

Programmazione distribuita

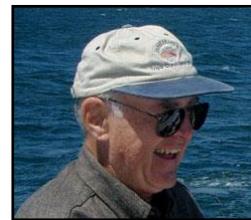
Lezione 1, Prologo ai sistemi distribuiti: Un sistema distribuito è un sistema formato da un insieme di macchine autonome connesse alla rete.

Middleware: è uno strato software che si occupa di connettere le macchine fra di loro in modo da far percepire all'utente il sistema come un'unica entità integrata.

Motivazioni dietro i sistemi distribuiti: esistono motivazioni sia di tipo economico che tecnologico:

- **Economico:** rispondono in maniera precisa alle esigenze ed alle richieste della economia di mercato che è caratterizzata da numerose e frequenti acquisizioni, integrazioni e fusioni di aziende. (il professore fece l'esempio **Fiat-Cruiser**).
- **Tecnologico:** permettono una migliore gestione e integrazione delle tecnologie legacy (si cerca quando possibile di integrarle, perché è inutile cambiarle **Esempio di MediaWorld con il terminale emulato**), permettendo di assecondare i rapidi e repentinamente cambiamenti dell'industria.

Legge di Moore: è una delle leggi più famose dell'informatica (**legge perché per ora funziona, domani non si sa, altrimenti l'avremmo chiamata teorema**), essa afferma che il numero di transistor all'interno di un processore raddoppia ogni due anni, per tenere il passo con la sempre crescente necessità di potenza di calcolo.



Leggi per le reti:

- **Legge di Sarnoff:** il valore di una rete di broadcast è direttamente proporzionale al numero degli utenti connessi, $V = a * N$.
- **Legge di Metcalfe:** il valore di una rete di comunicazione è direttamente proporzionale al quadrato del numero di utenti, $V = a * N + b * N^2$.
- **Legge di Reed:** il valore di una rete sociale è direttamente proporzionale ad una funzione esponenziale in N : $V = a * N + b * N^2 + c * 2^N$.

| Legge: | Sarnoff | Metcalfe | GFN (Reed) |
|--|-------------------------|------------------------|------------------------|
| Transazioni opzionali | Sintonizza trasmissione | Collega colleghi | Unisciti / Crea gruppi |
| Esempi | OnSale, Accesso remoto | Yahoo! Annunci, e-mail | eBay, chat room |
| Valore di N membro netto | N | N^2 | 2^N |
| Valore combinato di reti membro N, M . | $N + M$ | $N^2 + M^2 + 2NM$ | $2^N \times 2^M$ |

Queste leggi portano all'interessante conclusione che le reti sono naturalmente portate ad accorparsi, integrando i servizi in modo da fornirli agli utenti di tutte le reti che si stanno fondendo. Vantaggi per tipo di rete (per legge) (vedi **tabella sopra 4° riga**):

- **Reti di tipo Broadcast (Sarnoff):** per le reti di questo tipo il vantaggio è legato semplicemente all'numero di utenti a cui è possibile vendere il servizio, reti più grandi permettono un bacino di utenza maggiore.
- **Reti di comunicazioni e sociale (Metcalfe e Reed):** per reti di questo tipo il vantaggio invece è molto maggiore, infatti accorpando più reti il numero di comunicazioni che è possibile effettuare aumenta in modo notevole (il professore fece l'**esempio di Tizio e Caio** dividendo l'aula).

Lezione 2, Open Distributed Processing Reference Model:

The Reference Model of Open Distributed Processing (RM-ODP): Con lo scopo di facilitare lo sviluppo dei sistemi distribuiti è stato creato dall'ISO/IEC un modello di riferimento che integra il modello a 7 Layer ISO/OSI, ponendo maggiore attenzione sul problema della **comunicazione** piuttosto che sulla **connessione**. Tale modello punta ad astrarre e standardizzare anche il concetto di **portabilità** e di **trasparenza** all'interno di un sistema distribuito. In questo senso, **RM-ODP** estende ed ingloba, il modello **ISO/OSI**, usando quest'ultimo come modello per la comunicazione tra componenti eterogenee.

Caratteristiche di un sistema distribuito (**Altissima probabilità di domanda**):

- **Remoto:** Le componenti di un sistema distribuito devono poter essere locali o remote, quindi anche potenzialmente localizzate su macchine diverse.
- **Concorrenza:** Un sistema distribuito è per sua stessa natura concorrente, in quanto la contemporanea esecuzione di due (o più) istruzioni è possibile su macchine diverse e **non esistono** strumenti come **lock** e **semafori**.
- **Assenza di uno stato globale:** Non esiste un modo per poter determinare lo stato globale del sistema, in quanto la distanza e l'eterogeneità (**diversi clock**) del sistema non permette di definire con certezza lo stato in cui si trova ciascun nodo.
- **Malfunzionamenti parziali:** Ogni componente di un sistema distribuito può smettere di funzionare in maniera indipendente dalle altre componenti e questo non inficia le funzionalità che sono localizzate su altre macchine.
- **Eterogeneità:** Un sistema distribuito per sua stessa definizione è eterogeneo (formato da parti diverse) sia del punto di vista hardware che software.
- **Autonomia:** Un sistema distribuito non ha un singolo punto dal quale esso puo' essere controllato, coordinato e gestito. Quindi, la collaborazione va ottenuta mediando le richieste del sistema distribuito con quelle del sistema che gestisce ciascun nodo, tramite politiche di condivisione e di accesso, formalmente specificate e rigidamente applicate (questa caratteristica si lega ai **malfunzionamenti parziali**).
- **Evoluzione:** un sistema distribuito può cambiare anche in modo sostanziale durante la sua vita, sia per motivi "**ambientali**" che **tecnologici**. La flessibilità di un sistema distribuito deve assicurare che la migrazione verso ambienti, tecnologie differenti e applicazioni nuove puo' essere effettuata con successo e senza costi eccessivi.
- **Mobilità:** Così come appare naturale che gli utenti siano mobili, altrettanto naturale deve essere la mobilità dei nodi e delle risorse (ad esempio, dati) all'interno del sistema in modo da poter adattare al meglio le prestazioni del sistema (il famoso **Akamai**).

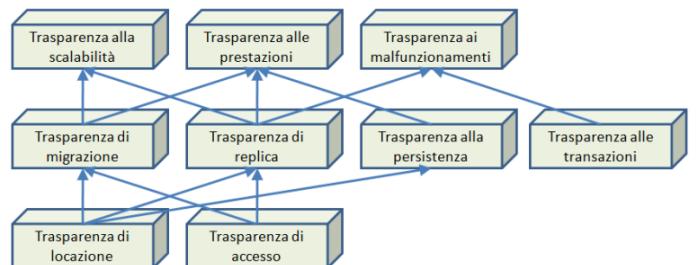
Requisiti non funzionali di un sistema distribuito (Altissima probabilità di domanda**):** Sono dei requisiti che anche se non esprimono una funzione specifica del sistema, contribuiscono a garantire la qualità del sistema stesso. In particolare, i sistemi distribuiti devono essere:

- **Aperti:** in modo da supportare la portabilità di esecuzione e l'interoperabilità (capacità di collaborare insieme tra diverse componenti) attraverso interfacce e servizi ben documentati e aderenti a standard noti e riconosciuti. In modo da permettere **l'evoluzione** e la possibilità di cambiare fornitore senza particolari rischi per la intera architettura.
- **Integrati:** così da incorporare al proprio interno sistemi e risorse differenti senza dover utilizzare strumenti ad-hoc. Questo permette di trattare in maniera efficiente (economica) il problema della eterogeneità hardware e software.
- **Flessibili:** Ovvero essere in grado di evolversi e avere la capacità di integrare sistemi legacy al loro interno.
- **Modulari:** in modo da permettere ad ogni componente di poter essere autonoma e indipendente verso il resto del sistema.
- **Supportino la federazione di sistemi:** in modo da unire diversi sistemi, dal punto di vista amministrativo oltre che architetturale, per lavorare e fornire servizi in maniera congiunta.

- **Facilmente gestibili:** in modo da permettere il controllo, la gestione e la manutenzione per configurarne i servizi, la loro qualità (Quality of Service) e le politiche di accesso.
- **Supporto per la qualità del servizio (Quality of Service):** per poter fornire i servizi con vincoli di tempo, di disponibilità e di affidabilità, anche in presenza di malfunzionamenti parziali.
- **Scalabili:** perché qualsiasi sistema distribuito accessibile da Internet puo` essere soggetto a picchi di carico non prevedibili e deve essere in grado di gestirli.
- **Sicuri:** utenti non autorizzati non possono accedere a dati sensibili.
- **Trasparenti:** mascherando i dettagli e le differenze della architettura sottostante che assicura la distribuzione dei servizi sulle componenti del sistema.

La trasparenza di un sistema distribuito: come abbiamo visto una delle principali caratteristiche dei sistemi distribuiti è la trasparenza, essa in modo analogo a come viene intesa in altri ambiti informatici permette di nascondere dettagli implementativi in modo da **semplificare le operazioni di sviluppo**. Nonché permettere un **maggior riuso delle applicazioni sviluppate**.

Tale operazione viene fatta attraverso il **Middleware**.



La trasparenza all'interno di un sistema distribuito viene fornito attraverso vari livelli tra loro interconnessi:

- **Livello Base:**
 - **Trasparenza di Accesso:** nasconde le differenze nella rappresentazione dei dati e nel meccanismo di invocazione per permettere la interoperabilità tra oggetti. Gli oggetti devono essere accessibili attraverso la stessa interfaccia, sia che siano acceduti da locale sia che siano acceduti da remoto.
 - **Trasparenza di Locazione:** non permette di utilizzare informazioni circa la localizzazione nel sistema (**IP fisico**) di una particolare componente, che viene identificata ed utilizzata in maniera indipendente dalla sua posizione. Questo tipo di distribuzione fornisce una vista logica del sistema di naming, in modo da disaccoppiare il nome da una posizione all'interno della rete (praticamente il **DNS**).
- **Livello di Funzionalità:**
 - **Trasparenza di Migrazione (locazione/accesso):** Nasconde la possibilità del sistema di spostare oggetti o dati da un nodo all'altro della rete, continuando ad essere raggiungibile e utilizzabile da altri oggetti. In modo da ottimizzare le prestazioni (**AKAMI** e le **CDL Content Delivery Network**).
 - **Trasparenza di Replica:** Con questo tipo di trasparenza, il sistema maschera il fatto che una singola componente viene replicata in un certo numero di copie (dette **repliche**) che vengono posizionate su altri nodi del sistema, e che offrono esattamente lo stesso tipo di servizio della componente originale. Le repliche vengono utilizzate per ragioni di:
 - **Prestazioni:** facendo in modo di replicare componenti laddove (all'interno del sistema) maggiori sono le richieste per quel tipo di servizi, in modo da minimizzare la latenza per accedervi.
 - **Scalabilità:** il sistema in presenza di aumento del carico di lavoro.
 - **Trasparenza alla Persistenza (Trasparenza/Locazione):** Questo tipo di trasparenza scherma l'utente dalle operazioni che compie il sistema per rendere persistente (cioè in memoria secondaria) un oggetto durante una fase di non utilizzo.
 - **Trasparenza alle Transazioni:** nasconde all'utente le attività di coordinamento che vengono svolte per assicurare la consistenza dello ` stato degli oggetti in presenza della concorrenza e nella gestione delle transazioni (**ACID**).
- **Livello di Efficienza:** Scalabilità, prestazioni e malfunzionamenti.

- **Trasparenza di Scalabilità (migrazione/replica):** assicura l'operabilità del sistema in caso di picchi di carico, senza dover modificare l'architettura e l'organizzazione. (differenza tra scalabilità **verticale** e **orizzontale**). Tale trasparenza è basata sulla trasparenza di replica e migrazione.
- **Trasparenza alle Prestazioni:** Progettisti/sviluppatori ottengono alte prestazioni senza conoscere i meccanismi utilizzati. Tale trasparenza viene ottenuta tramite il:
 - **Bilanciamento del carico:** su più macchine (**Migrazione/Replica**);
 - **Minimizzazione della latenza:** tramite sistemi come Akamai (**Migrazione/Replica**);
 - **Ottimizzazione delle risorse:** spostando risorse tra le varie memorie e ottimizzandoli (**Persistenza**)
- **Trasparenza ai malfunzionamenti:** in presenza di malfunzionamenti parziali il resto del sistema continua a funzionare, tale trasparenza è basata sulla **trasparenza di replica** ma anche sulla **Trasparenza alle transazioni** (facendo il **rollback** di eventuali transazioni iniziate ma non finite).

L'eccesso di trasparenza può portare a degli errori progettuali, quindi non tutti i modelli la utilizzano al 100%. Inoltre, implementare ogni tipo di trasparenza ha un costo non indifferente, per questo prima di procedere si procede ad un'attenta **analisi costi/benefici**. Anche se alcuni tipi di trasparenza (**Locazione ed Accesso**) sono ritenuti fondamentali e pertanto vengono sempre implementati.

Senza lezione specifica:

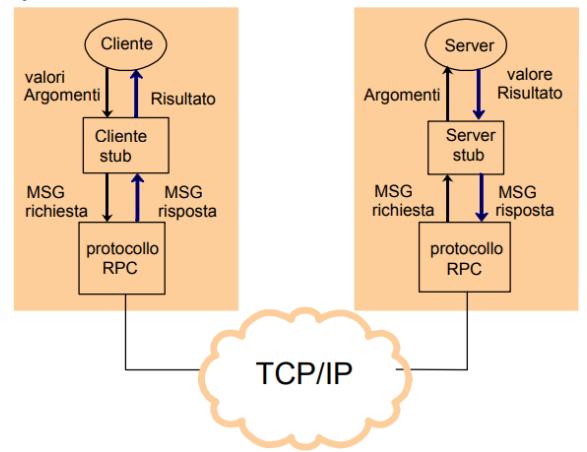
Il Middleware: (**Lo chiede in modo veloce XD**) nato intorno agli anni 60/70 per offrire soluzioni, scalabili ed eterogenee. Serve a integrare sistemi eterogeni esso è uno strato software che si trova tra applicazioni ed hardware. Esistono tre principali strati del Middleware:

- **Middleware di infrastruttura:** si occupa di comunicazione tra OS diversi e gestione di concorrenza in maniere da essere portabile (**per esempio la JVM**).
- **Middleware di distribuzione:** automatizza compiti comuni per la comunicazione come:
 - **Marshalling:** invio di parametri per l'invocazione di servizi in remote.
 - **Multiplexing:** utilizzo dello stesso canale per più invocazioni/richieste.
 - **Gestione della semantica delle invocazioni:** unicast, multicast, attivazione on-demand (invocazioni a più oggetti).
 - **Riconoscimento e gestione dei malfunzionamenti di rete.**
- **Middleware per servizi comuni:** Implementa in automatico servizi comuni, come persistenza, transazioni, sicurezza, accesso, etc... (**accesso tramite Google per esempio**).

La funzione principale del middleware è quella di nascondere dettagli inutili per la progettazione effettiva del software in modo che il programmatore possa concentrarsi solo sull'effettivo sviluppo del programma. La scelta del middleware è molto importante, perché si deve scegliere qualcosa che duri nel tempo e che sia usato, affidabile e supportato (**Scarano domina anche in questo**).

L'evoluzione del Middleware: deriva dal **Remote Procedure Call (RPC)**, creato dalla **Sun** negli anni 80, con l'obiettivo di invocare una procedura in esecuzione su un'altra macchina. Utilizza il paradigma di **sincronia dell'invocazione forzata** ovvero il client veniva bloccato fino a quando il server non rispondeva alla richiesta.

L'RPC si basa sull'implementazione del **Client Stub** e **Server Stub** che si occupano di forzare il Marshalling per risolvere problemi di incompatibilità. Tali parti di codice, con il passare degli anni, vennero automatizzate attraverso un linguaggio specifico creato ad-hoc chiamato **Interface Definition**



Language (IDL). Tale sistema viene ancora oggi utilizzato per parti fondamentali di internet come il **DHCP** e il **DNS**.

Passaggio tra RPC e Modello ad Oggetti distribuiti: All'inizio degli anni 90 il modello RPC viene esteso in maniera de permettere l'invocazione di oggetti in remoto, questo ha portato a una vera e propria trasformazione del modello e all'integrazione al suo interno del paradigma della programmazione ad oggetti.

Common Object Request Broker Application (Corba): (**Lo chiede**) sviluppato nel 1991 è la prima proposta significativa di ambiente di programmazione basato su sistemi distribuiti. Si basa su servizi offerti dall'**Object Request Broker (ORB)** che astrae il meccanismo di invocazione in modo completamente trasparente al client. Con il passare degli anni a causa di problemi iniziali di progettazioni dovuti a un'implementazione di tipo **reference** (tutti i produttori seduti al tavolo che facevano ognuno i propri interessi) Corba inizio a mostrare alcune mancanze come l'assenza di **interoperabilità** tra ORB diverse (ogni produttore modificava la sua).

Microsof.NET: sulla quale sono basati tutti gli operativi Microsoft, si basa sul **Common Language Runtime (CLR)** linguaggio che permette l'integrazione di tutti i linguaggi supportati da Microsoft. Utilizza una macchina virtuale per risolvere un problema di eterogeneità livello software.

Enterprise Java: nasce con l'obiettivo di limitare la complessità è basato su un modello a componenti, modello che viene gestito da una componente server si chiama **Container** (**risposta al 90% delle domande XD**).

<<Da grandi poteri derivano grandi responsabilità>>

Riferito a Corba che voleva fare tutto...

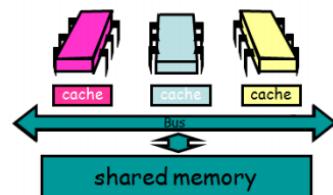
Concetto di middleware implicito: si basa su corba, i **primi middleware erano esplicativi** (bisognava invocare qualcosa per connettersi), questo portava a dei problemi come: la complessità, mancanza di controllo sugli errori, mancanza di portabilità, (il prof sta facendo l'esempio di JDBC dove dobbiamo ogni volta scrivere tutto). Nel MW implicito servizi vengono forniti automaticamente dal sistema sulla base delle richieste specificate durante il **deployment**, (esempio della banca, usare codice senza conoscerlo) praticamente dico al MW quali servizi voglio tramite un file **XML** e lui automaticamente li implementa.

Lezione 3, Programmazione concorrente e Thread in Java: il nostro sistema è formato da tante componenti diverse, questo lo rende implicitamente concorrente.

Motivazioni: La motivazione principale dietro la nascita dei **Thread** è l'effetto di **Thermal Noise**, tale effetto disturba la crescita secondo la legge di **Moore** del numero di transistor. Praticamente:

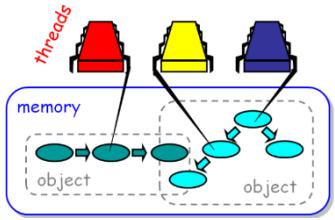
Non si possono avere "tanti" transistor su un processore, che siano anche "facili da raffreddare" e che siano "veloci": si deve rinunciare ad una di queste caratteristiche.

Questo ha portato a un radicale cambiamento per quanto riguarda la progettazione di CPU, passando da un'architettura a **Single Core** ad una a **Multi Core**.



Questo cambiamento ha cambiato il modo in cui i programmati intendessero il loro lavoro, infatti fino a poco tempo fa i miglioramenti della tecnologia comportavano un automatico miglioramento delle prestazioni software (CPU con clock maggiore eseguivano il codice con più velocità). Oggi invece un programma per essere più efficiente deve essere in grado di sfruttare al massimo il **Parallelismo delle applicazioni**.

Le sfide: questo nuovo paradigma di programmazione ci porta inevitabilmente a scontrarci con sfide di natura “gestionale” in quanto avendo più processori che accedono e lavorano contemporaneamente con una singola memoria condivisa si vanno a creare tutta una serie di problemi di concorrenza e di sincronia.



Cosa ci riserva il futuro: Architettura **master/slave** IBM, un core potente e tanti piccoli “schiavi” che eseguono operazioni più semplici.

Diversi tipo di programmazione concorrente: tipo di programmazione che coinvolge diversi processi che vengono eseguiti concorrentemente:

- Eseguita su calcolatori diversi
- Processi concorrenti sulla stessa macchina (**multitasking**)
- Programmazione concorrente nello stesso processo (“**processi lightweight**” all’interno del processo: **Thread**): strumento efficace che permette di costruire programmi che eseguono più operazioni contemporaneamente.

Apache: è server molto famoso, nonché un ottimo esempio di applicazione **Multithread**. Per aumentare le performance Apache all’avvio instanzia un certo numero di thread che tiene in uno stato di **sospeso** pronti ad essere risvegliati ed usati.

I Thread: (**Lo chiede in modo veloce XD**) I thread sono processi all’interno dei processi, vengono anche detti “**processi leggeri**” e offrono diversi vantaggi, essi sono infatti più facili e veloci da creare rispetto al processo che li “ospita” e condividono con quest’ultimo risorse e memoria, cosa che rende estremamente efficiente la comunicazione a discapito di alcuni problemi di concorrenza.

All’interno di java i un thread un metodo può essere creato in due modi:

- **Instanziando l’oggetto thread:** soluzione facile solo che ci vincola a non poter estendere altre classi, visto che java non permette l’ereditarietà multipla ed è necessario estendere la classe Thread.
- **Estrarre la gestione, passando un task ad un executor:** tramite l’interfaccia Runnable (metodo consigliato).

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

Alcuni metodi della classe Thread:

.sleep(Int t): Sospende (stato di **Wait**) l’esecuzione di un thread per un tot di millisecondi, Se il thread in pausa viene interrotto viene lanciata l’eccezione **InterruptedException**.

.interrupt(): Se lo stato del thread è su **wait**, questo metodo invia un **InterruptedException**, se invece il thread si trova in uno stato diverso da wait viene interrotto e il flag **interrupt** viene settato a vero.

.Interrupted(): controlla lo stato del flag **interrupt** è utili per gestire le eccezioni.

.join(int t): Obbliga gli altri thread ad attendere il completamento del thread che invoca tale metodo prima di continuare. E inoltre possibile passare un tempo massimo in millisecondi di attesa, prima che gli altri thread continuino la propria esecuzione.

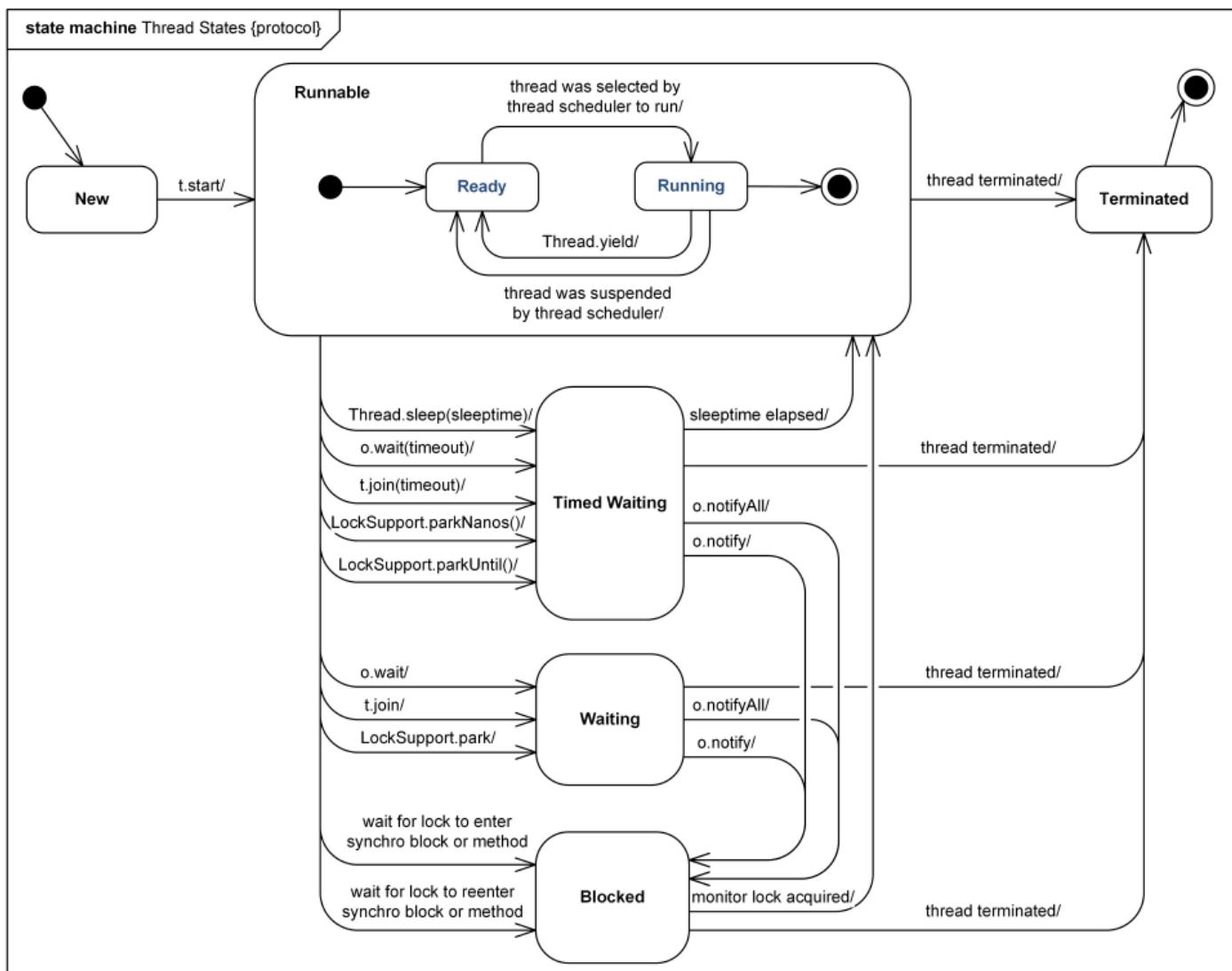
.inAlive(): Controlla che un thread sia ancora in esecuzione.

