risultati; il metodo getSingleResult esegue la query e ritorna un singolo risultato (lancia un’eccezione se trova molteplici risultati).

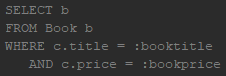
Per eseguire query di aggiornamento, quindi che utilizzino UPDATE o DELETE, si utilizza il metodo executeUpdate, il quale ritorna il numero di entità affette dall’aggiornamento.

La creazione di una query è davvero molto semplice: basta utilizzare il metodo createQuery dell’Entity Manager, il quale accetta come parametro la query in JPQL. L’esecuzione di tale metodo ritorna un oggetto di tipo Query o TypedQuery, in base al numero di argomenti passati (vedremo tra poco).

Le query caratterizzate dal WHERE possono essere caratterizzate da parametri variabili, i quali possono essere specificati in due modi: tramite i parametri posizionali ed i parametri con nome. I parametri posizionali sono caratterizzati da un punto di domanda seguito da un intero, il quale parte da 1 e continua a crescere andando avanti nella query.

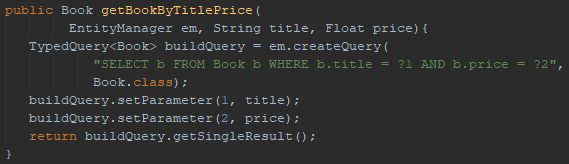


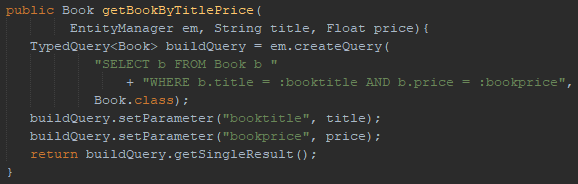
I parametri con nome sono caratterizzati da un carattere di due punti seguito dal nome del parametro, il quale verrà utilizzato per impostare un valore.



Per impostare valori ai parametri che abbiamo appena visto, basta utilizzare il metodo setParameter, caratterizzato da due argomenti: il primo argomento indica il parametro su cui impostare il valore, mentre il secondo argomento indica il valore da impostare.

Il primo esempio che vediamo mostra come vengono impostati i parametri posizionali, mentre il secondo mostra come vengono impostati i parametri con nome.

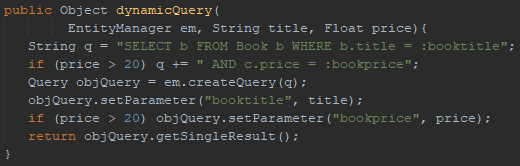




Inoltre, è bene sapere che è possibile gestire le query in modo tipizzato e non. Quando si crea una query, in generale, essa viene conservata in una variabile Query o TypedQuery: se si esegue la query, essa ritornerà un tipo di oggetto in dipendenza dalla variabile scelta per contenerla. Se si utilizza Query, l’esecuzione ritornerà una lista di oggetti non tipizzati; se si utilizza TypedQuery<TYPE>, l’esecuzione ritornerà una lista di oggetti con tipo specificato TYPE. L’utilizzo di TypedQuery richiede un argomento aggiuntivo nel metodo createQuery: sarà necessario passare come secondo parametro la classe del tipo TYPE.

### 4.3.1: Dynamic Query

Le dynamic query vengono definite e tradotte in SQL al momento durante l’esecuzione dell’applicazione. Sono dette dinamiche proprio perché possono essere costruite tramite concatenazione a runtime, anche in base a determinati controlli e condizioni.

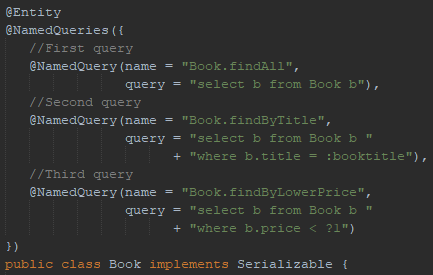


### 4.3.2: Named Query

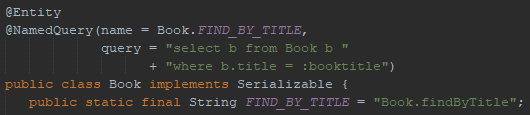
Le named query sono statiche e non possono cambiare per alcun motivo, quindi non sono per nulla flessibili come le query dinamiche. Il loro essere statiche porta ad essere decisamente efficienti siccome il loro JPQL viene tradotto in SQL non appena l’applicazione parte, invece di tradurre a runtime.

