1. **Descrivere la Context and Dependency Injection**

   Java EE 6 ha introdotto Context and Dependency Injection (CDI) che trasforma quasi tutti I compnenti Java EE in beans injectable (iniettabili), interceptable (intercettabili) e manageable (gestibili).

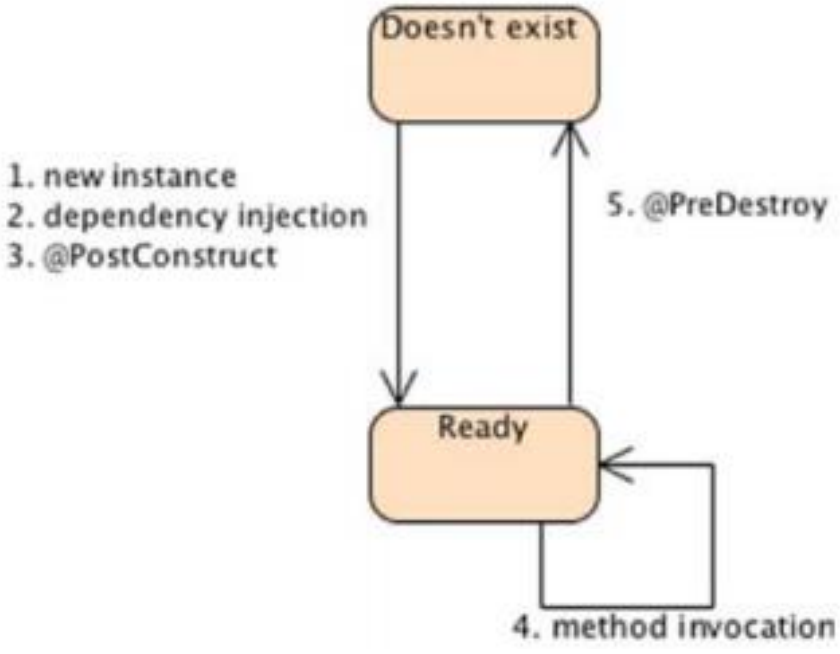
   CDI è costruito sul concetto di “loose coupling, strong typing”, ciò significa che **i bean sono debolmente accoppiati ma viene mantenuta la tipizzazione forte.**

   La dependency injection (DI) è un design patter che disaccoppia componenti dipendenti e fa parte dell’inversione del controllo, ovvero il container assume il controllo del business code e fornisce servizi tecnici gestendo il ciclo di vita dei component e la configurazione dei components.

   La DI è stata introdotto in Java EE 5 consentendo agli sviluppatori di iniettare risorse e utilizzarle concentrandosi solo sulla logica di business ignorando completamente il resto come ad esempio il ciclo di vita stesso delle risorse, il cui compito è delegato al container.

   Per poter abilitare CDI è obbligatorio l’uso del Deployment Descriptor. Questo strumento serve

per identificare i Beans nel class path. Senza di esso CDI non sarà in grado di utilizzare injection, interception, decoration e così via.
Ogni oggetto gestito da CDI ha uno scope ben definito e un ciclo di vita vincolato ad un contesto specifico. Con CDI un bean è leagato a un contesto e rimane in quel contesto finché viene distrutto dal container.
Quando invochiamo un bean il container è responsabile della creazione dell'istanza risolvendo le dipendenze e invocando qualsiasi metodo annotato con @PostCOnstrut prima del richiamo del metodo di business del bean.
Prima che il container elimini il bean, vengono eseguiti i metodi annotati con @PreDestroy.