.getState(): Restituisce lo stato di un thread (runnable, terminated, ecc....).

.yield(): Cambia lo stato di un thread da **run** a **ready**.

.start() e .run(): Start porta il thread in uno stato di ready, run invece contiene il codice che il thread dovrà eseguire una volta nello stato di running.

Diagramma degli stati di un thread:



Problemi dei thread: i thread basano la loro strategia di comunicazione sulla condivisione della memoria, questo tipo di tecnica è estremamente efficiente ma genera alcuni problemi come:

- **L'interferenza fra thread:** si ha quando più thread lavorano su una stessa variabile ed eseguendo le loro operazioni contemporaneamente si interfogliono.

1. Thread A: Retrieve c.
2. Thread B: Retrieve c.
3. Thread A: Increment retrieved value; result is 1.
4. Thread B: Decrement retrieved value; result is -1.
5. Thread A: Store result in c; c is now 1.
6. Thread B: Store result in c; c is now -1.

```

class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }
}
  
```

- **Inconsistenza della memoria:** si ha quando diversi thread hanno visioni diverse dei dati presenti in memoria. Tale problema può essere risolto creando una relazione **happens-before** (succede-prima, anche questo si ottiene con la **sincronizzazione**, per esempio nell'esempio affianco si fa una **join** sul thread A e quindi il thread B verrà eseguito dopo A **happens-before**).

- Un esempio: se un (1, one) thread esegue:

```

int counter = 0;
// ...
counter++;
// ...
System.out.println(counter);
  
```

- Si stampa 1...e con due thread A e B?
- Se A esegue counter++ e poi dopo B esegue la stampa ...
- ...può capitare che la modifica di A sia visibile a B (che stampa 0)

Entrambi questi problemi possono essere risolti tramite la **sincronizzazione** questo tipo di soluzione porta a sua volta a problemi di **Race Condition** (più thread cercano di accedere alla stessa risorsa in maniera concorrente, questo tipo di problema è molto complesso da individuare perché è molto difficile riprodurre tali situazioni tramite il Debugger, molto simile al **Principio di Heisenberg**) questo a sua volta porta a problemi di **Liveness** come il:

<< Non è possibile misurare simultaneamente la posizione ed il momento (quantità di moto, cioè massa per velocità) di una particella >>

- **Deadlock:** Quando due thread si attendono a vicenda per un tempo infinito senza fare nulla. (una porta due persone che vogliono passare). Questo problema in Java si risolve tramite l'uso del metodo **bow** (inchino),

```
// ...
public static void main(String[] args) {
    final Friend alphonse = new Friend("Alph");
    final Friend gaston = new Friend("Gas");

    new Thread(new Runnable() {
        public void run() { alphonse.bow(gaston); }
    }).start();

    new Thread(new Runnable() {
        public void run() { gaston.bow(alphonse); }
    }).start();
} // end main
```

- Si creano Alphonse e Gaston
- Classe anonima di tipo Runnable passata al costruttore di Thread
- con il metodo da eseguire ...
- ... e si lancia



- **Livelock:** Quando due thread si attendono a vicenda compiendo in continuazione azioni che non portano a nulla. (Destra sinistra tra due persone per strada)
- **Starvation:** Quando un thread in coda attende per un tempo infinito il suo turno, senza che gli venga mai concesso.

Keyword volatile: è una keyword utilizzata per rendere l'accesso ad una variabile asincrono in modo da risolvere il problema dell'inconsistenza della memoria (**risolvono solo il problema della memoria**) (**operazione atomica**).

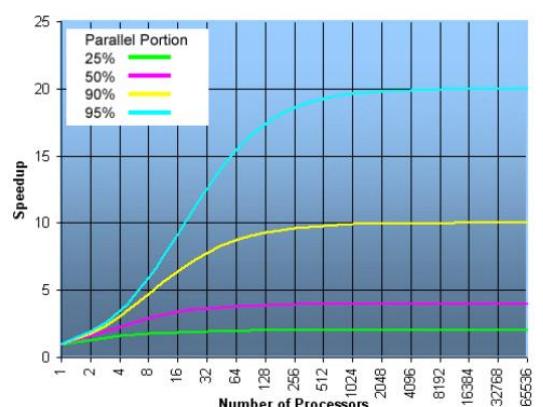
Lezione 4, Programmazione concorrente e Thread parte 2:

La legge di Amdahl: è una legge sull'utilità del calcolo concorrente, definisce il vantaggio che si ha se il programma è sviluppato ad-hoc per sfruttare la programmazione multithread. (esempio dei 5 amici, che cooperano per dipingere una stanza e si devono organizzare per ottimizzare il tempo),

Lo Speedrun: è il tempo impiegato per svolgere un'operazione **X** con **N** processori dove **P** è la parte del programma che è possibile parallelizzare.

$$S = \frac{1}{1 - P + \frac{P}{N}}$$

Questa legge ci dice che la parte sequenziale di un programma rallenta molto qualsiasi Speedrun che speriamo di ottenere. Quindi è inutile spendere molti soldi in processori, conviene invece rendere preponderante la parte del programma parallelizzabile.



Gli strumenti per la sincronizzazione dei thread: questi strumenti sono un trade-off (o uno o l'altro) tra:

- **Facilità d'uso:** non tutti sono semplici da usare;
- **Efficienza:** possono creare colli di bottiglia e rallentare di molto l'esecuzione.

Metodi sincronizzati: Permettono di risolvere facilmente gli errori di concorrenza **sacrificando efficienza**. Si usa la keyword **synchronized** in questo modo l'esecuzione della porzione di codice successiva viene eseguita in mutua esclusione. Quando un thread si trova all'interno di una porzione di codice sincronizzata gli altri thread rimangono sospesi fino a quando il primo non ha completato. Questo crea una relazione **happens-before** e quindi risolve anche il problema **d'inconsistenza della memoria**. (I costruttori non posso essere sincronizzati, perché non servono).

```
public class SynchronizedCounter {  
    private int c = 0;  
    public synchronized void increment() {  
        c++;  
    }  
    public synchronized void decrement() {  
        c--;  
    }  
    public synchronized int value() {  
        return c;  
    }  
}
```

Problema del leaking: Nasce dal fatto che non si può sincronizzare il costruttore. Praticamente usiamo il riferimento all'oggetto prima che quest'ultimo sia effettivamente costruito.

Lock Intrisici: è un'entità associata ad ogni oggetto, garantisce sia la **mutua esclusione** che il **happens-before**. Praticamente il thread prende il lock dell'oggetto che deve usare, lo usa e poi lo rilascia. Si usa **synchronized** su piccole porzioni di codice. Questo tipo di sincronizzazione viene detta "**a grana fine**". (**Amdahl è felice**)

```
public void addName(String name) {  
    synchronized(this) {  
        lastName = name;  
        nameCount++;  
    }  
    nameList.add(name);  
}
```

Sincronizzazione dei metodi statici: Un metodo statico **synchronized** previene la esecuzione interfogliata di tutti gli altri metodi statici sincronizzati. In pratica, si acquisisce il lock dell'oggetto **ClassName.class**

```
public static void foo() {  
    synchronized (ClassName.class) {  
        // Body  
    }  
}
```

Attenzione: Metodi sincronizzati statici garantiscono accesso in mutua esclusione a metodi sincronizzati statici, mentre metodi sincronizzati di istanza garantiscono accesso in mutua esclusione ai metodi sincronizzati di quella istanza.

Metodi atomici: i processori offrono la possibilità di eseguire operazioni non interrompibili (o si completano del tutto o non si fa nulla). In java la cosa si implementa tramite l'uso di variabili **volatile**.

The diagram shows a Java code snippet for an `AtomicCounter` class using `AtomicInteger`. Red arrows point from specific code elements to the right, where they are annotated with terms. The annotations are:

- import java.util.concurrent.atomic.AtomicInteger; → Package
- Class AtomicCounter { → Classe
- private AtomicInteger c = new AtomicInteger(0); → Variabile istanza
- public void increment() { → Metodo non sincronizzato
- c.incrementAndGet(); → Uso di metodi atomici
- }
- public void decrement() { → Uso di metodi atomici
- c.decrementAndGet(); → Lettura
- }
- public int value() { → Lettura
- return c.get(); → Lettura
- }

```
import java.util.concurrent.atomic.AtomicInteger;  
  
class AtomicCounter {  
    private AtomicInteger c = new AtomicInteger(0);  
  
    public void increment() {  
        c.incrementAndGet();  
    }  
  
    public void decrement() {  
        c.decrementAndGet();  
    }  
  
    public int value() {  
        return c.get();  
    }  
}
```

Lezione 5, Singleton e DoubleChecked Locking:

I **Singleton**: (**Lo chiede in modo veloce XD**) è un pattern (i pattern piacciono a **Scarano** 😊) utilizzato per limitare le instanziazioni possibili di una classe ad una sola, in modo certo senza affidare questa limitazione al programmatore. Molto utile quando si devono utilizzare delle risorse I/O (stampanti, gestori di file, ecc...). L'utilizzo dei **singleton** non è sempre necessario, in quanto java instanzia un oggetto solo quando viene usato la prima volta (**Lazy Allocation**). la situazione si complica invece con i thread dove potrebbero esserci problemi di **interleaving** e quindi di **leaking**, che potrebbero portare alla creazione di due istanze. Per risolvere questo problema esistono due soluzioni possibili:

- **Usare le synchronized:** questo metodo funziona solo in parte, infatti se andiamo a sincronizza l'intero metodo **getInstance** risolviamo il problema ma ne perdiamo in efficienza, se invece andiamo ad effettuare una sincronizzazione più precisa vediamo che l'algoritmo fallisce.
- **Double e-checking lock:** Un'altra soluzione sta nell'effettuare due sulla variabile di instanzia per vedere se è stata inizializzata anche se così non è detto che funzioni al 100% in java perché, la creazione del singleton può non essere ancora stata completata prima che un altro thread utilizzi il riferimento (problema causato dalla visibilità della memoria il famoso **leaking**).

```
class Singleton {  
    private static Singleton instance;  
    private Singleton() {}  
    // ...  
    public static Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

E se ci sono più thread?

Il loro interleaving può creare errori:
“più” singleton

PERCHÉ NON FUNZIONA ...

```
class Singleton {  
    private static Singleton instance;  
    private Singleton() {}  
    // ...  
    public static Singleton getInstance() {  
        if (instance == null) {  
            synchronized (Singleton.class) {  
                instance = new Singleton();  
            }  
        }  
        return instance;  
    }  
}
```



PERCHÉ NON FUNZIONA ...

```
public static Singleton getInstance() {  
    if (instance == null) {  
        synchronized (Singleton.class) {  
            if (instance == null)  
                instance = new Singleton();  
        }  
    }  
    return instance;  
}
```



- se due thread A e B arrivano ad eseguire “insieme” qui
- A entra in sezione critica
- crea una istanza
- esce dalla sezione critica
- e restituisce il “primo” singleton
- ma a questo punto B entra e crea un “nuovo” singleton
- che sovrascrive il
- La chiamata al costruttore “sembra” essere prima dell’assegnazione di `instance`
- ma questo non è detto che accada
- Un Singleton non inizializzato potrebbe essere assegnato a `instance`
- e se un altro thread controlla il valore ...
- poi lo può usare (in maniera scorretta)

```
public class Something {  
    private Something() {}  
  
    private static class LazyHolder {  
        private static final Something INSTANCE = new Something();  
    }  
  