Le named query vengono specificate in nell’annotazione @NamedQuery posta davanti alla definizione della classe di ritorno. Se la classe di ritorno possiede molteplici query, allora le annotazioni vengono raccolte in un’unica annotazione @NamedQueries.

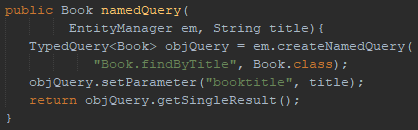
L’annotazione @NamedQuery richiede due parametri: il primo è il nome della query, il secondo è la query stessa. L’annotazione @NamedQueries, invece, richiede come parametro tutte le @NamedQuery, raccolte tra le parentesi { }.



Per convenzione si usa adottare la seguente sintassi per il nome completo della query: ClassName.QueryName, dove ClassName è il nome della classe a cui si fa riferimento e QueryName è il nome reale della query. È una convenzione abbastanza obbligata, siccome potrebbero esistere, ad esempio, molteplici query con nome “findAll”, mentre ne esisterà sicuramente e solamente una con nome “Book.findAll”. Altra convenzione è quella di stabilire il nome della query come costante della classe, in modo da favorire modularità per eventuali cambi alla query.

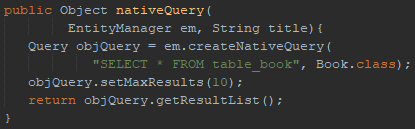


Definita la query, si passa alla creazione dell’oggetto che gestisce la query: in generale si sarebbe utilizzato il metodo createQuery con parametro la query in JPQL. Per le named query deve essere utilizzato il metodo createNamedQuery, anch’esso dell’Entity Manager, il quale accetta come parametro il nome stabilito della query. Nel caso si voglia gestire un TypedQuery, come già visto, vale anche qui il parametro addizionale. La gestione della query è invariata, cambia solamente, quindi, la sua creazione.



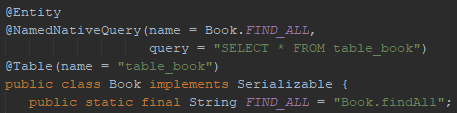
### 4.3.3: Native Query

Per gli appassionati di SQL che non amano le tecnologie diverse, ecco le native query! Questo tipo di query opera su sorgente SQL e la sua creazione ritorna sempre un oggetto Query (quindi con le native query non è possibile utilizzare TypedQuery). Per creare una native query deve essere utilizzato il metodo createNativeQuery, appartenente all’Entity Manager, il quale richiede due parametri: la query in SQL e la classe a cui la tabella specificata fa riferimento. La gestione della query è invariata.



**Native Named Query**

Variante delle native query sono le native named query, le quali sono un ibrido tra le native query e le named query: vengono specificate proprio come le named query, con la differenza che l’annotazione utilizzata è @NamedNativeQuery, la quale prende come parametri il nome della query e la query vera e propria in SQL. Inoltre, va specificata tramite l’annotazione @Table il nome della tabella a cui la classe che segue fa riferimento, in modo da mapparla alla tabella trattata nella query in SQL. Vediamo l’esempio:



Ovviamente nel caso si cerchi di fare qualcosa di più elaborato, bisognerà mettere mano ad annotazioni di mapping con database relazionali.

Per raccogliere molteplici annotazioni @NamedNativeQuery, proprio come con le named query, si utilizza l’annotazione @NamedNativeQueries.

# Capitolo 5: Enterprise JavaBeans

«Da 227»

## 5.1: Introduzione agli EJB

## 5.2: Scrivere gli EJB