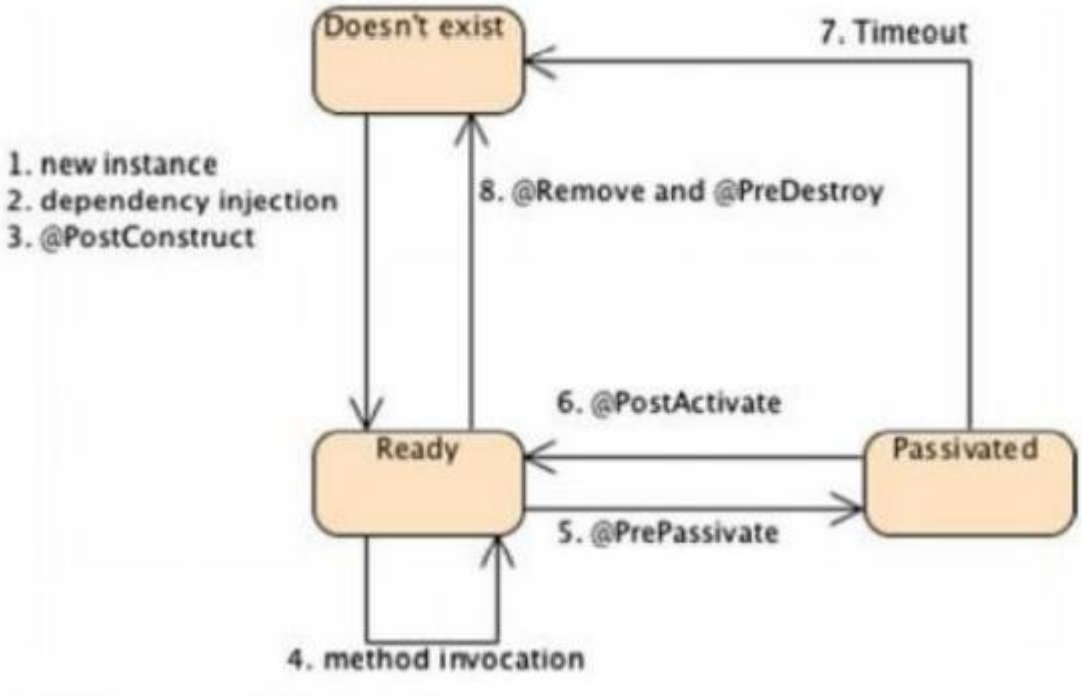


2. **Descrivere il ciclo di vita dei Bean Stateful e come fare per intercettare i cambiamenti di stato.**

   I bean stateful si differenziano dai bean Singleton e Stateless per il mantenimento dello stato di conversazione con il proprio Client e quindi hanno un ciclo di vita diverso. Il container genere un'istanza e la assegna solo a un client, e quindi, ogni richiesta da quel client, viene passata alla stessa istanza. Seguendo questo principio se ci sono 1000 utenti simultanei esisteranno 1000 istanza del bean Stateful. Se un Client non richiama l'istanza del bean, quest'ultimo viene passivato e riattivato quando ne ha bisogno.
   Il ciclo di vita è così descritto:
   1 – Il ciclo di vita di un bean stateful inizia quando un client richiede un riferimento al bean (usando l'injection dependency o la ricerca JNDI). Il container cerca una nuova istanza di bean di sessione e la archivia in memoria.
   2 – Se l'istanza appena creata utilizza dependency injection tramite annotazioni o il deployment descriptor, il container inietta tutte le risorse necessarie.

3 – Se l’istanza ha un metodo annotato con @PostConstruct, il container lo richiama.
4 – Il bean esegue la chiama richiesta e rimane in memoria, in attesa di richieste.
5 – Se il client rimane inattivo per un periodo di tempo, il container richiama il metodo annotato con @PrePassivate, se presente, e passa l’istanza del bean in un’archiviazione permanente.
6 – Se il client richiama un bean passivato, il container lo riattiva in memoria e richiama il metodo annotato con @PostActivate, se presente.
7 – Se il client non richiama un’istanza di bean passiva per il periodo di timeout della sessione, il container lo distrugge.
8 – In alternativa al passaggio 7, se il client chiama un metodo annotato da @Remove, il container richiama quindi il metodo annotato con @PreDestroy, se presente, e termina la vita dell’istanza del bean.
Spesso uno stateful apre risorse come socket o connessioni al db. Poiché il container non può avere le risorse aperte per ogni bean, bisogna chiudere e riaprire le risorse ogni volta che il

bean viene attivato o passivato usando i metodi di callback.



3. Descrivere e confrontare Interceptors e Decorators

Gli interceptors consento di aggiungere cross-cutting concernes ai bean. Quando un client invoca un metodo su un Managed Bean, il container è in grado di intercettare la chiamata e elaborare la business logic prima che il metodo del bean sia invocato. Esistono 4 tipologie di interceptors:

- Interceptor associati a un costruttore della classe Target @AroundConstruct
- Interceptor che si interpone ad uno specifico metodo di business @AroundInvoke
- Interceptor che si interpone sui metodi di timeout @AroundTimeout
- Interceptor che si interpone sul ciclo di vita dell'istanza di destinazione (@PostConstruct, @PreDestroy)

L'idea dei decorator è di prendere una classe e avvolgere intorno ad essa un'altra classe. In questo modo quando chiama una classe decorata prima di arrivare alla classe target si passa per il decoratore. I decorators servono ad aggiungere logica ai metodi di business.

La differenza tra gli interceptor e i decorators, è che i decorator, dovendo implementare l’interfaccia della classe che decorano, devono necessariamente conosce la struttura della classe che decorano, invece, gli interceptor, non necessitano di conoscere la logica di business dei metodi che vengono intercettati.