    public static Something getInstance() {  
        return LazyHolder.INSTANCE;  
    }  
}
```

La soluzione definitiva sta nel rendere la variabile **Instance, volatile**. Altrimenti, si possono usare le classi statiche con l’idioma **“Initialization-on-demand holder”** che essendo un metodo statico, viene eseguito una sola volta (al caricamento) e stabilisce una relazione **happens-before**.

Lezione 7, Socket TCP: Nel paradigma della programmazione ad oggetti la computazione avviene tramite un insieme di oggetti che mantengono degli stati precisi ed espongono i loro comportamenti (**metodi**). Questo tipo di comunicazione avviene di solito in modo locale, mentre per la programmazione distribuita abbiamo la necessità che tutto questo si svolga tra oggetti che non risiedono sulla stessa macchina **Java Remote Method Invocation**.

I **Socket** sono degli **endpoint** di una comunicazione bidirezionale sulla rete che unisce due programmi. Ad ogni socket viene associato un **indirizzo IP** (identifica la macchina) e un **numero di porta** che identifica il processo (programma) a cui il socket fa riferimento. All'interno del socket le due macchine coinvolte vengono dette **client** e **server**. Il procedimento di comunicazione prevede che:

1. Il **client richieda** la comunicazione al server (di cui conosce indirizzo e porta).
2. Il **server accetti** la comunicazione e instanzi un **socket di comunicazione** (sotto forma di thread) sulla porta fornitegli dal client.

Le **API** java implementano due principali tipi di socket:

- **Socket**: socket standard per comunicazioni.
- **Server-socket**: svolge il ruolo appunto di server, Instanziando un socket privato per ogni nuova richiesta da parte di un client.

Serializzazione: è un processo per salvare un oggetto in un supporto di memorizzazione lineare (ad esempio, un file o un'area di memoria), o per trasmetterlo su una connessione di rete. La serializzazione può essere in forma binaria o può utilizzare codifiche testuali (ad esempio il formato XML) direttamente leggibili dagli esseri umani. Lo scopo della serializzazione è di trasmettere l'intero stato dell'oggetto in modo che esso possa essere successivamente ricreato nello stesso identico stato dal processo inverso, chiamato **deserializzazione**.

- **private static final long serialVersionUID = -414713378645982122L**: l'algoritmo di serializzazione usa questo numero per verificare che la classe appena caricata corrisponde ad un oggetto serializzato, è **buona norma utilizzare questa variabile sempre**.

Gli Stream: la comunicazione tra client e server avviene tramite degli **stream** o flussi attraverso un meccanismo detto di **serializzazione**. Java offre una vasta gamma di interfacce e di classi per gestire i flussi.

Metodi della classe socket:

- **accept()**: invocato su oggetti di tipo **ServerSocket** è un metodo bloccante, l'esecuzione si blocca fino a quando non si collega un client, a questo punto il metodo restituisce un **socket** che non è altro del socket di comunicazione (classe **Socket**).
- **close()**: chiude la connessione e il socket.
- **setSocketTimeout()**: se entro un certo numero di millisecondo il server non riceve **accept** chiude le comunicazioni.

Lezione 8, Socket TCP 2 (with Curly):

Conversione da un oggetto locale ad un oggetto remoto: per far ciò si aggiunge uno strato software composto da due componenti:

- **Stub**: è un oggetto che si trova nel client è che svolge il compito di rappresentare il server per quest'ultimo, esso infatti espone gli stessi metodi implementati dal server in modo che il client li possa usare.
- **Skeleton (o client stub)**: è un oggetto che si trova nel server con lo scopo di comunicare a quest'ultimo le richieste di metodi che gli giungono dallo stub e di ritrasmettere i valori di ritorno provenienti dal server a quest'ultimo in modo che lo stub li possa a sua volta trasmettere al client.

Per garantire allo stub l'accesso ai metodi del server, sia **stub** che **skeleton**, implementano un'**interfaccia remota**.

Problema dell'indirizzamento: nell'esempio fornito dal prof sul libro, non è possibile indirizzare un oggetto impiegato in modo diretto, infatti l'indirizzo del server dove è presente l'oggetto che ci interessa viene passato da linea di comando. Per risolvere il problema viene implementato un servizio di naming, ad un indirizzo ben noto, che si occupa di mantenere l'associazione tra l'ID dell'oggetto e l'indirizzo verso il quale deve essere aperto lo skeleton.

Commenti importanti sull'esempio: l'esempio presenta numeri problemi/mancanze:

- **Nessun controllo sulle versioni:** la classe impiegato potrebbe avere una versione più recente su un lato o su un altro e quindi la comunicazione potrebbe avvenire in modo errato. (Per esempio, se è stato inserito un nuovo metodo);
- **Nessun controllo sugli errori e sulle eccezioni:** se un lato le genera l'altro deve stare pronto a riceverle.
- **Il protocollo non tratta casi di metodi con lo stesso nome ma con firme diverse;**
- **Mancata gestione di chiamate multiple;**
- **Servizio di naming semplicistico e primitivo;**

Funzionamento di getOutputStream(): detto in parole povere bisogna prima instanziare l'oggetto **OutputStream** e poi l'oggetto **InputStream()** con un **flush()** (Ogni byte bufferizzato è stato inviato) per mezzo, il motivo è complesso XD.

[L'esempio completo e spiegato, si trova al Cap. 2, par. 2.3, 2.4, 2.5, 2.6., 2.7.](#)

Lezione 10/11, java RMI:

Java Remote Method Invocation: è una libreria di integrazione java creata da Jim Waldo (ex esperienze con RPC), tale libreria offre la possibilità di invocare metodi appartenenti ad un oggetto remoto (non presente nella stessa JVM, i cui metodi sono esposti e utilizzabili tramite un'interfaccia), con un'architettura di solito basata su un **client** e un **server** (non è obbligatorio si può anche utilizzare un'architettura **peer to peer**). Il progetto nacque con due obiettivi principali:

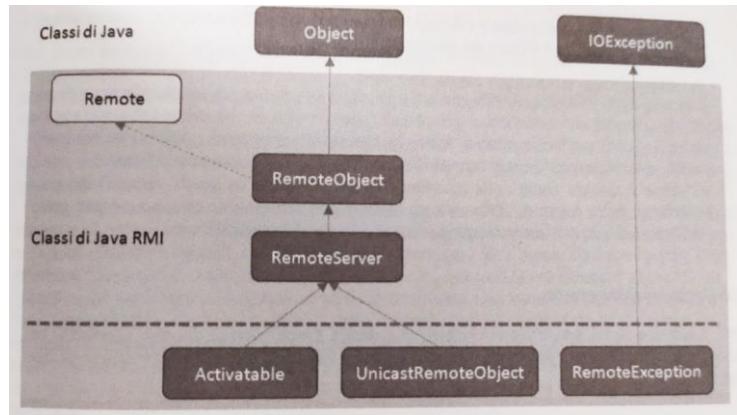
- **Semplicità nella progettazione:** in modo da ridurre errori e abusi, favorendone allo stesso tempo la diffusione.
- **Integrazione con il linguaggio:** in modo da far sentire i programmatori a loro agio in un ambiente già conosciuto. Tale integrazione arriva al punto da rendere disponibile un **Garbage collector distribuito** (evita **memory-leak** e **grafi a dente di sega**), in modo da rendere autonoma la gestione della memoria anche quando abbiamo a che fare con programmi distribuiti.

Altri obiettivi:

- **Invocazione trasparente di metodi remoti:** con lo scopo di offrire un meccanismo semplice e "trasparente" per invocare metodi remoti.
- **Non-trasparenza della natura locale/remota di un oggetto:** anche se questo obiettivo va contro il concetto di "trasparenza" è importante rendere visibile al programmatore la differenza tra oggetto remoto e oggetto locale.
- **Rendere minima la complessità del client e server:** in modo compatibile con gli altri obiettivi.
- **Preservare la sicurezza fornita da java:** Una delle sue caratteristiche principali sin dalla sua pubblicazione.
- **Modalità di invocazione:** deve prevedere che esistano diversi tipi di invocazioni, fornendo sia quello di tipo **unicast** che quella di tipo **multicast**. Deve inoltre permettere che l'oggetto server sia attivato anche solo al momento dell'invocazione.
- **Livelli di trasporto multipli:** deve essere aperto a modifiche future del protocollo di trasporto basato sui **socket**.

Java RMI è contenuto in 5 package:

- **Java.rmi e java.rmi.server**: meccanismi basilari per le invocazioni remote;
- **Java.rmi.activation**: per oggetti attivabili;
- **Java.rmi.dgc**: per il **Distributed Garbage Collection**;
- **Java.rmi.registry**: per il servizio di localizzazione;



Interfacce ed eccezioni remote: prima di definire un oggetto remoto è necessario creare un'interfaccia **remota** che estenda i metodi dello stesso, tale interfaccia deve estendere (implementare) l'interfaccia **java.rmi.Remote** che è un'interfaccia **Marker** ovvero vuota, che serve solo a segnalare che l'interfaccia che stiamo andando a creare, definisce metodi accessibili da remoto. Ogni metodo che andiamo a dichiarare in un'interfaccia remota, deve a sua volta essere un metodo remoto e deve quindi soddisfare delle caratteristiche:

- **Dichiarare l'eccezione java.rmi.RemoteException:** è una **Checked-Exception** (viene controllata dal compilatore e vista come un errore se non c'è). Questo con lo scopo di rendere il programmatore cosciente che si sta andando a lavorare con un oggetto remoto che ha tutta una serie di **problematiche diverse** rispetto ad un oggetto locale.
- **Parametri dichiarati attraverso la propria interfaccia remota.**

Riassumendo l'interfaccia remota definisce un livello ulteriore di accessibilità (oltre a public, protected, etc...), i metodi remoti infatti, sono più accessibili dei metodi public.

Implementazioni remote, si trova al Cap. 3, par. 3.1

Meccanismo di invocazione remota:

Riferimenti remoti: nel modello distribuito il client interagisce con il server tramite uno **stub** che rappresenta l'interfaccia remota, dell'oggetto remoto in locale. Dal punto di vista della **JVM** infatti il tipo dello stub è uguale al tipo del server. Quindi un client può accedere tradizionalmente al tipo di un oggetto remoto, controllando quale interfaccia remota implementa, attraverso **instanceOf**.

Localizzazione e invocazione di oggetti remoti: per poter invocare il metodo remoto, il client deve avere a disposizione il riferimento remoto. Questo può essere reperito in due maniere:

- Ottenendo il riferimento come risultato di altre invocazioni (remote o locali) di metodi;**
- Attraverso un servizio di directory:** ovvero tramite un **name-server** che Java RMI fornisce nella classe **java.rmi.Naming**, tale classe permette di gestire riferimenti remoti ad oggetti tramite un **ID** (stringa) oltre a fornire metodi per:
 - Ricerca:** `lookup();`
 - Registrare:** `bind(), unbind(), rebind();`
 - Elencare:** `list();` gli identificativi registrati e contenuti all'interno del name-server.

Come abbiamo visto in precedenza l'invocazione dei metodi remoti avviene allo stesso modo dei metodi locali, con l'unica eccezione della gestione della **RemoteException**, di cui il client non sa nulla, tranne per il fatto che si è verificato un problema in un punto qualsiasi della comunicazione (Inizio, durante, fine). Per questo motivo per garantire la **semantica delle operazioni** i metodi remoti devono essere **idempotenti**, ovvero che a ripetute applicazioni dello stesso metodo con gli stessi parametri devono restituire sempre lo stesso risultato.

Passaggio di parametri: un metodo remoto può dichiarare solo parametri o valori che siano **serializzabili** (implementino l'interfaccia **Serializable**). Inoltre, il passaggio di dati a oggetti remoti avviene per **copia** e

quindi non per **riferimento** come invece avviene per oggetti locali. Questo porta a dei problemi che Java RMI risolve garantendo che nel caso in cui **vengano passati, nella stessa invocazione due riferimenti allo stesso oggetto**, allora punteranno allo stesso oggetto, mantenendo così **l'integrità referenziale**.

[Esempio di tutto ciò, si trova al Cap. 3, par. 3.2](#)

Differenza tra il modello a oggetti locale e quello remoto: sono state apportate alcune modifiche agli oggetti della classe **object**:

- **X.hashCode():** viene ridefinito in maniera che restituisca lo stesso codice per due stub diversi gli oggetti remoti che si riferiscono allo stesso oggetto remoto. In questo modo le chiavi possono essere utilizzate in tabelle hash.
- **X.equals():** restituisce un booleano che è vero se il riferimento remoto passato è uguale a quello di X. Il confronto viene effettuato sugli **stub**.
- **X.toString():** Restituisce informazioni addizionali come la macchina su cui l'oggetto si trova.

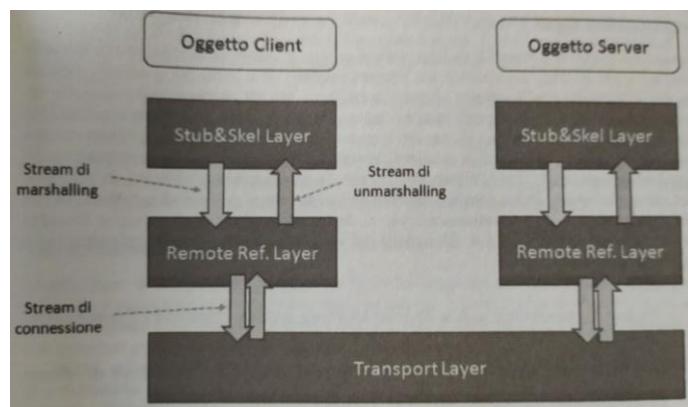
[Maggiori info, si trovano al Cap. 3, par. 3.3](#)

La sicurezza in java: per gestire la sicurezza java offre una **sandbox** (zona sicura), per garantire che le azioni svolte dai programmi siano controllate e ristrette, così di prevenire danni sia di natura intenzionale che non. La sicurezza java si basa su tre livelli:

- **La sicurezza intrinseca di java:** ovvero tutte quelle caratteristiche del linguaggio che garantiscono la sicurezza:
 - **Linguaggio fortemente tipizzato:** ogni dato ha un tipo, e solo poche operazioni di **casting** vengono eseguite automaticamente dal compilatore.
 - **Gestione automatica della memoria:** tramite il **garbage collector**.
 - **Assenza di puntatori:** insieme all'impossibilità di effettuare aritmetica sugli stessi, impedisce accessi illegali alla memoria.
 - **Lazy allocation:** la memoria del programma viene allocata a tempo di esecuzione, questo rende impossibile sapere in anticipo dove il programma andrà a finire e quindi impossibile rimpiazzare quelle parti di memoria con software malevolo.
- **Il Classloader:** carica le classi a tempo di esecuzione in modo da non permettere modifiche alle classi **built-in** del linguaggio (non posso modificare String per mandare le mie stringhe via mail).
- **Il Bytecode verifier:** controlla l'assenza di codice malevolo all'interno del bytecode e ne verifica la conformità con gli standard del linguaggio, questo è necessario perché il bytecode una volta generato è liberamente accessibile e quindi potrebbe essere manomesso.
- **Il Security Manager:** si occupa di definire i confini della sandbox. Viene interpellato dalla macchina virtuale per ciascuna operazione potenzialmente pericolosa e fornisce le autorizzazioni sulla base delle **policy** stabilite dall'utente.

I tre layer della architettura: il sistema di Java RMI è strutturato su tre livelli (layer):

- **Stub/Skeleton layer:** che comprende gli stub lato server e gli skeleton lato client. Con le versioni più moderne di java, questi due componenti vengono creati automaticamente, semplificando di molto la vita di programmatore al netto di una piccola perdita di efficienza.
- **Remote Reference Layer:** che specifica il comportamento della invocazione la semantica del riferimento (unicast, multicast. Etc...). in particolare, questo layer si occupa di fare da tramite tra il livello stub/skeleton e il livello di trasporto.



- **Trasport Layer:** che si occupa della connessione e della sua gestione. Chiamato **Java Remote Method Control**, ma con il tempo questo protocollo è stato sostituito o comunque reso modificabile per garantire una migliore compatibilità di **Java RMI**.

L'applicazione dell'utente si trova in cima a questi tre livelli.

Distribuite Garbage Collection, si trova al Cap. 3, par. 4.2

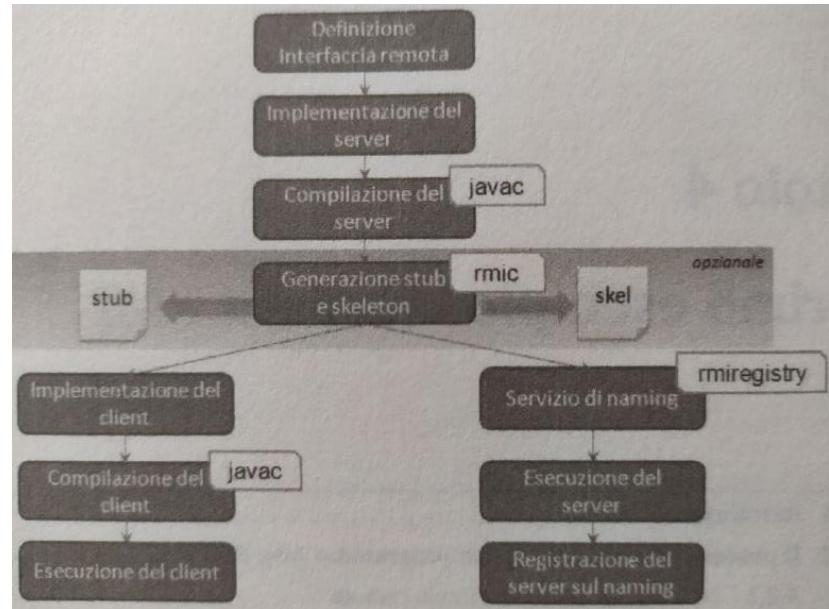
Caricamento dinamico delle classi, si trova al Cap. 3, par. 4.3

Marshalling: è un'operazione di serializzazione che modifica la semantica dei riferimenti remoti, aggiungendo informazioni all'oggetto. Il meccanismo di Marshalling di Java si basa sulla specializzazione della classe **ObjectOutputStream**.

Approfondimento, si trova al Cap. 3, par. 5.4

Processo di creazione di un programma in Java RMI:

Descrizione dei passi ed esempio di HelloWorld, si trova al Cap. 4, par. 4.2, 4.3

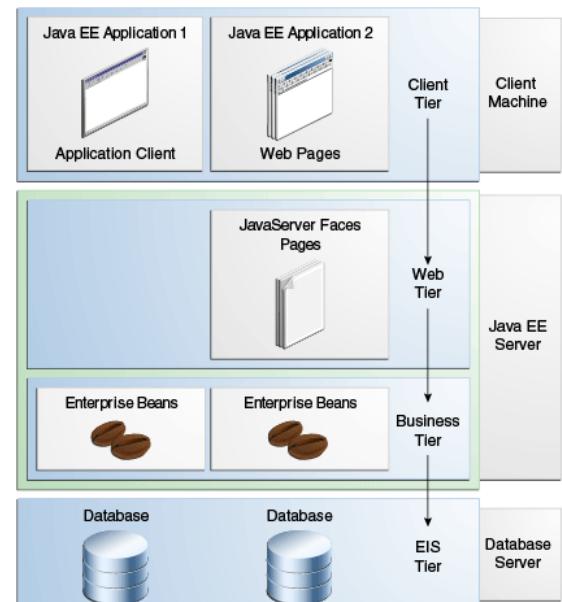


Lezione 13, Introduzione a Java Enterprise:

Java EE: Nasce dalla necessità degli sviluppatori di: applicazioni distribuite, transazionali e portatili che sfruttino la velocità, la sicurezza e l'affidabilità della tecnologia lato server. Con lo scopo ulteriore di ridurre i tempi di sviluppo, il capitale investito e garantire la massima efficienza del prodotto creato.

Java EE, utilizza un modello di applicazione distribuito su più livelli.

- I **componenti di livello client** vengono eseguiti sul computer client e sono di due tipi:
 - **Client Web:** semplice pagina web che funge da UI, molto utile visto che tutti hanno un browser.
 - **Client applicativo:** applicazione software installabile, che viene utilizzato in caso di necessità di un'interfaccia più ricca.
- I **componenti di livello Web** vengono eseguiti sul server Java EE.
- I **componenti di livello aziendale** eseguono la logica di business e si basano sull'utilizzo di Java Beans.
- Il **software EIS** (Enterprise Information System) funziona sul server EIS.



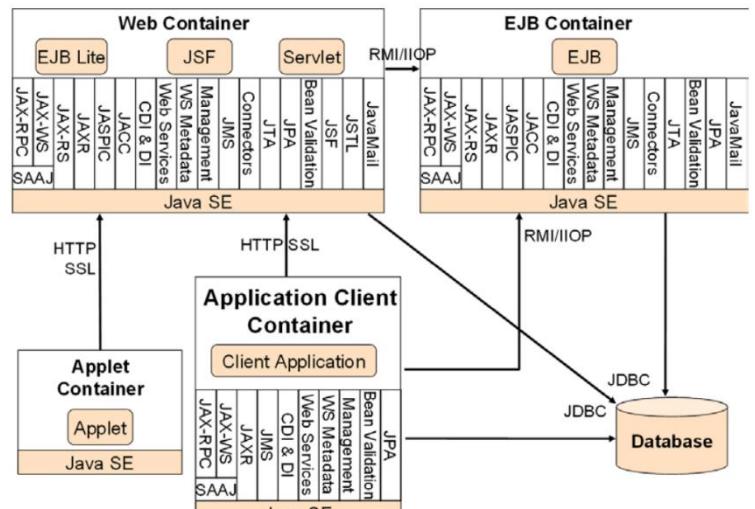
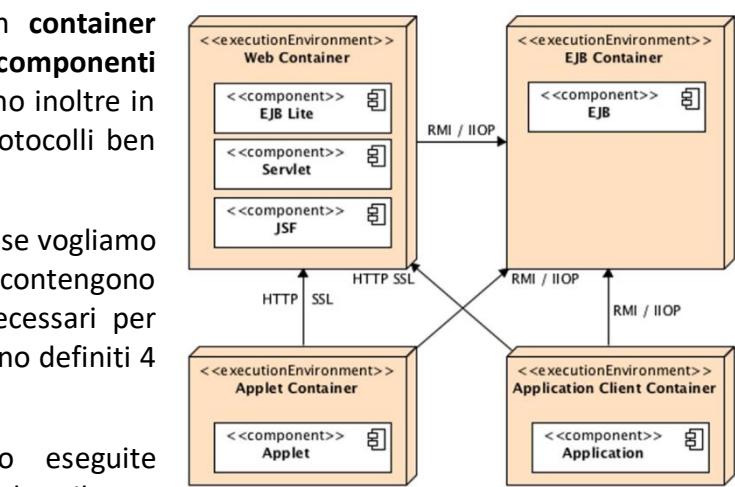
Molto importante è anche l'organizzazione in **container** ognuno con uno specifico ruolo, che contengono **componenti** e che offrono alcuni servizi. Questi container sono inoltre in grado di comunicare fra di loro, tramite dei protocolli ben definiti.

La logica di Java EE si basa sulle componenti, che se vogliamo sono la naturale evoluzione degli oggetti (non contengono solo in software, ma anche le classi e i file necessari per esempio il **deployment descriptor**). In Java EE sono definiti 4 tipi di componenti:

- **Le applet:** applicazioni che vengono eseguite all'interno di browser web e utilizzate per lo sviluppo di UI.
- **Applicazioni:** sono programmi eseguiti su un client. Sono in genere GUI o programmi di elaborazione batch.
- **Le applicazioni Web:** (composte da servlet, filtri servlet, listener di eventi Web, pagine JSP e JSF) sono eseguite in un contenitore Web e rispondono alle richieste HTTP dai client Web.
- **Applicazioni aziendali:** Implementano la logica di business e sono: Enterprise Java Beans, Java Message Service, Java Transaction API, chiamate asincrone, servizio timer, RMI / IIOP. vengono eseguiti in un contenitore EJB.

Tutto questo viene assemblato e gestito da un server Java EE. Infatti, in JEE è presente un ulteriore fase oltre la compilazione e l'esecuzione, questa fase detta **Packaging** è divisa a sua volta in due fasi **Assemblaggio** e **Deployment** in cui le componenti utili al programma e quindi all'esecuzione vengono prima unite fra di loro e successivamente affidate ad un container che le prende in carico e le esegue.

Packaging: Per essere distribuiti in un contenitore, i componenti devono prima essere impacchettati in un archivio formattato standard. In Java EE questo tipo di archivio è chiamato **Java Archive (jar)** ma ce ne sono anche altro per esempio **Web Archive (war)**. All'interno del package sono contenute anche informazioni sugli oggetti e i file, chiamati **descrittori** di solito queste informazioni vengono contenute in dei file **XML**.



Java EE Container: Normalmente, le applicazioni multilivello web/client sono difficili da scrivere perché coinvolgono molte righe di codice intricato per gestire vari aspetti come: la gestione delle transazioni e dello stato, il multithreading, il pool di risorse e altri dettagli complessi di basso livello.

Java EE semplifica la scrittura delle applicazioni poiché la logica aziendale è organizzata in componenti riutilizzabili. Inoltre, il server Java EE fornisce servizi sottostanti sotto forma di **container** per ogni tipo di componente. Poiché non è necessario sviluppare questi servizi da soli, si è liberi di concentrarsi sulla risoluzione del problema aziendale a portata di mano.

I **container** si basano su macchine virtuali standard edition e sul un grande numero di operazioni che quest'ultima riesce a realizzare. Vengono utilizzati sia sul server che sul lato client, dove rendono facile la gestione dell'applicazione verso servizi esterni.

Un ottimo esempio di utilizzo dei container è la gestione della persistenza. Infatti, questi ultimi permettono di rendere persistente un oggetto senza preoccuparsi della gestione di JDBC, senza nemmeno preoccuparsi della gestione delle tabelle.

Ciclo di vita degli Enterprise Java Beans: (**Ci boccia la gente con sta cosa**) viene completamente gestito dal container. Di norma gli oggetti nascono con una **new** e vengono distrutti quando non hanno più riferimenti a loro stessi e vengono presi dal garbage collection. In Java EE invece il ciclo di vita viene gestito interamente (sia **nascita** che **distruzione**) dal container, infatti quest'ultimo instanzia l'oggetto quando serve e lo elimina quando diventa inutile sostituendo completamente il garbage collector. Con questo metodo gli oggetti possono essere addirittura riutilizzati per altri scopi, per esempio se un bean diventa inutile e sta per essere distrutto mentre un **sta per essere creato** si riutilizza il vecchio sovrascrivendolo. Questo porta ad un'enorme incremento delle performance.

Annotazioni e Deployment Descriptors: per dire di quali applicazioni, oggetti, classi ho bisogno ci sono due metodi, il primo metodo è usare le annotazioni il secondo è utilizzare un **Deployment Descriptor (il file XML dei package praticamente)**. Tutto quello che si può fare in XML non si può fare con le annotazioni ma vale il viceversa. In questo modo un **POJO (Plain Old Java Object)** può diventare un componente (uno degli obiettivi di JEE, praticamente). Io so fare i POJO in Java SE e posso tranquillamente portali in Java EE senza troppe modifiche).

| Annotazioni: | Deployment Descriptor: |
|---|--|
| <pre> @Stateless @Remote(ItemRemote.class) @Local(ItemLocal.class) @LocalBean public class ItemEJB implements ItemLocal, ItemRemote { @PersistenceContext(unitName = "chapter01PU") private EntityManager em; public Book findBookById(Long id) { return em.find(Book.class, id); } } </pre> | <pre> <ejb-jar xmlns="http://xmlns.jcp.org/xml/ns/javaee" → xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" → xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee → http://xmlns.jcp.org/xml/ns/javaee/ejb-jar_3_2.xsd" → version="3.2"> <enterprise-beans> <session> <ejb-name>ItemEJB</ejb-name> <remote>org.agoncal.book.javaee7.ItemRemote</remote> <local>org.agoncal.book.javaee7.ItemLocal</local> <local-bean/> <ejb-class>org.agoncal.book.javaee7.ItemEJB</ejb-class> <session-type>Stateless</session-type> <transaction-type>Container</transaction-type> </session> </enterprise-beans> </ejb-jar> </pre> |

- **Non ci sono import:** lo sa il container di cosa ha necessità e cosa deve importare.
- **Stateless:** significa che l'oggetto non ha memoria è composto da solo di metodi, per questo può essere utilizzato da tutti i programmi che hanno necessità di usare quei metodi.
- **Remote e Local:** significa che il bean è sia remoto che locale.
- **PersistenceContext:** è l'insieme delle componenti che vanno rese persistenti allo stesso modo. Banalmente è il database a cui facciamo riferimento.
- **Non ci sono instanziazioni,** perché appunto il container fa tutto lui.

Extra: Java EE si basa su standard, viene infatti chiamato **specifiche ombrello** perché raggruppa un numero di altre specifiche (o richieste di specifiche Java).

La piattaforma Java EE viene sviluppata tramite il **Java Community Process JCP**, ovvero tramite un gruppo di esperti composto dalle parti interessate (è la community stessa a decidere cosa implementare il Java EE e le aziende che hanno interessi diretti), le versioni inoltre vengono rilasciate con una cadenza molto lente, in modo da permettere alle aziende di rimanere al passo.

Java EE, **fornisce un ambiente aperto**, ovvero non ti blocca sull'utilizzo di software da parte di un singolo fornitore (non hai il produttore con il coltello alla gola).

Lezione 14-15, Context and Dependency Injection:

Inversion control: si tratta di un design pattern (vocabolario condiviso) utilizzato nella programmazione enterprise. In breve, il controllo del codice non è del programmatore ma del container, che offre servizi al codice, ed è quest'ultimo che decide quanto instanziare un oggetto (**dependency injection**) o fare un **context injection**, ovvero il programmatore non è solo nello sviluppo, ma ha il container che lo "aiuta". Questo è possibile tramite i **managed beans** ovvero dei bean creati per essere gestiti dal container.

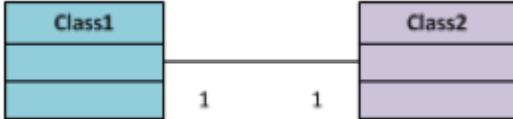
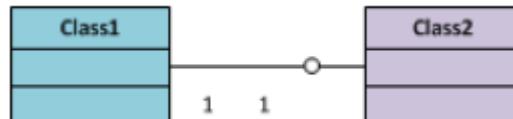
<< La seconda risposta per tutto è: IL Container. Ma dipende sempre...>>

loose coupling e strong typing: tutto è tipizzato e niente viene convertito dal compilatore se non è esplicitamente richiesto tramite il casting, questo è un vantaggio per quanto riguarda la correttezza del codice. Allo stesso tempo la gestione e le esecuzioni delle classi non è vincolata da altre classi abbiamo un **basso accoppiamento** e questo è un enorme vantaggio (esempio programmatore classe 1 alle Hawaii programmatore classe due a San Francisco).

Ci permette, inoltre di inserire diverse componenti che vengono eseguiti all'interno del programma tra cui:

- Interceptors;
- Decorator;
- Gestione degli eventi.

Esempio di accoppiamento:

| Due classi strettamente accoppiate: | Debolmente accoppiate: |
|---|--|
| <pre>public class Class1 { public Class2 Class2 { get; set; } } public class Class2 {}</pre> | <pre>public class Class1 { public IClass2 Class2 { get; set; } } public interface IClass2 {}</pre> <pre>public class Class2 : IClass2 {}</pre> |
|  |  |
| La classe 1 è accoppiata alla 2, ovvero la classe due per essere eseguita ha bisogno che la due sia implementata e funzionante. | Per rendere più "lasca" questa dipendenza si può definire un interfaccia che ha la definizione dei metodi e all'interno abbiamo una classe che implementa questa interfaccia. Anche se non risolviamo il problema, perché da qualche parte nel costruttore ci deve essere un'invocazione della effettiva costruzione, ovvero dobbiamo sapere com'è costruito classe due. |
| Maggiori info, con spiegazione completa a questo link. | |

Per risolvere definitivamente il problema ci avvaliamo del **content injection** dove praticamente container si prende tutta la responsabilità di implementare la classe due che abbiamo richiesto.

Principio di Hollywood: quando vai a fare un colloquio come attore ti dicono "ti facciamo sapere". Praticamente è il container che decide quando iniettare la risorsa, non la risorsa (l'attore).

Pattern di dependency injection: è una parte dell'**inversion of control** il concetto è di richiedere l'inserimento di un oggetto di un certo tipo. Praticamente metto un salva posto è poi il container inietta la risorsa da me richiesta.

Managed Beans: sono dei bean java progettati per essere gestiti dal container, con queste caratteristiche:

- Non è una **non-static inner class**;
- È un classe concreta o annotata come decoratore;
- Ha uno scope, un EL name, un insieme di Interceptor e un gestore del ciclo di vita (queste sono caratteristiche opzionali).
- Costruttore di default che non deve essere final;
- Attributi privati accessibili tramite getter e setter;
- Sono gestiti da un container;
- Contengono metadati;

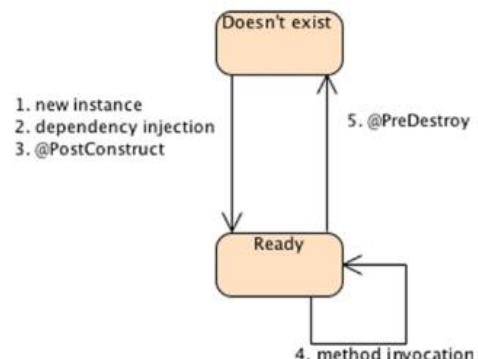
Vantaggi dei managed beans:

- **Non devo gestire lo stato di uno oggetto:** ci pensa il container (esempio del carrello di Amazon nei giorni, in teoria andrebbe gestita da noi ma invece da ora ci pensa il container che gestisce tutta la parte di database ecc...);
- **Sono legati a contesti ben definiti:** vengono usati in contesti che sono ben definiti;
- **Hanno un approccio type safe:** non sto accedendo a delle cose amorfie ma a delle risorse iniettate con una precisa struttura derivata dalla classe java. (se è uno studente so che ha il **metodo getName** per esempio);
- **Possono essere specializzati:**
- **Possono essere usati nel layer di presentazione;**
- **Ogni classe java può essere una bean CDI (Content Dependence Injection);**

Ciclo di vita dei managed bean: nei POJO il tutto era molto semplice, gli oggetti nascevano con la **new** e morivano quando non servivano più tramite il garbage collector.

Adesso la situazione è più complessa. Il ciclo di vita si divide in quattro fasi gestite dal container:

1. **New Instance:** prima di questa fase il bean non esiste (problemi filosofici), il container crea un oggetto tramite new (il container);
2. **Dependency injection:** inietta le dipendenze (perché anche il bean può avere delle dipendenze);
3. **Post construct:** eseguo i metodi **@PostConstruct** (metodi post costruzione) e mi trovo nello stato di **ready** dove il bean è pronto a rispondere alle chiamate fino a quando sarà necessario;
4. **Pre destroy:** una volta che il bean diventa inutile vengono eseguiti i metodi annotati come **@predestroy** (che per esempio chiudono connessioni) e il bean torna allo stato di inesistenza.



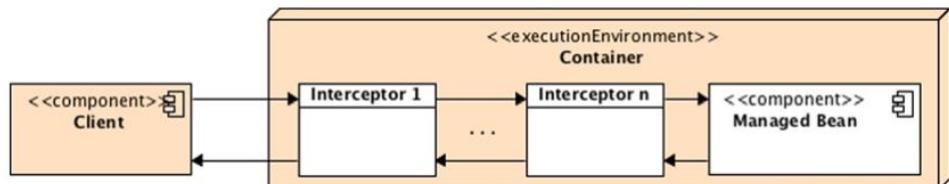
Cosa molto importante è che per chi ha richiesto l'inezione di questa risorsa tutte queste fasi sono del tutto **trasparenti**, vengono eseguite tutte dal container.

Infine, abbiamo **scoope** che è il contesto del bean, ovvero descrive che tipo di vita ha il bean, ed è importante per permetterà al container di ottimizzare la vita di quest'ultimo (deve durare poco, deve lavorare per una sessione e quindi durare tanto ? informazioni di questo genere).

Interceptor: si frappongono nelle invocazione di un metodo di business. Praticamente sto inserendo dei metodi all'interni di metodi da me scritti, senza modificare il codice sorgente di questi ultimi.

Vengono molto utilizzati nella **programmazione legata agli aspetti**.

Esempio: ho tanti metodi che trattano i dati personali di una persona e per motivi legali, in ogni momento che ognuno di questi metodi viene eseguito voglio l'assenso dell'utente.



Giustamente se i metodi sono tanti devo andare a modificarli a uno a uno (ed è una rottura di palline).

La soluzione è far intercettare ognuno di questi metodi da un altro. In pratica **modifico la maniera in cui il container gestisce l'invocazione** dei metodi in modo che aggiunga delle caratteristiche a tutti o ad alcuni di questi metodi, senza modificare il codice sorgente, semplicemente modificando il codice XML, addirittura non serve manco ricompilarlo e possono essere create anche **catene di intercettori** per eseguire più operazioni.

La potenza è che disaccoppiano la logica di business (quello che il metodo deve fare), dalle quello che fanno gli Interceptor, mantenendo però il controllo della tipizzazione.

Primo esempio di CDI Bean: L'esempio tratta di un sistema di gestione di una librerie che permette di creare un oggetto libro, aggiungergli degli attributi tra cui l'**ISBN** o l'**ISSN** (in caso di libri vecchi), per poi renderlo persistente (salvarlo in un DataBase). Il problema sta nel capire quando iniettare il **NumberGenerator**, che genera l'ISBN e quando quello che genera l'ISSN. Normalmente utilizzando un approccio di programmazione standard faremo questo:

“Selezionando” a mano ad ogni creazione di un nuovo oggetto il tipo di generatore di codice da utilizzare:

```

public class BookService {
    private NumberGenerator numberGenerator;

    public BookService(NumberGenerator numberGenerator) {
        this.numberGenerator = numberGenerator;
    }

    public Book createBook(String title, Float price, String description) {
        Book book = new Book(title, price, description);
        book.setIsbn(numberGenerator.generateNumber());
        return book;
    }
}

BookService bookService = new BookService(new IsbnGenerator());
BookService bookService = new BookService(new IssnGenerator());

```

Soluzione poco pratica e molto macchinosa.

In Java EE tutto questo può essere fatto dal container tramite le **@Inject**.

@Inject: è un annotazione di iniezione che permette di iniettare codice all'interno del nostro. Il punto in cui si esegue l'iniezione viene detto **Injection Point**. In Java EE esistono tre meccanismi di iniezione:

- **Property:** Quando viene eseguita sugli attributi (come nell'immagine sopra);
- **Constructor:** Quando l'iniezione viene eseguita sul costruttore;

```

@.Inject
private NumberGenerator numberGenerator;

public Book createBook(String title, Float price, String description) {
    Book book = new Book(title, price, description);
    book.setIsbn(numberGenerator.generateNumber());
    return book;
}

```

```

@.Inject
public BookService (NumberGenerator numberGenerator) {
    this.numberGenerator = numberGenerator;
}

```

- Setter:** Quando l'注射 viene eseguita sul setter:

@Inject

```
public void setNumberGenerator(NumberGenerator numberGenerator) {
    this.numberGenerator = numberGenerator;
}
```

È importante notare che ognuno di questi metodi funziona e che la scelta fra questi va puramente in base ai gusti del programmatore.

Default Injection: nel caso in cui nel nostro esempio non avessimo il problema dell'ISBN/ISSN ci basterebbe semplicemente iniettare il NumberGenerator con L'ISBN, questo tipo di iniezione viene detta di **default**. Praticamente se non dichiariamo nessun **Qualifier**, Java EE impone automaticamente quello di Default.

@Inject @Default

```
private NumberGenerator numberGenerator;
```

`@Default` is a built-in qualifier that informs CDI to inject the default bean implementation. If you define a bean with no qualifier, the bean automatically has the qualifier `@Default`. So code in Listing 2-6 is identical to the one in Listing 2-5.

Listing 2-6. The ISBNGenerator Bean with the @Default Qualifier

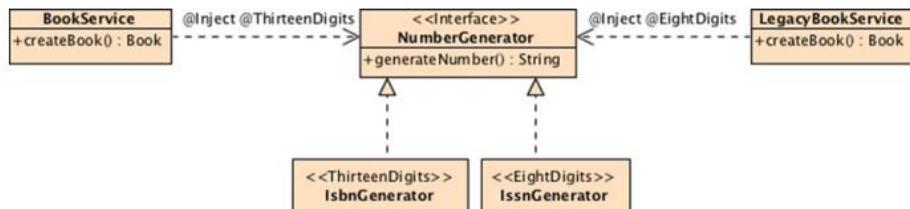
```
@Default
public class ISBNGenerator implements NumberGenerator {

    public String generateNumber() {
        return "13-84356-" + Math.abs(new Random().nextInt());
    }
}
```

Se invece ci troviamo (come nel nostro esempio), ad avere più risorse da iniettare, si utilizzano i **Qualifier**.

Qualifier: è un meccanismo che permette di informare il container su quale classe iniettare utilizzando il sistema delle annotazioni (quindi evitato l'uso del tanto odiato XML XD). Più nello specifico del nostro esempio su quale implementazione di NumberGenerator iniettare.

Come si vede dal diagramma entrambi le implementazioni, fanno riferimento alla stessa interfaccia.



La prima cosa da fare è dichiarare i Qualifier successivamente vano applicati e infine possono essere utilizzati a fianco alle `@Inject`:

| ISBN: | ISSN: |
|--|--|
| @Qualifier <code>@Retention(RUNTIME)</code> <code>@Target({FIELD, TYPE, METHOD})</code> <code>public @interface ThirteenDigits { }</code> @ThirteenDigits <code>public class ISBNGenerator implements NumberGenerator {</code> <code> public String generateNumber() {</code> <code> return "13-84356-" + Math.abs(new Random().nextInt());</code> <code> }</code> <code>}</code> <code>public class BookService {</code> <code> @Inject @ThirteenDigits</code> <code> private NumberGenerator numberGenerator;</code> <code></code> <code> public Book createBook(String title, Float price, String description) {</code> <code> Book book = new Book(title, price, description);</code> <code> book.setISBN(numberGenerator.generateNumber());</code> <code> return book;</code> <code> }</code> <code>}</code> | @Qualifier <code>@Retention(RUNTIME)</code> <code>@Target({FIELD, TYPE, METHOD})</code> <code>public @interface EightDigits { }</code> @EightDigits <code>public class ISSNGenerator implements NumberGenerator {</code> <code> public String generateNumber() {</code> <code> return "8-" + Math.abs(new Random().nextInt());</code> <code> }</code> <code>}</code> <code>public class LegacyBookService {</code> <code> @Inject @EightDigits</code> <code> private NumberGenerator numberGenerator;</code> <code></code> <code> public Book createBook(String title, Float price, String description) {</code> <code> Book book = new Book(title, price, description);</code> <code> book.setISBN(numberGenerator.generateNumber());</code> <code> return book;</code> <code> }</code> <code>}</code> |

Questo tipo di sistema offre numerosi vantaggi, infatti posso rinominare la mia implementazione come mi pare, rinominando i Qualifier, senza che il punto di iniezione cambi (questo garantisce il tanto agognato **basso accoppiamento**).

Qualificatori con membri: per evitare di creare un grande numero di qualificatori all'interno del nostro codice è possibile “personalizzarli” inserendo dei membri, esempio:

In questo modo abbiamo un unico qualificatore che agisce per ognuno dei nostri casi.

```
@Inject @NumberOfDigits(value = Digits.THIRTEEN, odd = false)
private NumberGenerator numberGenerator;

And the concerned implementation will do the same.

@NumberOfDigits(value = Digits.THIRTEEN, odd = false)
public class IsbnEvenGenerator implements NumberGenerator {...}
```

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface NumberOfDigits {

    Digits value();
    boolean odd();
}

public enum Digits {
    TWO,
    EIGHT,
    TEN,
    THIRTEEN
}
```

Qualificatori Multipli: un altro per ridurre il numero di qualificatori usati è quello di utilizzare qualificatori multipli, per esempio uno per il numero di cifre e uno per la parità o la disparità:

```
@ThirteenDigits @Even
public class IsbnEvenGenerator implements NumberGenerator {...}

The injection point would use the same syntax.

@Inject @ThirteenDigits @Even
private NumberGenerator numberGenerator;
```

Alternatives: a volte può essere necessario scegliere quale risorsa iniettare in base a un determinato scenario di utilizzo. Per esempio, se stiamo facendo testing potremmo desiderare di generare un numero fittizio. Per fare ciò ci vengono in aiuto le alternative, che sono disattivate di default e nel caso in cui volessimo usarle devono essere attivate ([Pagina 36 libro](#)).

Producers: di norma all'interno di Java EE non è possibile iniettare tipi primitivi o **POJO** (String, java.util.data, java.util.String queste cose qui...) perché sono impacchettati in file **rt.jar** e questo tipo di archivio non contiene **deployment descriptor (beans.XML)** come invece lo hanno i beans CDI. Quindi per iniettare questo tipo di oggetti dobbiamo utilizzare i produttori.

```
public class NumberProducer {

    @Produces @ThirteenDigits
    private String prefix13digits = "13-";

    @Produces @ThirteenDigits
    private int editorNumber = 84356;

    @Produces @Random
    public double random() {
        return Math.abs(new Random().nextInt());
    }
}
```

```
@ThirteenDigits
public class IsbnGenerator implements NumberGenerator {

    @Inject @ThirteenDigits
    private String prefix;

    @Inject @ThirteenDigits
    private int editorNumber;

    @Inject @Random
    private double postfix;

    public String generateNumber() {
        return prefix + editorNumber + postfix;
    }
}
```

Disposer: alcuni tipo di oggetti e tipi di dati, una volta creati ed utilizzati necessitano di essere distrutti (connessioni JDBC, JMS devono essere chiuse), per fare ciò si utilizzano i disposer. Praticamente con il **producer** creiamo e con il **disposer** ne gestiamo la distruzione o la chiusura, come nell'esempio che gestisce una connessione con un JDBC.

```
public class JDBCConnectionProducer {

    @Produces
    private Connection createConnection() {
        Connection conn = null;
        try {
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();
            conn = DriverManager.getConnection("jdbc:derby:memory:chapter02DB", "APP", "APP");
        } catch (InstantiationException | IllegalAccessException | ClassNotFoundException) {
            e.printStackTrace();
        }
        return conn;
    }

    private void closeConnection(@Disposes Connection conn) throws SQLException {
        conn.close();
    }
}
```

```
@ApplicationScoped
public class DerbyPingService {

    @Inject
    private Connection conn;

    public void ping() throws SQLException {
        conn.createStatement().executeQuery("SELECT 1 FROM SYSIBM.SYSDUMMY1");
    }
}
```

Scopes: Ogni oggetto gestito in CDI ha uno scopo ben definito e un ciclo di vita che è legato a quest'ultimo (nei POJO invece è tutto semplice, lo creo con new e poi ci pensa la GC), infatti è il container che si preoccupa di creare un oggetto CDI ed è solo lui che può distruggerlo (non si può fare in modo manuale), praticamente conoscendo lo scopo di un beans il container sa qual è il modo più efficiente per gestirlo (crearlo, usarlo, distruggerlo). Per questo motivo CDI mette a disposizione degli **scopes** ben definiti, con la possibilità di crearne dei personalizzati se necessario:

- **Application Scope (@ApplicationScoped):** si estende per l'intera durata di un'applicazione. Il bean viene creato una sola volta per la durata dell'applicazione e viene eliminato quando il file l'applicazione è chiusa (sono bean di supporto per gestire o spostare dati).
- **Session Scope (@SessionScoped):** Utilizzato per richieste HTTP o su metodi di invocazione per la sessione di un singolo utente. Il bean viene creato per la durata di una sessione http e viene scartato al termine della sessione (Preferenze utente, dati di accesso, carrello Amazon).
- **Request Scope (@RequestScoped):** utilizzato per una singola richiesta HTTP o un metodo invocazione. Il bean viene creato per la durata dell'invocazione del metodo e viene scartato quando il metodo termina (l'invio di una form per esempio).
- **Conversation Scope(@ConversationScoped):** si estende tra più invocazioni all'interno i limiti della sessione con i punti iniziale e finale determinati dall'applicazione. Le conversazioni vengono utilizzate su più pagine come parte di un flusso di lavoro a più fasi. ([Pagine 42-43 libro](#)).
- **Dependent pseudo-scope (@Dependent):** il ciclo di vita è uguale a quello del client. Viene creato ogni volta che viene iniettato e il riferimento viene rimosso quando il target di iniezione è rimosso. Questo è l'ambito predefinito per CDI.

```
@Dependent @ThirteenDigits
public class IsbnGenerator implements NumberGenerator {...}

Being the default scope, you can omit the @Dependent annotation and write the following:

@ThirteenDigits
public class IsbnGenerator implements NumberGenerator {...}
```

Interceptor: come abbiamo visto in precedenza sono dei costrutti che permettono al container di intercettare la chiamata ad un metodo e inserire della logica prima di quella del business (permettono di aggiungere codice ai metodi senza toccarli). Esistono 4 categorie di intercettori:

- **Constructor-level interceptors:** intercettore associato a un costruttore della classe target (@AroundConstruct);
- **Method-level interceptors:** intercettore associato a un metodo business specifico (@AroundInvoke);
- **Timeout method interceptors:** intercettore che si interpone sui metodi di timeout con @AroundTimeout (utilizzato solo con il servizio timer EJB);
- **Life-cycle callback interceptors:** intercettore che si interpone sul ciclo di vita dell'istanza target. (@PostConstruct e @PreDestroy).

Target Class Interceptors: ci sono diversi modi per definire un intercettore, il modo più semplice è inserirli all'interno del bean stesso.

In questo esempio, qualsiasi client utilizzi il metodo **createCustomer** o **findCustomerById**, verrà intercettato e sarà obbligato ad eseguire il metodo **logMethod**.

Un **around-invoke method** deve avere le seguenti caratteristiche:

- Non deve essere **static o final**;
- Deve avere come parametro un **Javax.interceptor.InvocationContext** e deve ritornare un oggetto;
- Deve lanciare una **Checked Exception**;

```
@Transactional
public class CustomerService {

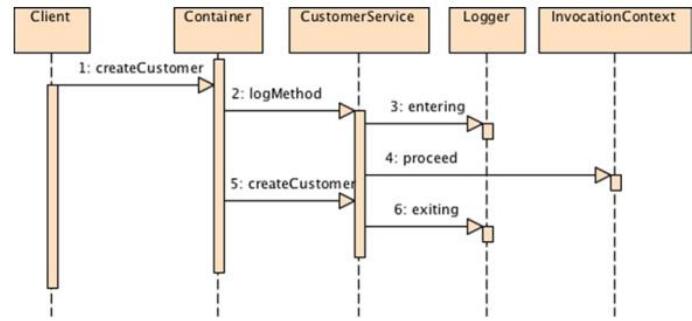
    @Inject
    private EntityManager em;
    @Inject
    private Logger logger;

    public void createCustomer(Customer customer) {
        em.persist(customer);
    }

    public Customer findCustomerById(Long id) {
        return em.find(Customer.class, id);
    }

    @AroundInvoke
private Object logMethod(InvocationContext ic) throws Exception {
    logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
    try {
        return ic.proceed();
    } finally {
        logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
    }
}
```

Funzionamento: il container intercetta la chiamata, ed esegue **logMethod**, che fa quello che deve fare, successivamente logMethod lancia **proceed()** per procedere con il prossimo intercettore se c'è altrimenti si procede con il metodo inizialmente invocato ovvero **createCustomer()**.



```

public class LoggingInterceptor {
    @Inject
    private Logger logger;

    @AroundConstruct
    private void init(InvocationContext ic) throws Exception {
        logger.fine("Entering constructor");
        try {
            ic.proceed();
        } finally {
            logger.fine("Exiting constructor");
        }
    }

    @AroundInvoke
    public Object logMethod(InvocationContext ic) throws Exception {
        logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
        try {
            return ic.proceed();
        } finally {
            logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
        }
    }
}
  
```

Class Interceptors: si possono anche definire delle classi di intercettori da utilizzare su interi bean oppure su specifici metodi di specifici bean.

```

@AroundInvoke
public Object logMethod(InvocationContext ic) throws Exception {
    logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
    try {
        return ic.proceed();
    } finally {
        logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
    }
}
  
```

Esempi pagine 47-48-49 libro.

Life-Cycle Interceptor: inserendo all'interno della classe intercettore un metodo annotato come **PostConstruct** quest'ultimo verrà invocato prima del metodo PostConstruct della classe target.

```

public class ProfileInterceptor {
    @Inject
    private Logger logger;

    @PostConstruct
    public void logMethod(InvocationContext ic) throws Exception {
        logger.fine(ic.getTarget().toString());
        try {
            ic.proceed();
        } finally {
            logger.fine(ic.getTarget().toString());
        }
    }

    @AroundInvoke
    public Object profile(InvocationContext ic) throws Exception {
        long initTime = System.currentTimeMillis();
        try {
            return ic.proceed();
        } finally {
            long diffTime = System.currentTimeMillis() - initTime;
            logger.fine(ic.getMethod() + " took " + diffTime + " millis");
        }
    }
}
  
```

```

@Transactional
@Interceptors(ProfileInterceptor.class)
public class CustomerService {

    @Inject
    private EntityManager em;

    @PostConstruct
    public void init() {
        // ...
    }

    public void createCustomer(Customer customer) {
        em.persist(customer);
    }

    public Customer findCustomerById(Long id) {
        return em.find(Customer.class, id);
    }
}
  
```

Esempi pagine 49-50 libro.

Chaining and Excluding Interceptors: è possibile associare ad una classe target una catena di intercettori:

```

@Stateless
@Interceptors({I1.class, I2.class})
public class CustomerService {
    public void createCustomer(Customer customer) {...}
    @Interceptors({I3.class, I4.class})
    public Customer findCustomerById(Long id) {...}
    public void removeCustomer(Customer customer) {...}
    @ExcludeClassInterceptors
    public Customer updateCustomer(Customer customer) {...}
}
  
```

Interceptor Binding: per garantirlo il **loose-coupling** all'interno i java EE è stata inserito il **binding** degli intercettori (praticamente se per usare un intercettore devo scrivere il nome della classe è comunque fortemente accoppiata la situazione), per questo è possibile definire un certo tipo di intercettore, implementarlo e poi utilizzarlo in modo disaccoppiato:

| Definizione: | Implementazione: |
|--|---|
| <pre>@InterceptorBinding @Target({METHOD, TYPE}) @Retention(RUNTIME) public @interface Loggable { }</pre> | <pre>@Interceptor @Loggable public class LoggingInterceptor { @Inject private Logger logger; @AroundInvoke public Object logMethod(InvocationContext ic) throws Exception { logger.entering(ic.getTarget().toString(), ic.getMethod().getName()); try { return ic.proceed(); } finally { logger.exiting(ic.getTarget().toString(), ic.getMethod().getName()); } } }</pre> |
| Utilizzo: | |
| <pre>@Transactional public class CustomerService { @Loggable public void createCustomer(Customer customer) {...} public Customer findCustomerById(Long id) {...} }</pre> | |

Priorizzare gli intercettori: L'associazione degli intercettori ci offre un livello di riferimento indiretto, ma ci fa perdere la possibilità di ordinare gli intercettori. Per questo è possibile dare una priorità agli intercettori usando il **@Priority** insieme a un valore di priorità (il valore più piccolo viene chiamato prima):

```
@Interceptor
@Loggable
@Priority(200)
public class LoggingInterceptor {

    @Inject
    private Logger logger;

    @AroundInvoke
    public Object logMethod(InvocationContext ic) throws Exception {
        logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
        try {
            return ic.proceed();
        } finally {
            logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
        }
    }
}
```

Infine, come per le **alternative gli intercettori** non sono attivati per default, vanno pertanto attivati in caso di necessità ([pagina 25 libro](#)).

Decorators: sono estremamente simili agli intercettori, anche se lavorano in modo diverso, infatti, se l'intercettore ignora la logica di business che viene dopo la sua esecuzione, perché appunto semplicemente è una classe assestante eseguita prima. Il decoratore no, perché non viene svolto prima delle **classe target** ma viene costruito intorno ad 'essa, modificandola e decorandola:

In questo esempio, invece di avere due classi una per l'ISBN e una per l'ISSN, si modifica la classe ISSN tramite un decoratore aggiungendo le 4 cifre mancanti.

```
@Decorator
public class FromEightToThirteenDigitsDecorator implements NumberGenerator {

    @Inject @Delegate
    private NumberGenerator numberGenerator;

    public String generateNumber() {
        String issn = numberGenerator.generateNumber();
        String isbn = "13-84356" + issn.substring(1);
        return isbn;
    }
}
```

I decoratori devono avere un punto di iniezione delegato (annotato con **@Delegate**), con lo stesso tipo dei bean che decorano (NumberGenerator nell'esempio). Questo consente al decoratore di invocare l'oggetto delegato (ovvero, il bean di destinazione IssnNumberGenerator) e quindi invocare qualsiasi metodo di business su di esso (come `numberGenerator.generateNumber ()`).

Come per le altre cose anche i decoratori sono disabilitati e vanno attivati in caso di necessità.

Eventi: permettono ai beans di interagire senza dipendenze a tempo di esecuzione. Un bean può, creare, dare fuoco (**fire()**) e gestire un evento. I produttori di eventi generano eventi utilizzando l'interfaccia javax.enterprise.event.Event. Un produttore genera eventi chiamando il metodo fire(), passa l'oggetto evento e non dipende dall'osservatore.

BookService genera un evento (**bookAddedEvent**) ogni volta che viene creato un libro. Il codice **bookAddedEvent.fire(book)** attiva l'evento e notifica ad eventuali **osservatori** di questo particolare evento.

```
public class BookService {
    @Inject
    private NumberGenerator numberGenerator;

    @Inject
    private Event<Book> bookAddedEvent;

    public Book createBook(String title, Float price, String description) {
        Book book = new Book(title, price, description);
        book.setIsbn(numberGenerator.generateNumber());
        bookAddedEvent.fire(book);
        return book;
    }
}
```

Un **osservatore** è un bean con uno o più metodi di osservazione, che in caso di attivazione dell'evento, eseguono il loro codice.

```
public class InventoryService {
    @Inject
    private Logger logger;
    List<Book> inventory = new ArrayList<>();

    public void addBook(@Observes Book book) {
        logger.info("Adding book " + book.getTitle() + " to inventory");
        inventory.add(book);
    }
}
```

Esempio:

| Osservato: | Osservatore: |
|--|---|
| <pre>public class BookService { @Inject private NumberGenerator numberGenerator; @Inject @Added private Event<Book> bookAddedEvent; @Inject @Removed private Event<Book> bookRemovedEvent; public Book createBook(String title, Float price, String description) { Book book = new Book(title, price, description); book.setIsbn(numberGenerator.generateNumber()); bookAddedEvent.fire(book); return book; } public void deleteBook(Book book) { bookRemovedEvent.fire(book); } }</pre> | <pre>public class InventoryService { @Inject private Logger logger; List<Book> inventory = new ArrayList<>(); public void addBook(@Observes @Added Book book) { logger.info("Adding book " + book.getTitle() + " to inventory"); inventory.add(book); } public void removeBook(@Observes @Removed Book book) { logger.info("Removing book " + book.getTitle() + " from inventory"); inventory.remove(book); } }</pre> |

Lezione 17-18, Java Persistent API:

L'utilità dei servizi: strutturare le applicazioni in servizi è utile perché permette di condividere questi ultimi, sia con gli utenti che con altri linguaggi o applicazioni, infatti non è necessario che la nostra applicazione si strutturata allo stesso modo di quella che offre il servizio, basta solamente che entrambe si scambino dati tramite un protocollo comune è aperto.

Caratteristiche trasversali: sono caratteristiche vengono fornite a tutti i livelli dell'architettura java EE.

End to End principe: la sicurezza di un sistema è data dalla somma del livello di sicurezza delle singole parti che lo compongono (lo stesso vale per l'efficienza). In parole povere se unico dei moduli non è detto che siano efficienti o sicuri come lo sono presi singolarmente e non è detto che non generino problemi inaspettati anche se i singoli moduli nel loro dominio funzionano perfettamente.

La persistenza in Java EE: i dati sono di solito memorizzati in database, uno strumento per assicurare la persistenza dei dati. La "vecchia" gestione dei database portava i programmatore a dover fare una distinzione tra la logica di questi ultimi e la logica della programmazione object oriented. Logicamente le due cose non sono molto diverse, infatti l'unica differenza sostanziale, sta nel fatto che le entità (DataBase) sono persistenti, mentre di oggetti (Object-Oriented) non lo sono (**Perché passa il Garbage Collector e li accoppa**).

Object-Relational Mapping (ORM): è un metodologia che permette di mappare oggetti in tabelle, unendo il mondo della programmazione ad oggetti con quello dei Database. JPA è un modulo presente fin da subito in Java EE, una astrazione ulteriore di JDBC che lo rende indipendente da SQL, (Prima doveva conoscere entrambi i linguaggi per gestire i DB).

Quindi ORM è un permette di assegnare oggetti a tabelle e lo fa tramite l'**Entity Manager** una componente JPA che si occupa di svolgere le cosiddette operazioni **crude** (Semplici, creazione tabella, inserimento di tuple ecc...), questo meccanismo si interfaccia bene con la gestione della persistenza di java.

Oggetti persistenti:

Creazione di un entità: un Entità è in parole il nome che un **Oggetto Persistente** assume nella logica dei DB. In Java EE la creazione di un entità si svolge, utilizzando due annotazioni, nel seguente modo:

1. **@Entity:** informa il container che quella che segue è un entità (oggetto persistente).
2. **@id:** rappresenta la nostra chiave primaria, che identificherà in modo univoco l'entità.
3. **Il resto:** sono attributi, costruttore e metodi getter e setter come una comune classe java.

L'annotazione **@Entity**, rende automaticamente l'oggetto persistente, incaricando il container (o meglio l'Entity Manager) di gestire tutto ciò che ne consegue (**Posso lavarmene le mani**).

Addirittura, in modo simile al loose coupling, non ho neanche bisogno di sapere che database sto usando.

Proprietà di un entità:

- Deve essere una **Classe Normale** con **Costruttore Public**;
- Se deve essere passata per valore deve implementare **Serializable**;

Riassumendo ciò che facciamo è scrivere dei metadati per associare l'oggetto alla tabella di un DataBase.

```
@Entity
public class Book {

    @Id @GeneratedValue
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;

    public Book() {
    }

    // Getters, setters
}
```

Configuration by Exception: è uno pattern che definisce l'importanza di fornire dei valori di default (**Convention**) quando ci sono molte configurazioni, come per JPA dove tutte le configurazioni sono già impostate a valori di default per il caso standard di utilizzo.

Struttura e trasformazione da oggetto a entità:

All'interno della struttura di JPA troviamo un layer che si occupa del operazione di mapping.

La struttura della tabella che si viene a creare da questa operazione è la seguente:

- Il **nome dell'entità** diventa il **nome della tabella**;
- Gli **attributi** diventano le **colonne**;
- I **tipi** vengono convertiti in modo molto **simile** a ciò che succede in **JDBC**.

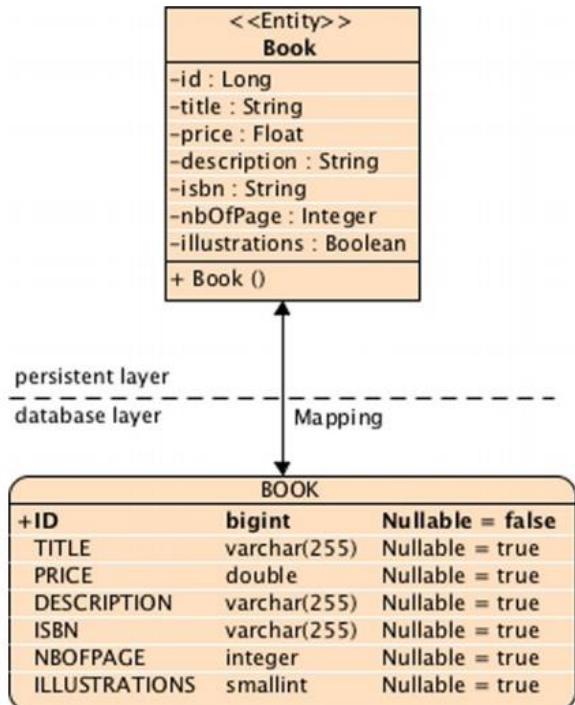
Abbiamo infine un file XML (**Persistens.xml**) dove vengono “nascoste” tutti quei dettagli che vengono nascosti al programmatore per garantire la trasparenza (Buttiamo tutte le cose brutte e nascoste sotto al tappeto in un file XML).

Dopo il **mapping** l'oggetto diventa semplicemente una riga della tabella.

Query e richieste: a questo punto le query non vanno più pensate su entità appartenenti a database ma su degli oggetti.

All'interno di Java EE le query vengono gestite tramite un **Entity Manager** che è quella parte del container (Braccio armato), che si occupa di gestire in prima persona gli oggetti resi persistenti. Per definire un **Entity manager**:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("chapter04PU");
EntityManager em = emf.createEntityManager();
em.persist(book);
```



1. Creare un oggetto **EntityManagerFactory**, passando come parametro una stringa che serve a importare le informazioni contenute all'interno del file XML.
2. Utilizzo la **Factory** per creare un **Entity Manager**, e lo utilizzo (tramite il metodo **persist**) per rendere persistente l'entità precedentemente creata book.

Definire query in Java EE: all'atto di definizione di una entità posso definire anche delle query sullo stesso oggetto. (si sente puzza di SQL, Cit. VS).

È importante notare come la **FROM** e la **SELECT**, facciano riferimento a degli oggetti e non a delle tabelle.