4. Descrivere le caratteristiche di JMS

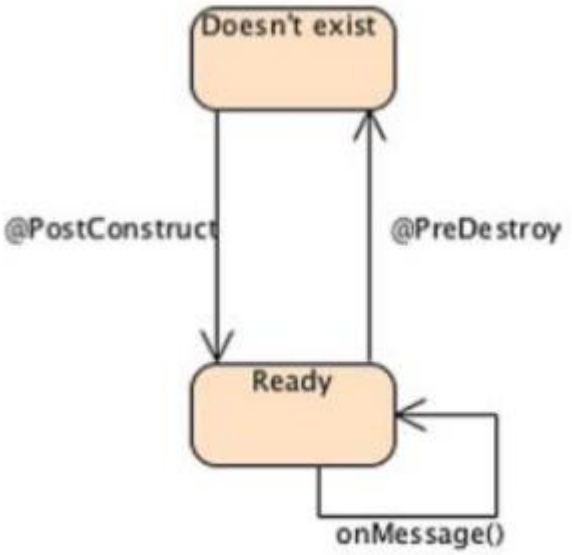
JMS è l'API che consente ad applicazioni Java nella rete di scambiarsi messaggi asincroni.
I messaggi vengono inviati da un Producer ad un Provider che memorizza il messaggio e che si occupa di inoltrarlo ad uno o più Consumer.
Il provider consente la comunicazione asincrona fornendo una destinazione in cui i messaggi possono essere conservati fino a quando non possono essere consegnati ad un Consumer.
Il provider, in altre parole, è colui che gestisce la destinazione dei messaggi che vengono inviati dai Producer. Esistono due diversi tipi di destinazioni:
- Modello point-to-point (P2P), dove la destinazione che viene utilizzata per contenere i messaggi viene chiamata queue, ovvero coda. Quando si usa questo modello, il producer invia un messaggio su questa coda dove viene conservato fino a quando non viene inoltrato ad un consumer. Dopo che il messaggio viene inoltrato al consumer viene rimosso dalla coda.
- Modello publish-subscribe (sub-pub), dove la destinazione è chiamata topic. In questo modello, il producer invia un messaggio su un topic che

conserva il messaggio e lo inoltra a tutti i consumer che sono iscritti a tale topic.
Questo meccanismo è consentito grazie agli administered object, ovvero oggetti configurati amministrativamente, anziché programmaticamente. Vi sono due administer object:
- connection factories, che viene utilizzata dai client per creare una connessione a una destinazione.
- destination, che indica il punto di distribuzione dei messaggi.
Una caratteristica molto importante di JMS è che il middleware orientato ai messaggi (MOM) consente lo scambio di messaggi in modo asincrono. Può essere visto come un buffer tra sistemi che producono e consumano messaggi.
I producer e i consumer non devono essere disponibili contemporaneamente per comunicare e non devono nemmeno conoscersi l'un l'altro, proprio grazie al MOM. L'unica cosa su cui consumer e producer devono essere d'accordo è il formato del messaggio e la destinazione intermedia.

I messaggi sono divisi in 3 parti**:**
- header, che contiene informazioni standard per identificare identificare ed instradare il messaggio.
- proprietà: composto da coppie nome/valore che l'applicazione può settare o leggere. Permettono alle destinazioni di filtrare i messaggi basandosi su questi valori.
- corpo: contiene il messaggio in sé e può avere diversi formati.

Al posto dei JMS client, grazie al container, è possibile utilizzare gli MDB (Message-Driver bean), ovvero un consumer asincrono invocato dal container all'arrivo di un messaggio. Grazie al container, gli MDB possono concentrarsi solo sulla logica di business mentre il container gestisce il multithreading, sicurezza e transaction. L'esecuzione asincrona è nasata sulla gestione di eventi, infatti gli MDB implementano l'interfaccia MessageListener e quindi il metodo onMessage() che servirà al provider per inoltrati i messaggi agli MDB.

Il ciclo di vita degli MDB è:



5. **Descrivere le caratteristiche di una Service Oriented Architecture.**

SOA definisce un modello logico secondo il quale sviluppare il software. Tale modello è realizzato dai Web Service che si presentano come moduli software distribuiti i quali collaborano fornendo determinati servizi in maniera standard.
Le caratteristiche principali sono:
- Software come servizio: al contrario del software tradizionale, un WS può essere consegnato ed utilizzato come un canale di comunicazione accessibile, in modo obliquo, da qualsiasi piattaforma. I WS consentono l'incapsulamento: i componenti possono essere isolati in modo tale che lo strato relativo al servizio vero e proprio sia esposto all'esterno: questo comporta l'indipendenza dall'implementazione e sicurezza del sistema interno.
- Interoperabilità: la logica applicativa incapsulata all'interno dei WS è completamente decentralizzata ed accessibile attraverso Internet da piattaforme, dispositivi e linguaggi di programmazione differenti.

- Semplicità di sviluppo e di rilascio: sviluppare un insieme di WS, intorno ad uno strato di software esistente, è un'operazione semplice che non dovrebbe richiedere cambiamenti nel codice originale dell'applicazione. Lo sviluppo incrementale dei WS avviene in modo semplice e naturale.
- Standard: concetti fondamentali che stanno dietro ai WS sono regolati da specifiche universalmente riconosciute e da standard approvati dalle più grandi ed importanti società d'Information Technology al mondo.
- Nuove opportunità di mercato: offrire servizi attraverso WS consente di catturare nuove opportunità di mercato e nuovi clienti semplicemente promuovendo i propri servizi.