```
@Entity
@NamedQuery(name = "findBookH2G2", -->
             query = "SELECT b FROM Book b WHERE b.title = 'H2G2'")
public class Book {

    @Id @GeneratedValue
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;

    // Constructors, getters, setters
}
```

Un esempio completo:

1. Creiamo un libro, (**oggetto non persistente**).
2. Creiamo un **Entity Manager**.
3. Rendiamo persistente l'oggetto libro, tramite una transazione (operazioni che si svolge con ACID bla bla bla... è sempre lei). **Tx.begin()** da il via alla **zona rossa** della transazione, **em.persist(book)** rende il nostro oggetto persistente e **tx.commit()** verifica che la transazione sia svolta in modo corretto (altrimenti fa il **rollback**, ovvero riporta tutto allo stato iniziale).
4. Creo una query chiedendo **all'Entity Manager** di crearne una passandogli il nome e la posizione, successivamente la eseguo richiedendo il risultato.
5. Chiudo **Entity Manager e Factory**.

```
public class Main {  
  
    public static void main(String[] args) {  
  
        // 1-Creates an instance of book  
        Book book = new Book("H2G2", "The Hitchhiker's Guide to the Galaxy", 12.5F, ←  
                        "1-84023-742-2", 354, false);  
  
        // 2-Obtains an entity manager and a transaction  
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("chapter04PU");  
        EntityManager em = emf.createEntityManager();  
  
        // 3-Persists the book to the database  
        EntityTransaction tx = em.getTransaction();  
        tx.begin();  
        em.persist(book);  
        tx.commit();  
  
        // 4-Executes the named query  
        book = em.createNamedQuery("findBookH2G2", Book.class).getSingleResult();  
  
        // 5-Closes the entity manager and the factory  
        em.close();  
        emf.close();  
    }  
}
```

Ciò che sotto al tappeto si cela:

Fornire la persistenza ha nascosto molte informazioni utili all'interno di un file XML che chiamiamo **Persistence Unit**, ed è qui che l'Entity Manager va a prendere tutte le informazioni che gli servono per fare il suo lavoro come: le credenziali di accesso, l'indirizzo del DB e così via...

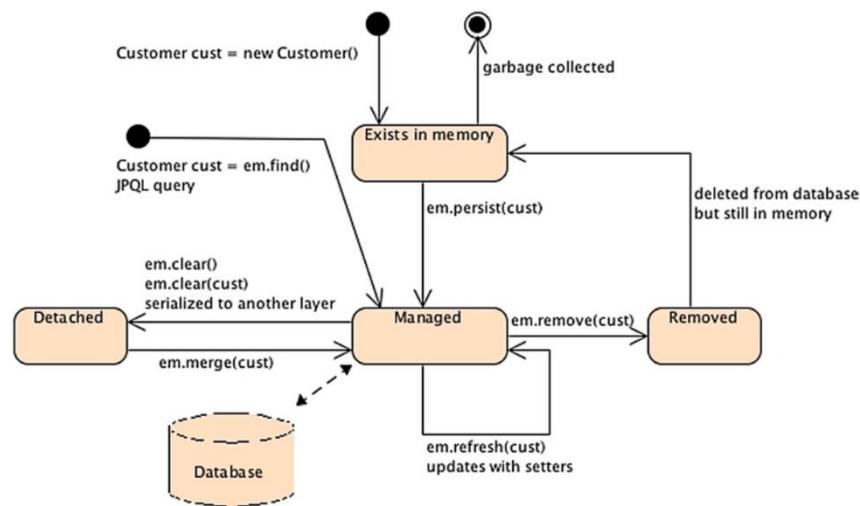
Anche in questo caso per fare modifiche semplicemente modificare questo file xml senza neanche dover interrompere l'esecuzione.

```
<?xml version="1.0" encoding="UTF-8"?>  
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"  
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence ←  
                     http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"  
             version="2.1">  
  
<persistence-unit name="chapter04PU" transaction-type="RESOURCE_LOCAL">  
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>  
    <class>org.agoncal.book.javaee7.chapter04.Book</class>  
    <properties>  
        <property name="javax.persistence.schema-generation-action" value="drop-and-create"/>  
        <property name="javax.persistence.schema-generation-target" value="database"/>  
        <property name="javax.persistence.jdbc.driver" →  
            value="org.apache.derby.jdbc.ClientDriver"/>  
        <property name="javax.persistence.jdbc.url" →  
            value="jdbc:derby://localhost:1527/chapter04DB;create=true"/>  
        <property name="javax.persistence.jdbc.user" value="APP"/>  
        <property name="javax.persistence.jdbc.password" value="APP"/>  
    </properties>  
    </persistence-unit>  
</persistence>
```

Senza queste specifiche un entità rimane un semplice indifeso POJO.

Ciclo di vita delle entità: (Alta probabilità) il ciclo di vita di un entità viene gestito dall'Entity Manager e si divide in 4 fasi:

- L'oggetto viene creato quindi esiste in memoria (stato di **Exists in memory**);
- L'oggetto viene preso in carico da un **Entity Manager** (stato di **Managed**), resta qui finché rimane persistente;
- L'oggetto rimosso (stato di **Removed**), attenzione viene rimosso dal gestore della persistenza, non viene eliminato, continua a esistere nella memoria centrale. Praticamente ridiventa un **POJO**.
- L'oggetto viene temporaneamente “abbandonato” (**Rimosso dal Persistence Context**, è comunque ancora nel DB solo che non viene gestito) ma non in modo definitivo, l'**Entity Manager** può tornare ad occuparsene (stato di **Detached**).

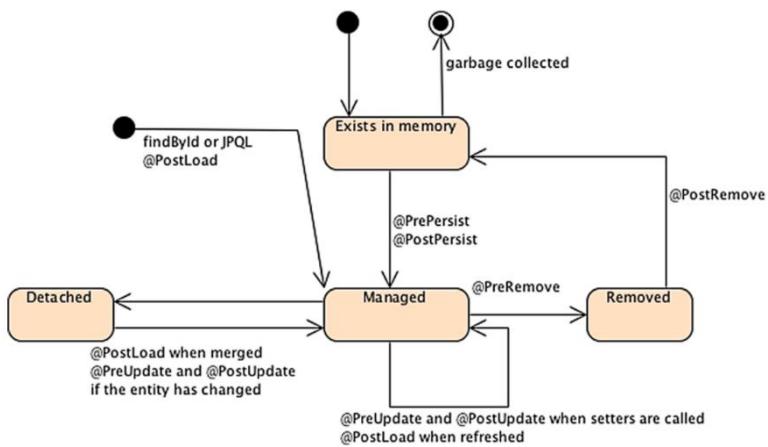


CallBacks: Come per gli oggetti in JPA è possibile inserire all'interno del ciclo di vita delle entità della logica di business intercettando le varie fasi del ciclo di vita tramite metodi **Pre** e **Post**.

Per esempio: Prima di inserire un entità all'interno del DB. L'Entity manager chiama il metodo annotato con **@PrePersist** se tutto va bene quello annotato con **@PostPersist** e così via.

```

@PrePersist
@PreUpdate
private void validate() {
    if (firstName == null || "".equals(firstName))
        throw new IllegalArgumentException("Invalid first name");
    if (lastName == null || "".equals(lastName))
        throw new IllegalArgumentException("Invalid last name");
}
  
```



Listeners: sono utilizzati per separare la logica di business da una specifica classe in modo che possa essere condivisa da tutte le classi. Per esempio, il metodo validate di prima diventa:

Una classe per essere utilizzata come listener deve:

- Avere un **costruttore pubblico senza argomenti**.
- La firma dei metodi di callback deve essere diversa tra di loro.
- Deve avere parametri di **tipo compatibile** con le entità che l'evento gli passerà.

```

public class DataValidationListener {

    @PrePersist
    @PreUpdate
    private void validate(Customer customer) {
        if (customer.getFirstName() == null || "".equals(customer.getFirstName()))
            throw new IllegalArgumentException("Invalid first name");
        if (customer.getLastName() == null || "".equals(customer.getLastName()))
            throw new IllegalArgumentException("Invalid last name");
    }
}
  
```

([Approfondimento pagina 221 a 224 libro](#)).

Validazione dei bean: serve a garantire la formattazione dei dati inseriti nel DB (come campi non nulli o formattati in un determinato modo) ([pagina 112 libro](#)).

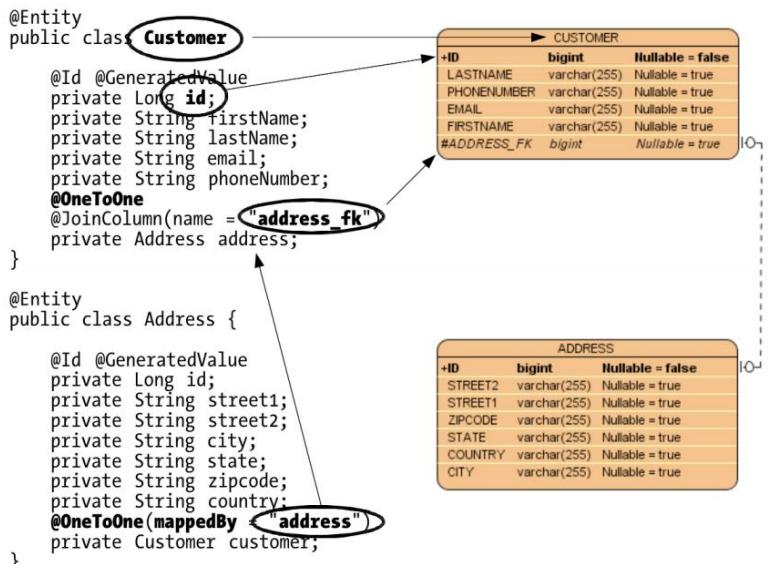
Elementary Mapping (@Tables, @SecondaryTables, @id, @GeneratedValue) da pagina 125 a pagina 128

Relationship Mapping: all'interno di JPA è possibile gestire tutto quello che riguarda le relazioni tra diverse entità (**chiavi esterne, join, relazioni bidirezionali o unidirezionali e cardinalità**), tramite delle specifiche annotazioni.

Gestione della cardinalità: la cardinalità all'interno di JPA viene gestita attraverso 4 annotazioni, che permettono di realizzare sia annotazioni unidirezionali che bidirezionali:

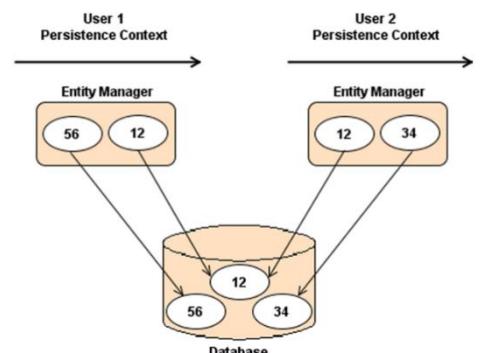
- **@OneToOne;**
- **@OneToMany;**
- **@ManyToOne;**
- **@ManyToMany;**

l'esempio a destra rappresenta una relazione bidirezionale tra **Customer** e **Address**. Se invece volessi creare un associazione di tipo bidirezionale mi basterebbe inserire la **FK** solo nella classe **Customer** ([Altri esempi da pagina 154 a 161](#)).



Entity Manager nel dettaglio: è la parte centrale di JPA e si occupa principalmente di gestire il ciclo di vita delle entità, le query e il persistence context.

Concetto di persistence context: è una sorta di cache dove l'Entity manager, “archivia” le entità prima che siano realmente inserita (**flush**), all'interno del database. È importante notare che all'interno del persistence context non possono esistere entità con lo stesso identificativo (**id**). Inoltre, come possiamo vedere nella figura a destra, se un entità è presente in due persistence context e viene flushata all'interno del db contemporaneamente, una delle due operazioni fallirà perché banalmente non possiamo scrivere contemporaneamente un oggetto (**sono transazioni**).



Metodi del Entity Manager: come ogni oggetto l'Entity manager espone diversi metodi che ci permettono di gestire i nostri oggetti persistenti.

Persist(): rende persistente un oggetto, ovvero inserisce i dati nel database quando non esistono ancora all'interno di esso. L'operazione viene svolta tramite una **transazione**:

```

Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
Address address = new Address("Rutherford Rd", "London", "8QE", "UK");
customer.setAddress(address);

tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();

assertNotNull(customer.getId());
assertNotNull(address.getId());

```

Cercare un'entità all'interno del database: per fare ciò esistono due metodi:

- **Find():** cerca l'entità all'interno del database tramite un identificativo (**@id**), caricandone in memoria sui il riferimento che i dati associati a quest'ultimo (quindi tutto l'oggetto).

```
Customer customer = em.find(Customer.class, 1234L)
if (customer!= null) {
    // Process the object
}
```

- **GetReference():** simile al metodo find tranne per il fatto che una volta trovato l'oggetto all'interno del db getReference carica in memoria solo il riferimento a quest'ultimo caricando i dati solo nel momento in cui questo vengono utilizzati (**lazy fetching**).

```
try {
    Customer customer = em.getReference(Customer.class, 1234L)
    // Process the object
} catch(EntityNotFoundException ex) {
    // Entity not found
}
```

Rimuovere un entità all'interno del database: per fare ciò si utilizza il metodo **Remove()**, che rimuove per l'appunto l'entità del DB e la rende **detached** all'interno dell'Entity manager (**l'entità non viene eliminata dalla memoria, infatti torna ad essere un POJO, viene eliminata solo quando il GC la accoppa**).

Rimozione degli orfani: con il temine **orfano** si fa riferimento a righe che non sono referenziate da nessuna altra tabella. JPA è in grado di rimuovere automaticamente tali entità se le viene esplicitamente chiesto in fase di definizione dell'oggetto (**cancellazione e modifica a cascata**).

```
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();

tx.begin();
em.remove(customer);
tx.commit();

// The data is removed from the database but the object is still accessible
assertNotNull(customer);
```

`@OneToOne (fetch = FetchType.LAZY, orphanRemoval=true)`

Sincronizzazione con il database: la prima sincronizzazione dei dati con il database avviene al primo **commit**, successivamente, JPA aggiorna automaticamente lo stato dell'istanza salvata nel db, ma per alcuni tipi di operazioni questo non è detto che sia sufficiente per questo sono stati definiti i seguenti metodi:

Flush(): forza il persistence provider a inserire i dati all'interno del database.

Nell'esempio a destra succedono cose, il **flush** infatti non aspetta la fine della transazione. E l'inserimento forzato all'interno del DB del Customer, prima che venga creato e committato address viole le regole della **FK**.

```
tx.begin();
em.persist(customer);
em.flush();
em.persist(address);
tx.commit();
```

Refresh(): è l'opposto del **flush**, praticamente sovrascrive lo stato attuale dell'oggetto caricato in memoria con lo stato dell'entità associata presente nel DB (**DB -> Oggetto in memoria**).

Gestire il contenuto del Persistence Context:

Contains(): verifica se una data entità si trova all'interno del mio **Persistence Context** ritornando un booleano;

Clear(): svuota il persistence context facendo decadere tutte le entità contenute all'interno di esso.

```
assertTrue(em.contains(customer));
em.detach(customer);
assertFalse(em.contains(customer));
```

Detach(): rimuovi del **Persistent Context** facendolo diventare detached (decaduto);

Merge(): Ricarica lo stato un entità all'interno del persistence context, in modo che ritorni ad essere gestito (in seguito a una clear per esempio).

```
tx.begin();
em.persist(customer);
tx.commit();

em.clear();

// Sets a new value to a detached entity
customer.setFirstName("William");

tx.begin();
em.merge(customer);
tx.commit();
```

Eventi a cascata: è possibile all'interno di JPA definire aggiornamenti e eliminazioni a cascata. Per esempio, all'interno della classe **Customer** troviamo la seguente annotazione:

```
@OneToOne (fetch = FetchType.LAZY, cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
@JoinColumn(name = "address fk")
```

JPQL (Java Persistence Query Language): (**Alta probabilità**) nasce dalla necessità di dover gestire tutte quelle operazioni complesse che riguardano la gestione di dati all'interno di un database attraverso JPA. Potente come SQL e molto simile a quest'ultimo ha come unica differenza quella di lavorare con le entità e non con le tabelle. **JPQL** traduce le query in **SQL**, la traduzione di questa query è un overhead, operazione che ad un primo sguardo potrebbe costare risorse se non fosse per il fatto che viene eseguita a tempo di compilazione o meglio a tempo di **deployment** (le query vengono tradotte in base al database scelto).

All'interno di JPQL le query vengono suddivise in 5 categorie:

```
SELECT b
FROM Book b
WHERE b.title = 'H2G2'
```

```
SELECT <select clause>
FROM <from clause>
[WHERE <where clause>]
[ORDER BY <order by clause>]
[GROUP BY <group by clause>]
[HAVING <having clause>]
```

- **Query dinamiche**: generate dinamicamente all'interno di un applicazione ed eseguite. È la tipologia di query più utilizzata nonché la più versatile anche se ha degli svantaggi (non è possibile verificare sintatticamente il codice SQL all'interno di esse):

```
String jpqlQuery = "SELECT c FROM Customer c";
if (someCriteria)
    jpqlQuery += " WHERE c.firstName = 'Betty'";
query = em.createQuery(jpqlQuery);
List<Customer> customers = query.getResultList();
```

- **Named query**: sono query statiche, inserite all'interno di una classe all'atto della sua creazione.

Vengono utilizzate per compiti standard. Non possono essere modificate ma gli si può passare un parametro.

```
@Entity
@NamedQueries({
    @NamedQuery(name = "findAll", query="select c from Customer c"),
    @NamedQuery(name = "findVincent", query="select c from Customer c where c.firstName = 'Vincent'"),
    ...})
```

- **Criteria API (or Object-Oriented Queries)**: Sono query cosiddette orientate agli oggetti, uniscono le Named e le Dinamiche e hanno il grande vantaggio che la sintassi viene verificata a tempo di

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = builder.createQuery(Customer.class);
Root<Customer> c = criteriaQuery.from(Customer.class);
criteriaQuery.select(c).where(builder.equal(c.get("firstName"), "Vincent"));
Query query = em.createQuery(criteriaQuery).getResultList();
List<Customer> customers = query.getResultList();
```

- **Native query:** sono query native del SQL, per i nostalgici e figli dei fiori (goccia di sudore quando il capo entra perché potrebbe voler cambiare il db e so cazzo). Non si usano a meno che di casi molto rari e particolari.

```
Query query = em.createNativeQuery("SELECT * FROM t_customer", Customer.class);
List<Customer> customers = query.getResultList();
```

- **Stored procedure query:** sono memorizzate all'interno del database. Usate per compiti lunghi e ripetuti, oltre che per scopi statistici. Hanno il grande vantaggio che il codice è centralizzato e può essere utilizzato da diversi programmi.

```
CREATE PROCEDURE sp_archive_books @archiveDate DATE, @warehouseCode VARCHAR AS
    UPDATE T_Inventory
    SET Number_Of_Books_Left = 1
    WHERE Archive_Date < @archiveDate AND Warehouse_Code = @warehouseCode;

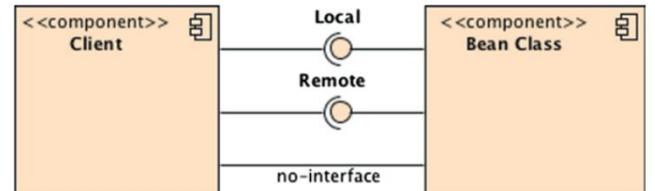
    UPDATE T_Transport
    SET Warehouse_To_Take_Books_From = @warehouseCode;
END
```

È importante notare che la query è un oggetto che deriva dalla classe query ([Altri esempi da pagina 198 a 208](#)).

Lezione 20, Enterprise JavaBeans, parte 1:

Enterprise JavaBeans (EJB): è un componente di JEE **server-side** che si occupa di incapsulare la logica di business e di gestire transazioni e sicurezza. All'interno di JEE vengono definiti tre tipi di sessioni EJB:

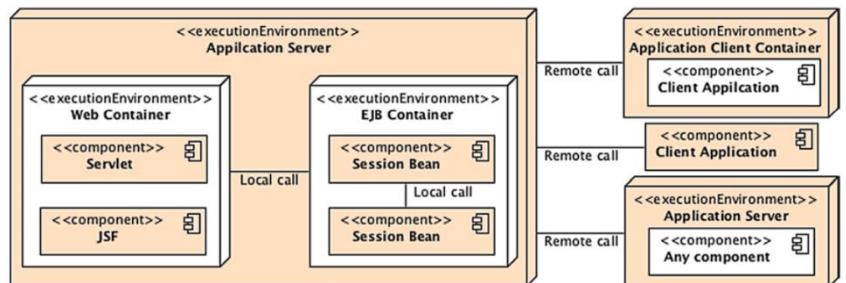
- **Stateless:** Il bean non viene instanziato né usato da nessun client e non ci sono conversazioni di stato fra i metodi. Praticamente fornisce solo servizi.
- **Stateful:** l'opposto di stateless, viene utilizzato per operazioni che dividono in più step.
- **Singleton:** è un bean che viene condiviso dai client e che supporta la concorrenza di accesso.



Un EJB è composto da due elementi principali:

- **La classe bean:** che contiene l'implementazione dei metodi di business e che può implementare nessuna o diverse interfacce. L'importante è che il bean sia annotato con @stateless, @stateful o @singleton.
- **Un'interfaccia di business:** che contiene la dichiarazione dei metodi di business che sono visibili al client e implementati nella classe del bean. Un bean può avere un'interfaccia locale, remota o nessuna.

Interfacce remote, locali o nessuna: come abbiamo detto un beans può non implementare un interfaccia. Per gli altri due casi l'utilizzo di un interfaccia remota o locale dipende dal tipo di chiamata che si sta andando a effettuare.



Esistono inoltre delle annotazioni per definire il tipo di interfaccia che si sta andando a implementare:

- **@Remote:** Denota un interfaccia remota di business. I parametri sono passati per valore e necessitano di essere serializzati.
- **@Local:** Denota un interfaccia locale. I parametri sono passati per referenza dal client al bean.

La definizione di queste annotazioni permette di specificare quale interfaccia utilizzare in quale caso, come nell'esempio a destra. ([Altri esempi pagina 236](#)).

Portable JNDI Name ([da pagina 237 a 239](#)).

```
@Local
public interface ItemLocal {
    List<Book> findBooks();
    List<CD> findCDs();
}

@Remote
public interface ItemRemote {
    List<Book> findBooks();
    List<CD> findCDs();
    Book createBook(Book book);
    CD createCD(CD cd);
}

@Stateless
public class ItemEJB implements ItemLocal, ItemRemote {
    // ...
}
```

Stateless Beans: sono i beans più usati all'interno di JEE. Hanno la particolarità che ogni operazione deve essere completata all'interno di una singola chiamata a un metodo. In parole poche offrono dei servizi che possono essere utilizzati senza che nulla venga instanziato:

Per esempio, creiamo un oggetto book e poi lo diamo a un bean **stateless** che lo rende persistente.

```
Book book = new Book();
book.setTitle("The Hitchhiker's Guide to the Galaxy");
book.setPrice(12.5F);
book.setDescription("Science fiction comedy series created by Douglas Adams.");
book.setIsbn("1-84023-742-2");
book.setNbOfPage(354);
statelessService.persistToDatabase(book);
```

Questo tipo di architettura permette di riutilizzare un bean su più client che si trovano ad eseguire le medesime operazioni. Per fare ciò il container crea per ogni **stateless EJB** un certo numero di istanza (**Pool**) in memoria condividendole tra i client, siccome questo tipo di bean non ha uno stato ognuna di queste copie in memoria è equivalente.

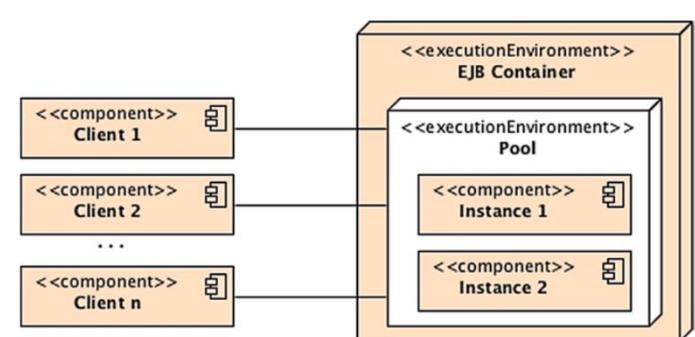
([Altri esempi pagina 240/241](#)).

Stateful Beans: sono beans che offrono metodi di business ai loro client ma non mantengono una conversazione con quest'ultimi. Sono di solito utilizzati in operazioni che si dividono in varie fasi, fungendo un po' da "memoria". Un ottimo esempio è quello della gestione di un carrello per un sito di eCommerce.

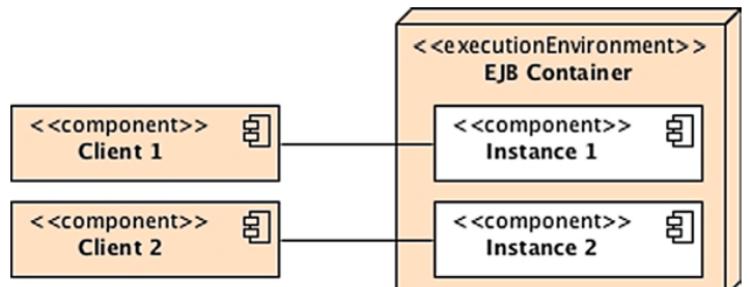
In questo esempio infatti, ogni volta che aggiungiamo un libro, lo passiamo a uno **stateful beans**, che lo inserisce all'interno di un carrello e alla fine fa il checkout.

Al contrario dei beans **stateless**, quelli **stateful** non possono essere condivisi tra più client, in quanto "personalizzati" o meglio in possesso di uno stato. Per questo motivo il container crea, per ogni client che ne ha bisogno, uno stateful bean **personale**.

([Implementazione del bean a pagina 243](#)).



```
Book book = new Book();
book.setTitle("The Hitchhiker's Guide to the Galaxy");
book.setPrice(12.5F);
book.setDescription("Science fiction comedy series created by Douglas Adams.");
book.setIsbn("1-84023-742-2");
book.setNbOfPage(354);
statefulComponent.addBookToShoppingCart(book);
book.setTitle("The Robots of Dawn");
book.setPrice(18.25F);
book.setDescription("Isaac Asimov's Robot Series");
book.setIsbn("0-553-29949-2");
book.setNbOfPage(276);
statefulComponent.addBookToShoppingCart(book);
statefulComponent.checkOutShoppingCart();
```



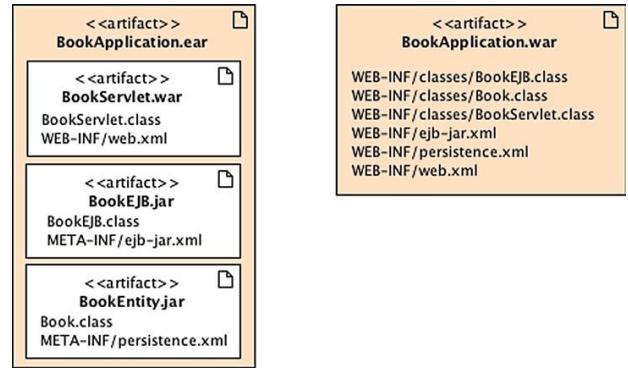
Singleton: Sono dei bean di sessione che vengono istanziati una sola volta per applicazione (tramite un pattern visto in precedenza, java EE basta annotare con **@singleton**).

Ovviamente vista la natura di questa categoria di beans, il container può instanziare un unico singleton che verrà poi condiviso tra tutti i client.



([Implementazione del bean a pagina 244/245/246](#)).

Packaging: come quasi tutte le componenti Java EE i beans necessitano di essere **impacchettati** (all'interno di un file **.jar** che contiene tutte le componenti utili al bean) prima di essere pronti al **deployment**. L'unione di tutti i vari bean e componenti viene inserito in un file **.ear** (contiene appunto più moduli) che è pronto al **deployment**:



Invocazione di EJB: un enterprise java beans può essere invocato da diversi tipi di componenti Java EE: un bean, un POJO, un Web Service, un Client Grafico. Tramite tre diversi tipi di invocazione:

Tramite iniezione: con questo metodo i invocazione la situazione varia in base a sé il bean dispone o meno di un **interfaccia**:

- Se il bean non ha interfacce quest'ultimo viene importato tramite un'annotazione:
- Se il bean dispone di numerose interfacce il client deve specificare quale vuole che sia referenziata:

```
@Stateless
public class ItemEJB {...}

// Client code injecting a reference to the EJB
@EJB ItemEJB itemEJB;
```

```
@Stateless
@Remote(ItemRemote.class)
@Local(ItemLocal.class)
@LocalBean
public class ItemEJB implements ItemLocal, ItemRemote {...}

// Client code injecting several references to the EJB or interfaces
@EJB ItemEJB itemEJB;
@EJB ItemLocal itemEJBLocal;
@EJB ItemRemote itemEJBRremote;
```

In particolare, se il bean risiede su un **server**

differente le **@EJB API** permettono di inserire come attributo l'indirizzo dell'interfaccia o del bean direttamente:

```
@EJB(lookup = "java:global/classes/ItemEJB") ItemRemote itemEJBRremote;
```

Invocazione con CDI (Context Dependecy Injection): nella maggior parte dei casi possiamo sostituire l'annotazione **@EJB** con **@Inject** questo ci permette di ottenere tutti i vantaggi della **Context Dependecy Injection**.

```
@Stateless
@Remote(ItemRemote.class)
@Local(ItemLocal.class)
@LocalBean
public class ItemEJB implements ItemLocal, ItemRemote {...}

// Client code injecting several references to the EJB or interfaces with @Inject
@Inject ItemEJB itemEJB;
@Inject ItemLocal itemEJBLocal;
@Inject ItemRemote itemEJBRremote;
```

Inoltre, per quanto riguarda l'iniezione di beans remoti visto che l'annotazione **@Injection** non prevede parametri dobbiamo prima produrre il bean remoto:

```
// Code producing a remote EJB
@Produces @EJB(lookup = "java:global/classes/ItemEJB") ItemRemote itemEJBRremote;

// Client code injecting the produced remote EJB
@Inject ItemRemote itemEJBRremote;
```

Invocazione diretta con JNDI: è un meccanismo di naming ispirato a RMI, rappresenta un'alternativa all'iniezione, permettendoci di risparmiare risorse, infatti, **JNDI** inserisce i dati solo se sono necessari.

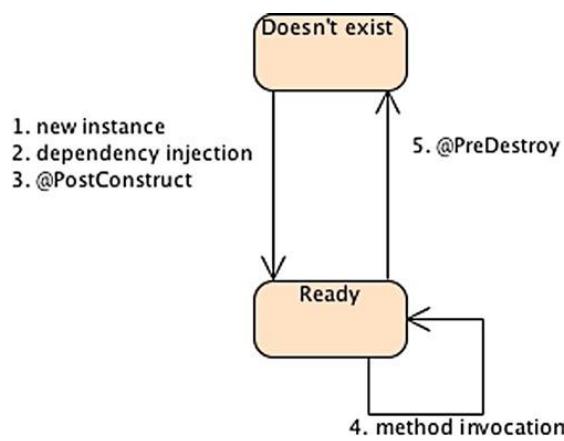
```
Context ctx = new InitialContext();
ItemRemote itemEJB = (ItemRemote) ctx.lookup("java:global/cdbookstore/ItemEJB!org.agoncal.book.javaee7.ItemRemote");
```

Lezione 21, Enterprise JavaBeans, parte 2:

Le massime del prof: Evita di scrivere codice già esistente e testato per fare il figo, perché non sei nessuno, al massimo inizierai a diventare qualcuno solo dopo 5 anni di esperienza.

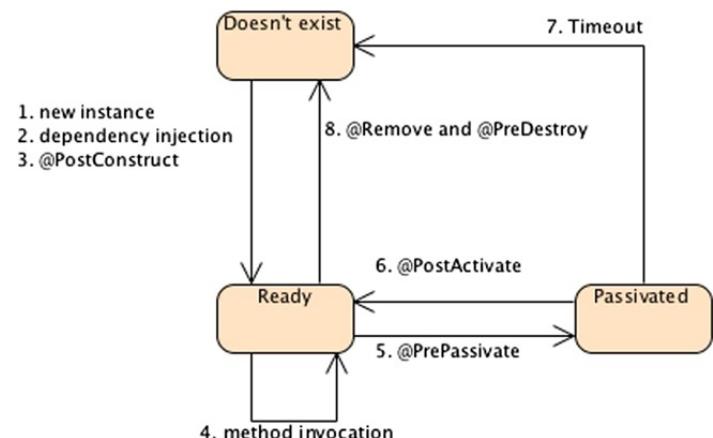
Ciclo di vita dei Session Beans: (**Possibile domanda teoria allo scritto**) per quanto riguarda gli **stateless beans** e i **singleton** il ciclo di vita è molto semplice:

1. Tutto inizia quando il beans viene richiesto (tramite **injection** o **JNDI lookup**) e il container instanzia una nuova sessione.
2. Se l'istanza appena creata utilizza a sua volta delle **injection**, il container soddisfa anche queste.
3. Se l'istanza implementa dei metodi **@PostConstruct**, vengono eseguiti dal container.
4. Il bean va nello stato di **ready** dove in caso di necessità esegue le richieste dei client, rimanendo in attesa quando non ne riceve.
5. Il container non ha più necessità del bean. Quindi invoca i metodi **@PreDestroy** se ce ne sono e poi elimina il bean.



Per quanto riguarda gli **stateful beans**, il ciclo è un po' diverso:

1. Tutto inizia quando il beans viene richiesto da un client (tramite **injection** o **JNDI lookup**) e il container instanzia una nuova sessione.
2. Se l'istanza appena creata utilizza a sua volta delle **injection**, il container soddisfa anche queste.
3. Se l'istanza implementa dei metodi **@PostConstruct**, vengono eseguiti dal container.
4. Il bean esegue le richieste del client e va in memoria in attesa di richieste future (stato di **ready**).
5. Se il client rimane inattivo per un periodo di tempo il bean il container si prepara a renderlo **passivo** eseguendo prima i metodi annotati con **@PrePassivate**, successivamente lo sposta nella memoria di massa.
6. Se il client invoca un bean in stato **passivo** il container prima di risveglierlo esegue i metodi annotati con **@PostActivate**, per poi spostarlo nella memoria centrale.
7. Se il client non invoca un bean **passivo** per un certo periodo di tempo scatta un **timeout** e il bean viene distrutto.
8. In alternativa se il client invoca il metodo annotato con **@Remove** il container esegue i metodi annotati con **@PreDestroy** e distrugge il bean.



Autorizzazioni: (confronto tra le due tipologie di autenticazione) permettono di controllare l'accesso al business code, esistono due modi di gestire le autorizzazioni in java EE:

Dichiarativa: Possono essere definite utilizzando **annotazioni** o un **XML Deployment descriptor**.

(Esempi pagina 275/276/277).

| | Annotation | Bean | Method | Description |
|--|---------------|------|--------|---|
| | @PermitAll | X | X | Indicates that the given method (or the entire bean) is accessible by everyone (all roles are permitted). |
| | @DenyAll | X | X | Indicates that no role is permitted to execute the specified method or all methods of the bean (all roles are denied). This can be useful if you want to deny access to a method in a certain environment (e.g., the method launchNuclearWar() should only be allowed in production but not in a test environment). |
| | @RolesAllowed | X | X | Indicates that a list of roles is allowed to execute the given method (or the entire bean). |
| | @DeclareRoles | X | | Defines roles for security checking. |
| | @RunAs | X | | Temporarily assigns a new role to a principal. |

Programmatica: permettono di realizzare un controllo sulle autorizzazioni a grana più fine, utilizzando due metodi forniti dall'interfaccia **SessionContext**:

- **isCallerInRole()**: Ritorna un booleano e controlla se il chiamante ha un ruolo assegnato.
- **getCallerPrincipal()**: ritorna **java.security.Principal** che identifica il chiamante.

```

@Stateless
public class ItemEJB {
    @PersistenceContext(unitName = "chapter08PU")
    private EntityManager em;

    @Resource
    private SessionContext ctx;

    public void deleteBook(Book book) {
        if (!ctx.isCallerInRole("admin"))
            throw new SecurityException("Only admins are allowed");
        em.remove(em.merge(book));
    }

    public Book createBook(Book book) {
        if (ctx.isCallerInRole("employee") && !ctx.isCallerInRole("admin"))
            book.setCreatedBy("employee only");
        else if (ctx.getCallerPrincipal().getName().equals("paul"))
            book.setCreatedBy("special user");
        em.persist(book);
        return book;
    }
}

```

Transazioni: (**Il pane quotidiano dello Scarano**) sono operazioni che devono essere eseguite in sequenza e devono essere completate o annullate per non andare a compromettere la consistenza dei dati. Il classico esempio è quello di un pagamento in banca, se rimuovo i soldi da un account devo essere sicuro che un altro account li riceva non posso perderli per la strada perché un'operazione non è andata a buon fine. Le transazioni si basano su 4 proprietà **ACID**:

- **Atomicità**: una transazione è composta da una o più operazioni raggruppate in un'unità di lavoro, alla fine della transazione o tutte le operazioni sono andate a buon fine (**commit**) o si devono annullare quelle che sono state eseguite (**rollback**).
- **Consistenza**: alla fine della transazione tutti i dati in uscita devono essere consistenti.
- **Isolata**: gli stadi intermedi della transazione non devono essere visibili esternamente all'applicazione.
- **Durabilità**: deve essere finita, ovvero una volta che si è conclusa i cambiamenti nei dati devono essere visibili alle altre applicazioni.

All'interno di JEE le transazioni vengono interamente e automaticamente gestite dal container in ogni loro parte.

Nell'esempio a sinistra vediamo come la gestione delle transazioni è del tutto invisibile all'utente, infatti, sotto nel diagramma di sequenza vediamo cosa realmente nel container accade all'invocazione del metodo **createBook()**.

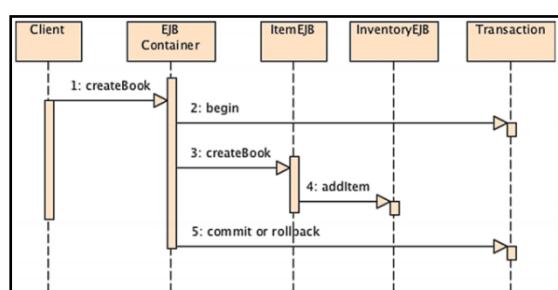
```

@Stateless
public class ItemEJB {
    @PersistenceContext(unitName = "chapter09PU")
    private EntityManager em;
    @Inject
    private InventoryEJB inventory;

    public List<Book> findBooks() {
        TypedQuery<Book> query = em.createNamedQuery(FIND_ALL, Book.class);
        return query.getResultList();
    }

    public Book createBook(Book book) {
        em.persist(book);
        inventory.addItem(book);
        return book;
    }
}

```

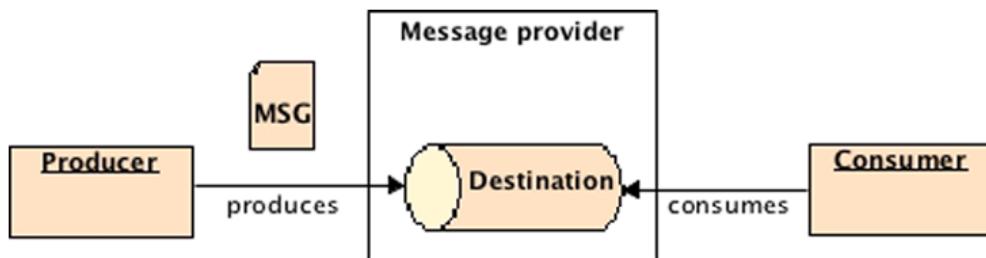


Lezione 23/24, Java Message Service: è un servizio che permette di scambiare messaggi all'interno di JEE. Questo servizio si basa sul concetto di **broker** o **provider di posta**, una struttura software che offre due principali macro-vantaggi:

- **Disaccoppiamento temporale dall'invio del messaggio alla sua ricezione** (Come per WhatsApp, dove se il destinatario non è online il messaggio rimane in attesa, sempre per il discorso che le cose fortemente accoppiate non ci piacciono in JEE).
- **Implementazione di meccanismi di connessione molto flessibili**, che riducono di molto i compiti del programmatore.

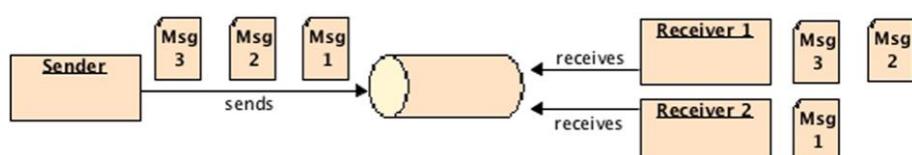
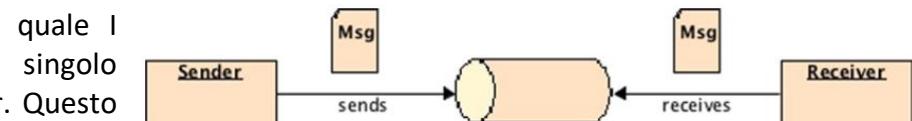
Message Oriented Middleware MOM: si basano sull'utilizzo di un broker o provider di posta (**Sistema asincrono**), che tiene "in caldo" i messaggi prima di spedirli al destinatario finale.

Il producer è la componente che "produce" il messaggio e lo inserisce all'interno di una **destinazione**, che rappresenta una sorta di casella della posta (ha una posizione fissa, non mi frega il consumer con cui voglio comunicare dove sta), dove il **consumer**, ovvero colui che "consuma" il messaggio sa di dover andare a controllare la presenza di nuovi messaggi.

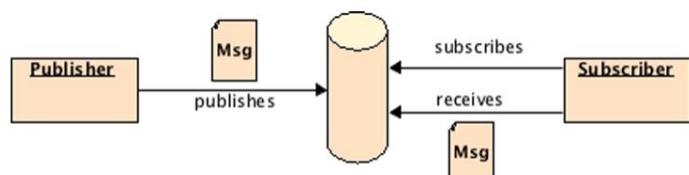


JMS è il set di API che si occupa di implementare questo sistema. In JMS esistono due tipi di destinazione ognuna applicata a uno specifico modello di messaggistica:

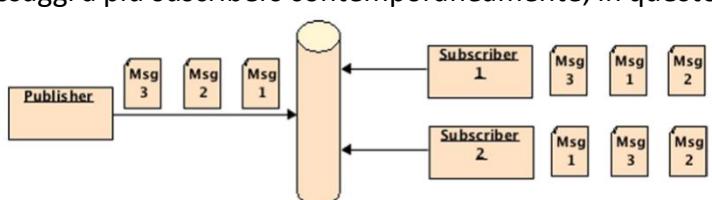
- **Point to point:** all'interno del quale i messaggi viaggiano da un singolo producer a un singolo consumer. Questo modello si basa sul concetto di **coda**. Viene anche utilizzato nel caso in cui un producer si trovi a dover inviare tanti messaggi a diversi consumer (**Multiple receivers**).



- **Publish-Sunbscribe:** in questo modello un singolo messaggio è inviato da un singolo producer per diversi consumer (viene chiamato **suscribers**, perché si deve prima iscrivere al **Topic** per ricevere messaggi).



Ovviamente un singolo publisher può inviare messaggi a più subscribers contemporaneamente, in questo caso c'è però una **dipendenza temporale**, visto che i messaggi vengono consegnati solo dopo l'iscrizione (quelli prima vengono persi):



<< Se la prima parola è "Coda" la seconda non è per forza "Stack" !!! >>

È importante capire che i messaggi possono anche trasportare job, per esempio nei vecchi esercizi sui thread, quando dividevamo il lavoro fra i thread.

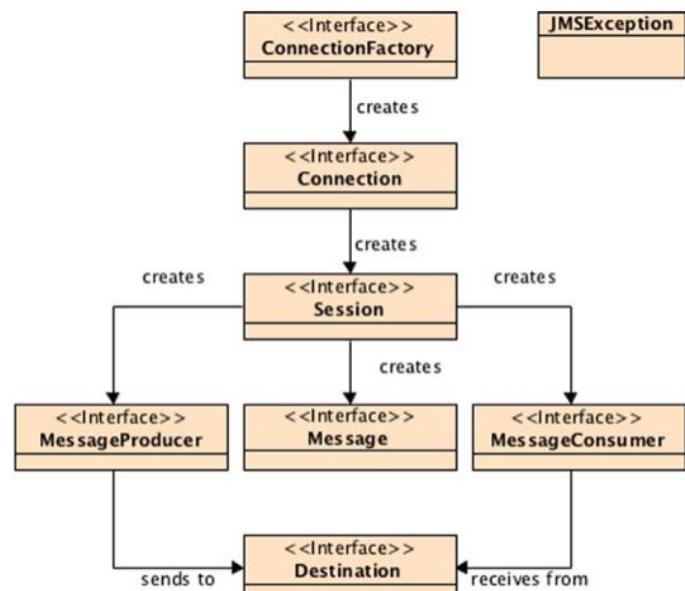
Java Messaging Service API in dettaglio:

ConnectionFactory: è un **administered object** (un oggetto che possono essere creato e gestiti solo dall'amministratore). Permette a un'applicazione di creare un oggetto **Connection**.

```
Context ctx = new InitialContext();
ConnectionFactory ConnectionFactory = (ConnectionFactory) ctx.lookup("jms/javaee7/ConnectionFactory");
```

Destination: è un altro **administered object**, che contiene tutte le informazioni sui provider come l'indirizzo di destinazione. Informazioni che sono nascoste ai client.

```
Context ctx = new InitialContext();
Destination queue = (Destination) ctx.lookup("jms/javaee7/Queue");
```



Connection: incapsula una connessione a un provider JMS. È una struttura **thread-safe** disegnata per essere condivisibile, tramite il concetto di **sessione**. Prima di essere utilizzata una connessione deve essere avviata con il metodo **.start()** (chiusa con il metodo **stop()**).

Sessioni: Viene creata tramite il metodo `Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);`. Il primo parametro specifica se la sessione implementa o meno le transazioni (se settato a **true** vuol dire che i messaggi non saranno inviati finché non sarà dato il **commit()**, se è settato su **false()** appena viene dato il metodo **send()** il messaggio viene inviato).

Messaggi: sono oggetti in cui vengono incapsulate informazioni, sono divisi in tre parti:

- **Header**: contiene informazioni sul messaggio, che client e provider utilizzano per identificare il messaggio. ([Tabella pagina 427](#)).
- **Properties**: è un meccanismo per inserire altri campi header all'interno del messaggio tramite applicazioni. ([Pagina 427](#)).
- **Bodies**: contiene i dati che si desidera inviare. È un campo molto versatile che può contenere molti dati differenti. ([Pagina 427/428](#)).



([Esempio di codice da pagina 430 a pagina 438](#)).

Differenza tra Properties e Header: (**Alta probabilità**) Le properties sono definibili (posso per esempio definire il colore di un maglione come blue) e possono essere aggiunte. Gli header invece non possono né essere aggiunti né specificati dal programmatore, perché standard e presenti all'interno di JMS. Inoltre, con le proprietà è il broker a effettuare i controlli, per esempio, se un messaggio non rientra nelle proprietà che desideriamo, non viene proprio inviato, questo porta a un notevole risparmio di risorse, visto che il client ha un notevole carico di lavoro in meno.

Meccanismi di affidabilità del broker: JMS assicura un certo numero di servizi di affidabilità atti a garantire un recapito affidabile.

- **Filtering messages:** permette di creare dei filtri, in modo che il broker invii al client solo messaggi che rispettano determinati requisiti.


```
context.createConsumer(queue, "JMSPriority < 6").receive();
context.createConsumer(queue, "JMSPriority < 6 AND orderAmount < 200").receive();
context.createConsumer(queue, "orderAmount BETWEEN 1000 AND 2000").receive();

context.createTextMessage().setIntProperty("orderAmount", 1530);
context.createTextMessage().setJMSPriority(5);
```
- **Time-to-live:** permette di definire un tempo (in millisecondi) dopo il quale il broker, elimina i messaggi dalla destinazione se non sono stati ancora consegnati.


```
context.createProducer().setTimeToLive(1000).send(queue, message);
```
- **Message Persistence:** JMS supporta due metodi di consegna per i messaggi:
 - **Persistent:** si assicura che il messaggio sia recapitato una sola volta (Più costoso in termini di risorse).
 - **Nonpersistent:** si assicura che il messaggio sia recapitato ma non controlla quante volta.

```
context.createProducer().setDeliveryMode(DeliveryMode.NON_PERSISTENT).send(queue, message);
```
- **Controllo degli Acknowledgment:** permette di ricevere una notifica dal consumer, di avvenuta consegna del messaggio. Nelle **sessioni transazionali**, gli ack vengono automaticamente utilizzati (servono in caso di roll-back). Nelle **sessioni non transazionali**, devono invece essere specificati ([Pagina 440](#)).


```
Producer -->|sends|> Queue (Msg)
Queue -->|receives|> Consumer (Msg)
```
- **Durable Consumer:** permette (all'interno del modello **pub-sub**) al consumer di ricevere i messaggi anche nel caso in cui non sia connesso al topic quando quei messaggi vengono inviati. (è molto oneroso in termini di risorse, ma alcune volte necessario).


```
context.createDurableConsumer(topic, "javaee7DurableSubscription").receive();
```
- **Settare una priorità:** permette di settare una priorità ai messaggi, in modo che vengano inviati prima di altri con priorità più bassa (valore da 0 a 9).


```
context.createProducer().setPriority(2).send(queue, message);
```

Tutte questi metodi possono ovviamente essere concatenati per definire un contesto:

```
context.createProducer().setPriority(2)
.setTimeToLive(1000)
.setDeliveryMode(DeliveryMode.NON_PERSISTENT)
.send(queue, message);
```

Message-Driven Beans: è un consumatore di messaggi asincrono, viene invocato dal container quando arriva un messaggio.

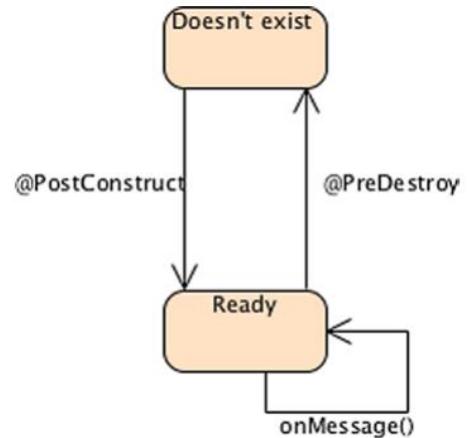
Hanno i seguenti requisiti:

```
@MessageDriven(mappedName = "jms/javaee7/Topic")
public class BillingMDB implements MessageListener {
    public void onMessage(Message message) {
        System.out.println("Message received: " + message.getBody(String.class));
    }
}
```

- Annotato con **@javax.ejb.MessageDriven** o l'equivalente nel deployment descriptor.
- Implementa l'interfaccia **MessageListener**.
- Classe pubblica non final o abstract.
- Costruttore pubblico e senza argomenti (perché il container non sa che argomenti passargli).
- Non devono essere definite metodi **finaline()**.

([Esempi e API da pagina 442 a 444](#)).

Ciclo di vita di un MDB: molto semplice, si passa dallo stato di esistenza a quello di pronto, tramite l'esecuzione dei metodi annotati con `@PostConstruct`, si rimane nello stato di pronto finché arrivano messaggi, si eseguono i metodi annotato con `@PreDestroy` e poi si elimina il bean.



([Altre semplice cose da pagina 445/447](#)).

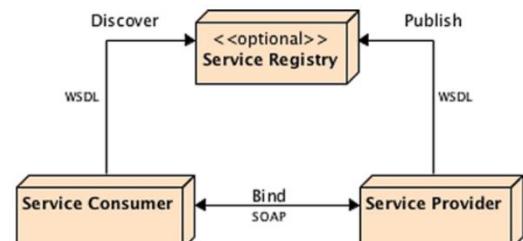
Lezione 26, Web Service and SOAP (è il tema finale di tutto il corso): Non sono altro che applicativi sul web che offrono servizi. L'intera architettura si basa su due protocolli:

- **SOAP:** che offre un gran numero di servizi ma è meno efficiente.
- **REST:** versione migliorata di SOAP con performance migliori.

Il concetto principale dei web services che il consumer non conosce nulla ,per quanto riguarda l'implementazione, del servizio che sta utilizzando, esso infatti si limita ad utilizzare l'interfaccia che quest'ultimo espone (**loose coupling**). I web services per questioni di standardizzazione utilizzando il protocollo HTTP (che è utilizzato a livello mondiale sulla porta 80). Tutto questo permette di esporre logica di business tramite servizi su internet e permette ai programmatore di evitare di scrivere codice già esistente, utilizzando servizi disponibile online e testati tramite delle interfacce in rete.

Software as Service: rappresenta un importante cambio di paradigma. Il prof ha fatto l'esempio di Bill Gates, di come sia stato uno dei primi a capire che vendere software e poi servizi è estremamente più remunerativo per tutta una serie di fattori. Poi l'esempio dell'aggiornamento che i software, che ormai avvengono sempre di più lato server e di come ormai i programmi vengono divisi in due parti: l'interfaccia che risiede sul client e che è personalizzabile (Per esempio esistono varie app che permettono di verificare i punti della patente, che si appoggiano tutte sullo stesso servizio) il resto risiede su server che offrono servizi e che sono completamente indipendenti dall'interfaccia. ([Altre informazioni sull'articolo capitolo 1,2 e 3](#)).

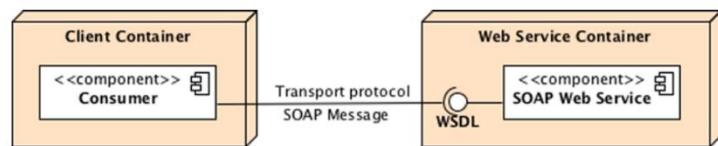
Simple Object Access Protocol (SOAP): è un protocollo che espone logica di business tramite servizi, garantendo un basso accoppiamento tramite un meccanismo che delega all'interfaccia la formattazione dei messaggi sia un response che in request. Praticamente il server **pubblica** un interfaccia e che il consumer **discover** e utilizza i servizi del provider. Un esempio di applicazione che utilizza i web service è SKY Scanner (biglietti aerei al miglior prezzo, utilizzando servizi offerti dai siti delle varie compagnie aeree).



I Web Services si basano su molte tecnologie e protocolli per trasportare e trasformare dati dal **consumer** al **service provider** seguendo uno standard unico. I principali sono:

- **Extensible Markup Language(XML):** utilizzato da SOAP per costruire e definire. Rappresenta una scelta “terza” per garantire uno standard comune di comunicazione. Permette di definire delle regole per garantire che il file sia correttamente formattato (Cosa molto utile, XML anche se odiato nasconde delle cose interessanti);

- **Web Services Descriptor Language (WSDL):** definisce i protocolli, le interfacce, i tipi di messaggi e le iterazione tra consumer e provider, è come un interfaccia di Java ma scritta in XML, serve a esporre i servizi del service provider;



Per esempio, sul libro vediamo come è possibile definire sia i messaggi in arrivo che i messaggi in uscita sia la che porta del servizio.

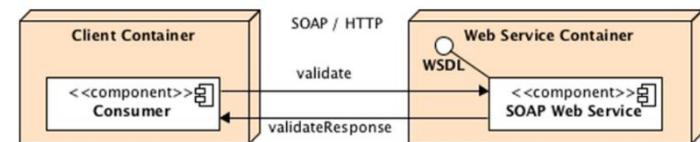
<< Non dobbiamo né scriverlo né leggerlo, ma dobbiamo capirlo !!! >>

Contiene tutte le informazioni utili al consumer per indentificare il provider, il protocollo di trasporto utilizzato e tutte le informazioni necessarie tramite una serie di elementi e attributi.

| Element | Description |
|-------------|---|
| definitions | Is the root element of the WSDL, and it specifies the global declarations of namespaces that are visible throughout the document |
| types | Defines the data types to be used in the messages. In this example, it is the XML Schema Definition (CardValidatorService?xsd=1) that describes the parameters passed to the web service request and the response |
| message | Defines the format of data being transmitted between a web service consumer and the web service itself. Here you have the request (the validate method) and the response (validateResponse) |
| portType | Specifies the operations of the web service (the validate method). Each operation refers to an input and output message |
| binding | Describes the concrete protocol (here SOAP) and data formats for the operations and messages defined for a particular port type |
| service | Contains a collection of <port> elements, where each port is associated with an endpoint (a network address location or URL) |
| port | Specifies an address for a binding, thus defining a single communication endpoint |

(Esempi da pagina 457 a 459 è importante saper leggere il codice XML).

- **Simple Object Access Protocol (SOAP):** è un protocollo di **message-encoding** utilizzato per definire la comunicazione dei web services. Si occupa in pratica di implementare lo scambio di messaggi XML tra provider e consumer. Se il WSDL definisce il tipo di messaggi che devono essere scambiati SOAP li implementa. (Esempi pagina 460).



| Element | Description |
|----------|--|
| Envelope | Defines the message and the namespace used in the document. This is a required root element |
| Header | Contains any optional attributes of the message or application-specific infrastructure such as security information or network routing |
| Body | Contains the message being exchanged between applications |
| Fault | Provides information about errors that occur while the message is processed. This element is optional |

- **HTTPS, SMTP, JMS:** come protocolli di trasporto;
- **Universal Descriptor Discover, and Integration (UDDI):** un servizio di registri e meccanismi di **discover** opzionale (se mi conetto a una mia applicazione strutturata in servizi non è detto che ho necessità di utilizzare un registro), permette al consumer di indentificare il provider e di conoscere gli standard per comunicare con quest'ultimo.

I tre pilastri dei Web Services sono: **WSDL,SOAP,UDDI**. (conoscenze importanti che dobbiamo sapere altrimenti ci boccia).

Writing SOAP Web Services: Per scrivere Web Services esistono due principali approcci:

- **Contract-first:** visto che il WSDL è il contratto tra consumer e service, può essere utilizzato per generare il codice java per consumer e provider.
- **Bottom-UP:** si parte da un codice normale e lo si rende web services tramite l'annotazione **@WebService** (**Le 11 Lettere della gloria, nel compito se mi chiede di scrivere un web service basta che aggiungo queste 11 lettere per avere 4 punti in più**).

Java Architecture for XML Binding (JAXB): Prima di essere inviato un oggetto scritto in java deve essere convertito in un file XML e successivamente dopo essere giunto a destinazione deve essere fatto l'opposto. In JEE questo processo viene svolto da **JAXB** tramite un insieme di API. Esempio:

Tutto questo è meno efficiente (perché appunto aggiungiamo delle operazioni di parsing), ma non è nulla rispetto agli enormi vantaggi lato servizi che otteniamo.

```
@XmlRootElement  
public class CreditCard {  
  
    @XmlAttribute(required = true)  
    private String number;  
    @XmlAttribute(name = "expiry_date", required = true)  
    private String expiryDate;  
    @XmlAttribute(name = "control_number", required = true)  
    private Integer controlNumber;  
    @XmlAttribute(required = true)  
    private String type;  
  
    // Constructors, getters, setters  
}
```

([Altri esempi pagina da 464 a 466](#)).

WSDL Mapping: A livello di servizio il sistema è definito in termini di messaggi XML, operazioni WSDL e messaggi SOAP. Al livello di Java invece troviamo: oggetti, interfacce e metodi. Quindi occorre un servizio che converta gli oggetti java in XML e viceversa.

- **JAXB:** utilizza annotazioni per determinare come **marshal/unmarshal** una classe to/from XML.
- **JSON Web Signature (JWS):** utilizzato per mappare classi java in WSDL, per effettuare il **marshal** di un invocazione un metodo tramite SOAP e **unmarshal** l'eventuale risposta del metodo.

```
@XmlRootElement  
public class CreditCard {  
  
    @XmlAttribute(required = true)  
    private String number;  
    @XmlAttribute(name = "expiry_date", required = true)  
    private String expiryDate;  
    @XmlAttribute(name = "control_number", required = true)  
    private Integer controlNumber;  
    @XmlAttribute(required = true)  
    private String type;  
  
    // Constructors, getters, setters  
}
```

@WebService: marca una classe java o un interfaccia in modo da farla diventare un Web Service.

```
@WebService  
public class CardValidator {...}
```

È possibile utilizzarlo anche su interfacce, specificando l'endpoint.

```
@WebService  
public interface Validator {...}  
  
@WebService(endpointInterface = "org.agoncal.book.javaee7.chapter14.Validator")  
public class CardValidator implements Validator {...}
```

([Esempi pagina da 467](#))

Sono disponibili due diverse tipologie di annotazioni:

- **WSDL mapping annotations:** Utilizzate per personalizzare le firme dei metodi esposti.
 - **@WebMethod:** Permette di modificare il nome del metodo presente in WSDL o di escluderlo da quest'ultimo in modo che non sia visibile al client. ([Esempi pagina da 468](#))
 - **@WebResult:** Controlla il nome del messaggio di ritorno in WSDL, permettendo di modificarlo, per esempio il risultato del metodo **validate()** viene rinominato come **IsValid**: ([Esempio pagina da 469](#))

```
@WebService  
public class CardValidator {  
  
    @WebMethod(operationName = "ValidateCreditCard")  
    public boolean validate(CreditCard creditCard) {  
        // Business logic  
    }  
  
    @WebMethod(operationName = "ValidateCreditCardNumber")  
    public void validate(String creditCardNumber) {  
        // Business logic  
    }  
  
    @WebMethod(exclude = true)  
    public void validate(Long creditCardNumber) {  
        // Business logic  
    }  
  
    @WebResult(name = "IsValid")  
    public boolean validate(CreditCard creditCard) {  
        // Business logic  
    }  
}
```

- **@WebParam:** simile a **@WebResult** ma si occupa di parametri, permette infatti di modificare il nome di un parametro e il suo tipo (**IN**, **OUT** o **INOUT**).

```
@WebService
public class CardValidator {
    public boolean validate(@WebParam(name= "Credit-Card", mode = IN) CreditCard creditCard) {
        // Business logic
    }
}
```

- **@OneWay:** viene utilizzata su metodi che non hanno un valore di ritorno (**void**), serve a informare il container in modo che possa ottimizzare la chiamata.

- **SOAP Binding Annotations:** Descrive come il web service è limitato a un protocollo di messaggistica. Esistono due tipi di stili di programmazione che definiscono il SOAP Binding e che differiscono sostanzialmente per il contenuto del **SOAPbody**:

- **Document:** il messaggio contiene un documento senza regole di formattazioni ulteriori (Scelta di default).
- **RPC:** il SOAP contiene un elemento con il nome del metodo o la procedura remota che è stata invocata.

è inoltre possibile scegliere fra due tipi di serializzazione/deserializzazione:

- **Literal:** i dati vengono serializzati in accordo allo schema XML.
- **Encoded:** la codifica del SOAP specifica come gli oggetti, array, ecc... devono essere serializzati.

Questo ci porta ad avere 4 possibili tipologie di configurazioni:

- **Document/Literal** (Configurazione di Default).
- **Document/Encoded**
- **RPC/Literal**
- **RPC/Encoded**

([Esempio pagina 471](#))

```
@WebService
@SOAPBinding(style = RPC, use = LITERAL)
public class CardValidator {
    public boolean validate(CreditCard creditCard) {
        // Business logic
    }
}
```

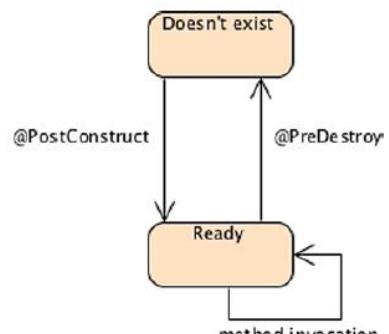
Handling Exceptions: in java, quando qualcosa va storto viene generata un'eccezione che viene catturata e gestita da un metodo, con SOAP questo meccanismo non funziona perché producer e consumer non è detto che parlino lo stesso linguaggio e per di più sono separati da uno strato di rete. Per ovviare a questo problema SOAP riconosce automaticamente le eccezioni quando vengono generate e le converte in un **SOAP Fault** all'interno di **SOAP Message**.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
    <soap:Fault>
        <faultcode>soap:Server</faultcode>
        <faultstring>java.lang.NullPointerException</faultstring>
    </soap:Fault>
</soap:Body>
</soap:Envelope>
```

([Approfondimento pagina 475/476](#))

Ciclo di vita: anche in questo caso è molto banale, con l'esecuzione di metodi annotati con **@PostConstruct** e **@PreDestroy**, dopo la creazione e prima della distruzione.

WebServiceContext: SOAP dispone di un ambiente basato sul contesto a cui si può accedere iniettandone il riferimento tramite l'annotazione **@Resource**, permettendoci così di ottenere informazioni come: L'endpoint della classe di implementazione, il contesto del messaggio, le informazioni di sicurezza ecc..



| Method | Description |
|----------------------|--|
| getHttpContext | Returns the MessageContext for the request being served at the time this method is called. It can be used to access the SOAP message headers, body, and so on. |
| getUserPrincipal | Returns the Principal that identifies the sender of the request currently being serviced. |
| isUserInRole | Returns a Boolean indicating whether the authenticated user is included in the specified logical role. |
| getEndpointReference | Returns the EndpointReference associated with this endpoint. |

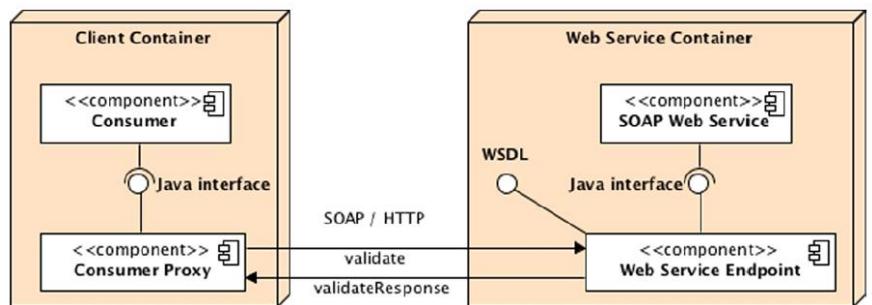
```
@WebService
public class CardValidator {
    @Resource
    private WebServiceContext context;

    public boolean validate(CreditCard creditCard) {
        if (!context.isUserInRole("Admin"))
            throw new SecurityException("Only Admins can validate cards");

        // Business logic
    }
}
```

([Altri concetti da 478 a 480](#)).

Invoking SOAP Web Service: Invocare un Web Services è molto simile a invocare un oggetto con RMI, come possiamo vedere dal grafico, all'interno del client che è un container a tutti gli effetti, possiamo trovare un **Consumer Proxy** che, a tutti gli effetti, rappresenta il Web Service all'interno del client un po' come faceva lo Stub in Java RMI. Questo proxy ha come compito quello di convertire le richieste ai metodi effettuate dal client in messaggi SOAP spendendoli via web al **Web Service Endpoint** che li elabora e invia la risposta come messaggio SOAP che viene convertito in un'istanza del tipo di ritorno scelto dal metodo e presentato al client dal **Consumer Proxy**. Il vantaggio che questo sistema offre è che sia il client che il server non hanno necessità di comprendere l'infrastruttura di rete né questi meccanismi che sono completamente invisibili al programmatore.



Invocazione programmatica: Se il consumer viene eseguito al di fuori del container, bisogna invocare programmaticamente il SOAP Web Service. Richiedendo prima un'istanza del servizio (**CardValidatorService()**) in modo da poter invocare i metodi localmente.

([Pagina 481](#)).

```
public class WebServiceConsumer {  
    public static void main(String[] args) {  
        CreditCard creditCard = new CreditCard();  
        creditCard.setNumber("12341234");  
        creditCard.setExpiryDate("10/12");  
        creditCard.setType("VISA");  
        creditCard.setControlNumber(1234);  
  
        CardValidator cardValidator = new CardValidatorService().getCardValidatorPort();  
        cardValidator.validate(creditCard);  
    }  
}
```

Invocazione tramite Injection: se il consumer viene eseguito all'interno di un container si può utilizzare **l'injection** per referenziare il SOAP Web Service client Proxy. In modo molto simile a **@Resource** o **@EJB**.

([Pagina 482](#)).

```
public class WebServiceConsumer {  
    @WebServiceRef  
    private static CardValidatorService cardValidatorService;  
  
    public static void main(String[] args) {  
        CreditCard creditCard = new CreditCard();  
        creditCard.setNumber("12341234");  
        creditCard.setExpiryDate("10/12");  
        creditCard.setType("VISA");  
        creditCard.setControlNumber(1234);  
  
        CardValidator cardValidator = cardValidatorService.getCardValidatorPort();  
        cardValidator.validate(creditCard);  
    }  
}
```

Invocazione con CDI: Si può anche direttamente iniettare il riferimento a un Web Service utilizzando l'annotazione **@Inject**.

([Pagina 482/483](#)).

```
public class WebServiceProducer {  
    @Produces  
    @WebServiceRef  
    private CardValidatorService cardValidatorService;  
}
```

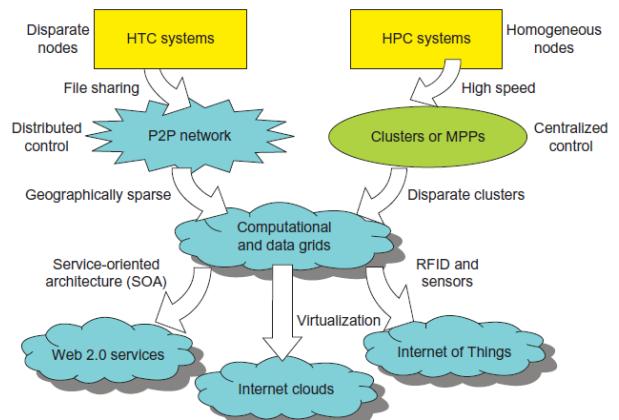
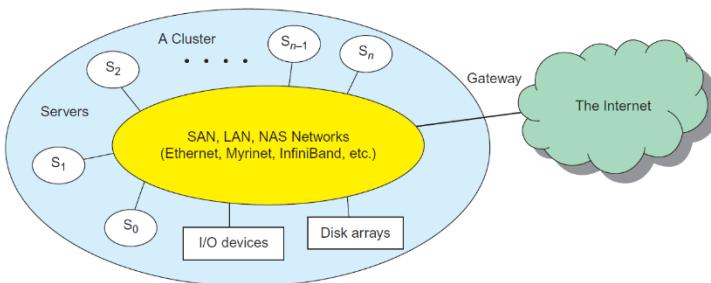
```
@Stateless  
public class EJBConsumerWithCDI {  
  
    @Inject  
    private CardValidatorService cardValidatorService;  
  
    public boolean validate(CreditCard creditCard) {  
        CardValidator cardValidator = cardValidatorService.getCardValidatorPort();  
        return cardValidator.validate(creditCard);  
    }  
}
```

Concetti rilevanti di Cloud Computing:

Questo grafico rappresenta l'evoluzione dei sistemi HTC e HPC negli anni.

High-Performance Computing (HPC): Sistemi basati sulle pure performance e utilizzati per risolvere task singoli nel minor tempo possibile. La loro potenza viene misurata tramite il **Linpack Benchmark**.

Cluster architettura: è costruita intorno a una rete a bassa latenza e ad alta banda che connette vari server tra di loro in modo che condividano potenza di calcolo prima di offrirla ai client tramite internet.



High-Throughput Computing (HTC): Sistemi in grado di poter eseguire il maggior numero di task possibili contemporaneamente, utilizzano un architettura P2P. Dove non è importante verificare con assoluta certezza che un task vada a buon fine (UDP per esempio con lo streaming video).

Reti P2P: all'interno di una rete P2P tutti gli host svolgono sia funziona di client che di server, quindi sia offrendo che consumando risorse, la rete è una rete “libera” ogni host si connette e disconnette in modo libero e indipendente, cosa possibile dal fatto che i dati sono ridondanti nella rete in modo da renderla sempre funzionale anche in mancanza di nodi.

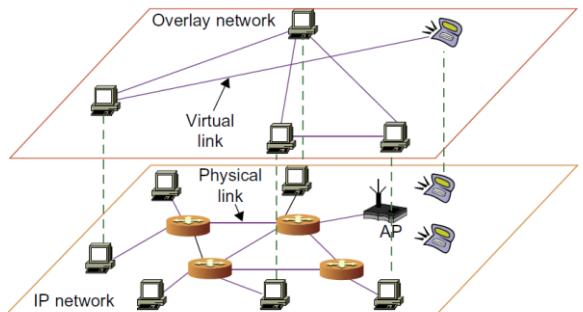


Table 1.5 Major Categories of P2P Network Families [46]

| System Features | Distributed File Sharing | Collaborative Platform | Distributed P2P Computing | P2P Platform |
|-------------------------|---|---|--|---|
| Attractive Applications | Content distribution of MP3 music, video, open software, etc. | Instant messaging, collaborative design and gaming | Scientific exploration and social networking | Open networks for public resources |
| Operational Problems | Loose security and serious online copyright violations | Lack of trust, disturbed by spam, privacy, and peer collusion | Security holes, selfish partners, and peer collusion | Lack of standards or protection protocols |
| Example Systems | Gnutella, Napster, eMule, BitTorrent, Aimster, KaZaA, etc. | ICQ, AIM, Groove, Magi, Multiplayer Games, Skype, etc. | SETI@home, Geonome@home, etc. | JXTA, .NET, FightingAid@home, etc. |

Computational and Data Grids: ricalca un po' il concetto delle reti idriche o fognare, io mi connetto a una griglia e ne utilizzo i servizi che propone senza interessarmi di dove quei servizi siano prodotti.

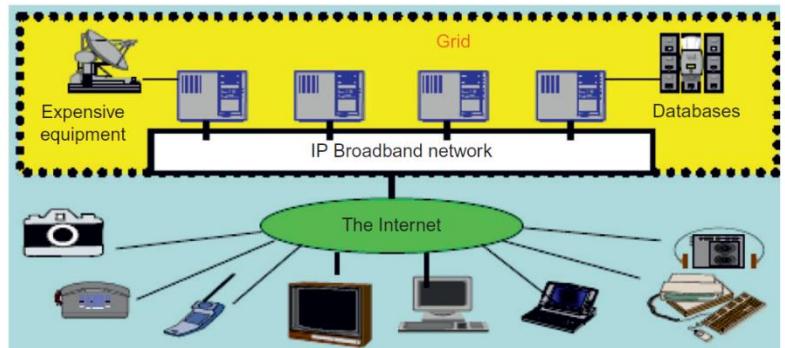
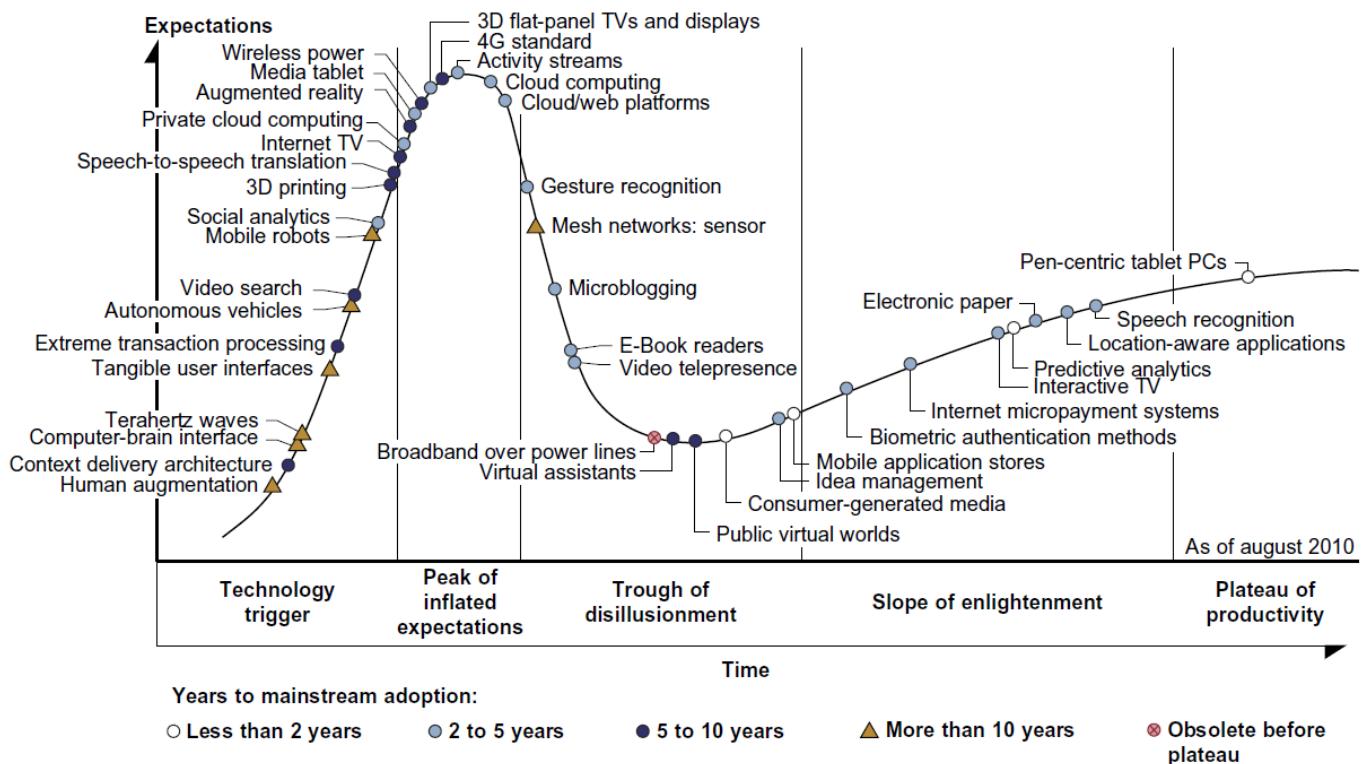


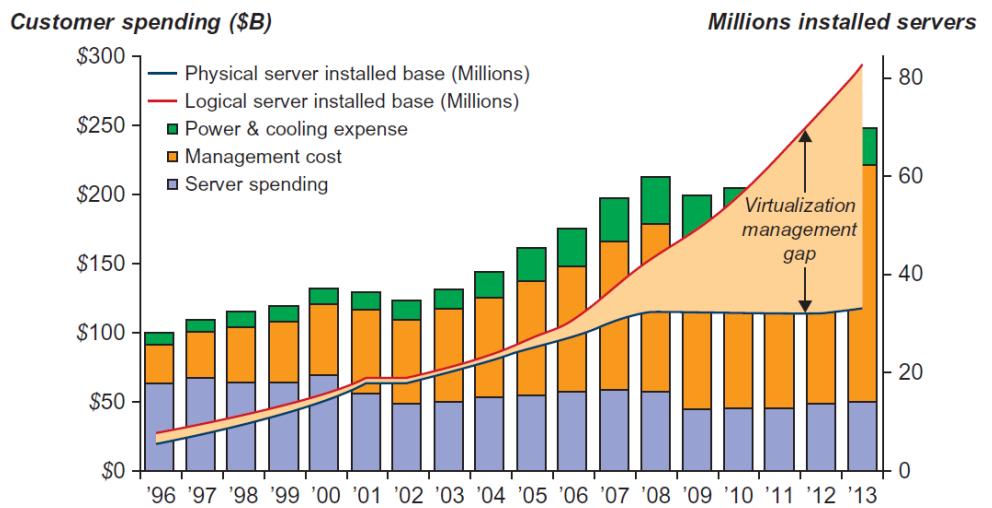
Table 1.4 Two Grid Computing Infrastructures and Representative Systems

| Design Issues | Computational and Data Grids | P2P Grids |
|-----------------------------|--|--|
| Grid Applications Reported | Distributed supercomputing, National Grid initiatives, etc. | Open grid with P2P flexibility, all resources from client machines |
| Representative Systems | TeraGrid built in US, ChinaGrid in China, and the e-Science grid built in UK | JXTA, FightAid@home, SETI@home |
| Development Lessons Learned | Restricted user groups, middleware bugs, protocols to acquire resources | Unreliable user-contributed resources, limited to a few apps |

The Hype Cycle of New Technologies: Questo grafico rappresenta l'andamento del hype per le nuove tecnologie. Abbiamo prima di tutto una fase ascendente che porta all'esagerazione poi una fase discendente e successivamente un affermarsi della tecnologia in questione.



Infrastrutture Virtuali: in questo grafico viene mostrato come il processo di virtualizzazione delle infrastrutture (pago un provider che mi offre i server su cui faccio girare la mia intera infrastruttura virtuale), ha portato a una riduzione di costi per l'hardware, a una stabilizzazione dei costi energetici e a un aumento dei costi di manutenzione negli anni.



Cloud service models: I servizi offerti dal cloud possono essere divisi in tre categorie principali:

- **Infrastructure as a Service (IaaS):** l'utente può utilizzare il SO, sistemi di storage, applicazioni e i componenti della rete, ma non controlla direttamente l'infrastruttura cloud. Praticamente ha a disposizione una macchina da configurare (macchine del lab di SO).

Table 4.1 Public Cloud Offerings of IaaS [10,18]

| Cloud Name | VM Instance Capacity | API and Access Tools | Hypervisor, Guest OS |
|----------------------|---|-----------------------------------|--------------------------------------|
| Amazon EC2 | Each instance has 1–20 EC2 processors, 1.7–15 GB of memory, and 160–1.69 TB of storage. | CLI or web Service (WS) portal | Xen, Linux, Windows |
| GoGrid | Each instance has 1–6 CPUs, 0.5–8 GB of memory, and 30–480 GB of storage. | REST, Java, PHP, Python, Ruby | Xen, Linux, Windows |
| Rackspace Cloud | Each instance has a four-core CPU, 0.25–16 GB of memory, and 10–620 GB of storage. | REST, Python, PHP, Java, C#, .NET | Xen, Linux |
| FlexiScale in the UK | Each instance has 1–4 CPUs, 0.5–16 GB of memory, and 20–270 GB of storage. | web console | Xen, Linux, Windows |
| Joyent Cloud | Each instance has up to eight CPUs, 0.25–32 GB of memory, and 30–480 GB of storage. | No specific API, SSH, Virtual/Min | OS-level virtualization, OpenSolaris |

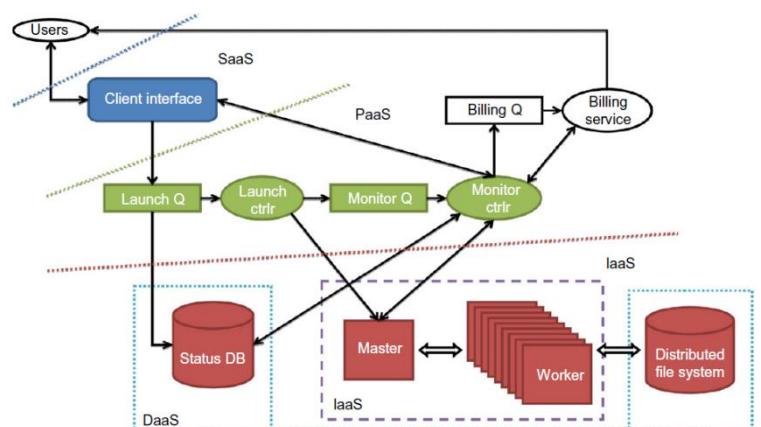
- **Platform as a Service (PaaS):** permette di distribuire e gestire l'esecuzione di un'applicazione utilizzando risorse fornite dal provider cloud tramite un ambiente adeguato. (Server di Minecraft che si noleggiavano).

Table 4.2 Five Public Cloud Offerings of PaaS [10,18]

| Cloud Name | Languages and Developer Tools | Programming Models Supported by Provider | Target Applications and Storage Option |
|----------------------------|---|---|--|
| Google App Engine | Python, Java, and Eclipse-based IDE | MapReduce, web programming on demand | Web applications and BigTable storage |
| Salesforce.com's Force.com | Apex, Eclipse-based IDE, web-based Wizard | Workflow, Excel-like formula, Web programming on demand | Business applications such as CRM |
| Microsoft Azure | .NET, Azure tools for MS Visual Studio | Unrestricted model | Enterprise and web applications |
| Amazon Elastic MapReduce | Hive, Pig, Cascading, Java, Ruby, Perl, Python, PHP, R, C++ | MapReduce | Data processing and e-commerce |
| Aneka | .NET, stand-alone SDK | Threads, task, MapReduce | .NET enterprise applications, HPC |

- **Software as a Service (SaaS):** permette di utilizzare software tramite un interfaccia web. L'esempio più immediato è Google Docs o Word Online.

Per capire la differenza fra questi tre service modelli di cloud è molto utile questa immagine che mostra la separazione e cosa offrono al cliente i vari modelli:



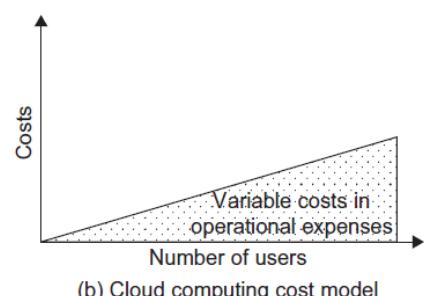
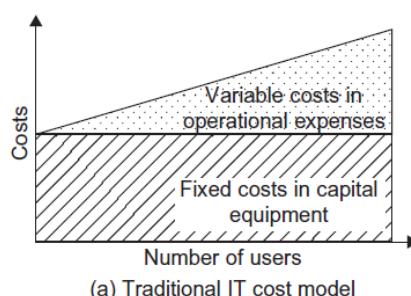
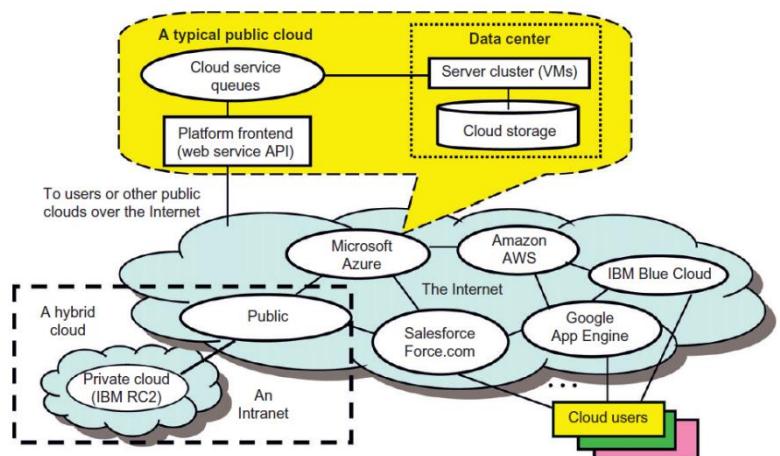
(Da pagina 200 a 206 del mini-libro).

Cloud Pubblici: costruito su internet può essere utilizzato da chiunque pagando una certa quota mensile o giornaliera.

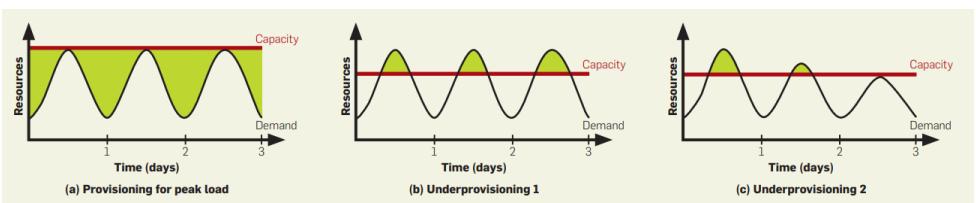
Cloud Privati: costruito all'interno di un dominio o all'interno di un intranet in possesso di una singola organizzazione, viene utilizzato per la gestione interna e per il funzionamento della stessa.

Cloud ibridi: utilizzato sia per scopi interno ad un'azienda o a un'organizzazione che per offrire servizi al pubblico tramite internet.

Model Cost: Grazie al cloud viene eliminata tutta quella parte di costi fissi per l'equipaggiamento che le aziende si trovavano a dover affrontare per la messa in opera della loro infrastruttura hardware.



Cloud Computing Economics: il cloud computing si rileva estremamente conveniente dal punto di vista economico in alcuni casi. Per esempio, quando la **domanda** per un servizio **varia nel tempo**, (come in questa figura), infatti se gli host del servizio decidono di **gestire i picchi** di carico di quest'ultimo si ritroveranno inevitabilmente ad acquistare delle macchine che per la maggior parte del tempo rimarranno inutilizzate (area verde), se d'altro canto decidono di **ignorare i picchi**, inevitabilmente con il passare del tempo gli utenti, trovando un servizio lento e inaffidabile, **migreranno** altrove con conseguente perdita di ricavi. Altri vantaggi si hanno in situazioni in cui **non è ben nota a priori la quantità di traffico** che il servizio si ritroverà a gestire, problema che viene risolto dalla rapida **scalabilità** (allocando altre macchine), che il cloud possiede. Infine, un'altra situazione in cui il cloud è molto utile è quando ci troviamo ad avere bisogno di **semplice e pura potenza di calcolo**, che il cloud non ha difficoltà a offrire (evitando alti costi per l'acquisto di un gran numero di macchine performanti, l'utente paga **quanto consuma**).



Concetti rilevanti di Microservice:

La morte dei Big Software: Visto la sempre crescente complessità dei software odierni con il passare del tempo le aziende si sono ritrovate a sviluppare software utilizzandone altri già esistenti con lo scopo di ridurre i costi oltre che per cercare di sviluppare programmi che siano adattino a diversi campi di utilizzo. L'insieme di tutti questi fattori ha portato alla morte dei cosiddetti **Big Software** (o **Monolith**) ovvero di quei software estremamente voluminosi.

Un altro fattore che si è cercato molto di ridurre è il cosiddetto **Total Cost of Ownership TCO**, ovvero il costo derivante dal possesso di centri di calcolo (**il cloud risolve questo problema**). Altri fattori sono stati: i grandi costi di manutenzione anche dopo il rilascio e la natura strettamente accoppiata dei moduli componevano questi software (**per ovvi motivi tecnologici**).

La soluzione a questi problemi è il cosiddetto **Software Small**, ovvero un tipo di prodotto software basato su servizi sviluppati e mantenuti indipendentemente, in questo modo i vari moduli sono indipendenti e vengono mantenuti e ottimizzati dal team che li ha sviluppato.

Micro Services: sorgono dalla cenere dei big software. Nell'architettura a micro-servizi la gestione del software è in **mano al cliente** che ne gestisce lo sviluppo e il mantenimento.

Le applicazioni si sviluppano in tanti piccoli servizi in modo che: **comunichino tra di loro** nel modo più "leggero" possibile, risultino **indipendenti dalla piattaforma** (Aperti all'Eterogeneità) e siano costruiti intorno alla necessità di **business**.

Un tipico esempio di software strutturato a micro-servizi è **Netflix**, il quale offre un servizio che ci offre una lista di film di cui potremmo essere interessati, in base a quelli che abbiamo visto di recente.

L'intera architettura a micro-servizi di base quindi sul prendere un'applicazione monolitica e dividerla in tanti piccoli servizi, ottenendo vantaggi soprattutto per quanto riguarda la **scalabilità** (se molti utenti utilizzano il servizio **Arancione**, lo prendo e lo moltiplico su più macchine).

All'inizio è complicato sviluppare in micro-servizi, però quando la complessità aumenta questa difficoltà si attenua fino a ripagare del tutto gli svantaggi.

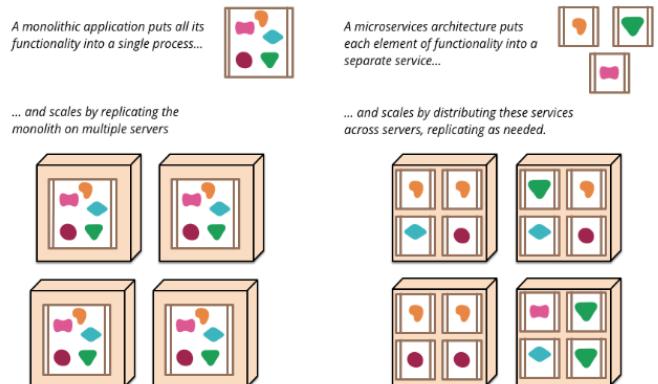


Figure 1: Monoliths and Microservices

Componenti come servizi: i micro-servizi sono WebService, sono indipendentemente deployabili e non c'è coesione. Anche in questo caso un producer espone qualcosa che può essere utilizzato da client (Netflix per esempio espone la lista io la consulto).

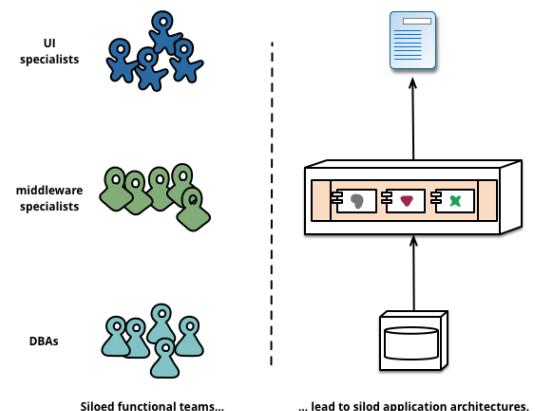
Ci sono alcuni svantaggi in tutto questo: Infatti effettuare un'invocazione standard di metodi, vado sulla rete questo porta inevitabilmente a dei cali di prestazione, che però sono nulla rispetto ai vantaggi che questa architettura offre.

Sono realizzati intorno alle competenze di business:

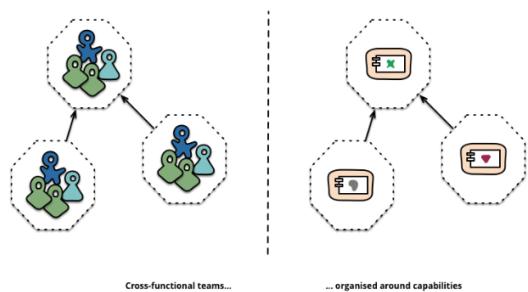
legge di Conway (legge empirica, non è detto che sia mantenuta, per ora funge, un po' come Moore):

Qualsiasi organizzazione che progetta un sistema produrrà un design la cui struttura è una copia della struttura di comunicazione dell'organizzazione.

In parole povere i nostri team sono composti da specialisti del medesimo settore o competenza, quest'ultimi produrranno una **architettura a Silos**, dove ognuno produce un pezzo differente e poi si mettono d'accordo di come farlo comunicare.



Nell'architettura a MS invece ogni team contiene specialisti in diversi settori o competenze (**Team Cross Functional**), con lo scopo di progettare una funzionalità che sia utile all'obiettivo di business e di legarle con nel modo più leggero possibile alle altre (vogliamo un sistema strettamente accoppiato).



I team sono indipendenti, piccoli (4-6 otto persone al massimo, **meno pizza necessaria XD**) e non lavorano su progetti ma sui prodotti, in modo che il team sia responsabile sia durante lo sviluppo che durante la produzione (**se faccio una c*****a poi ne pago il prezzo dopo, non scarico sulle spalle di altri**).

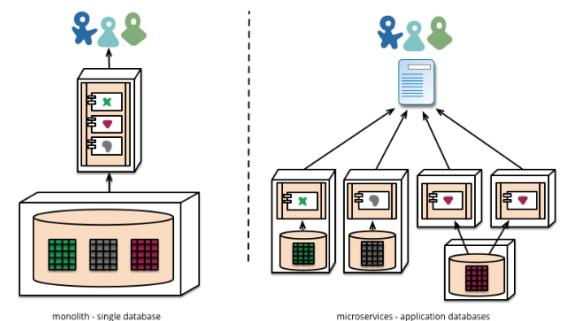
Smart endpoints and dumb pipes: I team che sviluppano in micro-servizi utilizzano per la comunicazione i principi e i protocolli su cui si basa il World Wide Web.

L'infrastruttura scelta è in genere stupida (Estremamente semplice e rapida) mentre nei software Monolith la comunicazione tra i vari componenti avviene tramite invocazioni di metodi o chiamate di funzioni, strategie che risultano estremamente lente e complesse è quindi necessario sostituire la comunicazione a grana fine con un approccio più grossolano.

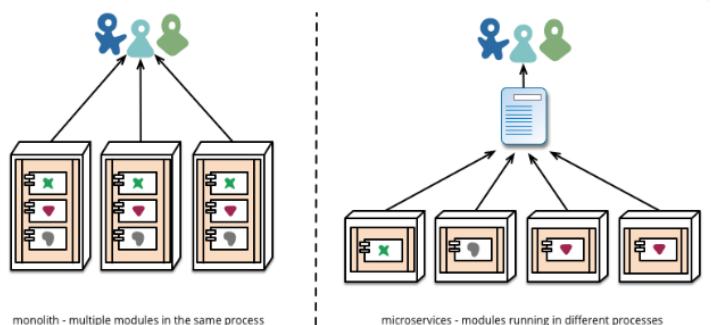
Gestione decentralizzata del calcolo e dei dati: All'interno dell'architettura MS è importante non evitare la standardizzazione: Non devo sicuramente obbligare nessuno a utilizzare uno specifico linguaggio o ecosistema, in modo da aumentare la produttività e l'efficienza dei singoli team (ovviamente ogni team si prende la responsabilità del linguaggio scelto). Ma devo anche assicurarmi che servizi scritti con linguaggi diversi siano perfettamente compatibili, per questo all'interno dei software MS è estremamente importante la **definizione dei contratti (interfacce)**, quello che c'è dietro quest'ultime invece importa.

Per esempio, in Amazon ogni team è libero di sviluppare il servizio come meglio crede, ma anche Amazon è libera di chiamare qualcuno alle 3 di notte in caso di problemi.

Invece per quanto riguarda i dati, di solito nella progettazione monolitica questi ultimi sono inseriti all'interno di un singolo database, nelle architetture MS invece ogni team è libero di scegliere il proprio DB in base alle proprie necessità. L'unica vincolo è che poi i vari DB siano in grado di riconciliarsi e di comunicare fra di loro.



Infrastructure Automation: ovvero utilizzare delle infrastrutture che automatizzino tutta la fase di delivery e integrazione. Tramite test e deployment automatici.



Progettati per i malfunzionamenti: All'interno di un architettura MS deve fare in modo di considerare che altri servizi possano non funzionare o non essere disponibili in determinati momenti ed è di fondamentale importanza che questo tipo di eventualità sia gestibile, evitando che l'intero ecosistema vada **Down** (Per esempio magari su Netflix non vedi i suggerimenti ma il resto funziona).

Progettazione evolutiva: è concetto di **progettazione agile** che permette di avere frequenti cicli di release, in pratica si cerca di rendere totalmente indipendenti questi servizi rendendone facile la sostituzione o l'aggiornamento (Se si nota che due servizi vengono aggiornati o sostituiti assieme, forse dovrebbero essere uniti).

Per quanto riguarda il **Planning delle release** in architetture MS si rilascia un servizio appena pronto in modo che ogni team funzioni alla sua velocità. È anche possibile fare del **versioning** dei servizi ma si cerca di evitarlo, perché con il passare del tempo le versioni che ci si porta "sulle spalle" aumentano, aumentando i costi di bug fix e di gestione (si cerca di implementare una retrocompatibilità costante invece).

<< A Scarano piace Netflix, conquisterà il mondo un giorno !!!>>

Domanda del Prof: Differenza tra MC e Web Services: sono due cose diverse, sono distribuiti su team, lo sviluppo indipendente, è un cambio a livello di paradigma nei MS troviamo la gestione del team, e tutto quello che sta scritto sopra. I WB sono più a basso livello sono l'implementazione in java se vogliamo.

Concetti di Server-Less computing:

Un miglioramento dell'architettura MC è quella del **Server-less Computing**, banalmente all'interno del SL non c'è un server. Questo tipo di architettura conosciuta anche come **Function as a Service** si basa sulla creazione di funzioni che vengono eseguite all'interno di macchine cloud e presentate all'utente come MC.

Il vantaggio è notevole, prima di tutto non devo gestire il server con tutte le sue problematiche e costi, inoltre le funzioni vengono eseguite in container isolati, (molto più efficienti delle VM che abbiamo visto nel cloud, visto che il sistema operativo sottostante rimane lo stesso, indipendentemente dall'uso che si fa della macchina).

In pratica l'intero sistema si basa su una funzione che viene invocata al verificarsi di un evento, per esempio quando un client fa una richiesta (**per educazione XD, si risponde**), ma posso reagire anche a qualcosa di più specifico come l'inserimento di tupla all'interno del DB, oppure ancora quando qualcosa viene modificato, quando vengono dati dei comandi vocali, qualsiasi cosa che possa generare un evento.

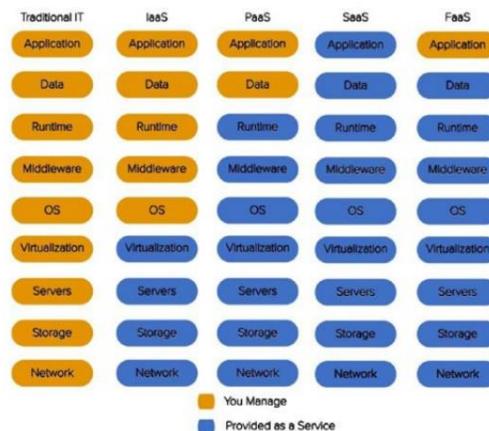
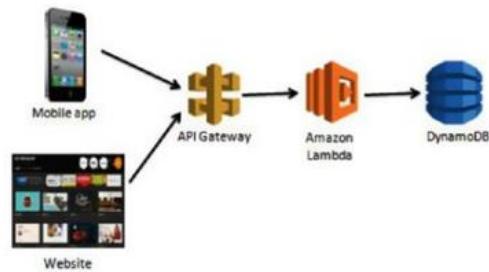
Esempio: Amazon per esempio usa un **API gateway**, che avvia una determinata funzione, definita **Lambda** (che si trovano su diverse macchine in cloud) e che fa ciò per cui è programmata. In questa architettura come vediamo manca un server.

Ci ritroviamo quindi a una gestione di questo tipo mostrato in figura: Dove per ogni tipo di servizio offerto dal cloud, abbiamo una parte gestita dal programmatore e dall'altra gestita dal provider di servizi.

Questo tipo di architettura in API permette una scalabilità estremamente efficiente e facile da realizzare. In particolare, quanto più piccoli sono i task e le funzioni o MC, più posso scalare efficientemente.

Benefici dello sviluppo Server-Less:

- Sviluppo e deployment rapido:** i costi di sviluppo sono bassi e i tempi di sviluppo rapidi, con la possibilità di rilasciare subito e migliorare nel tempo.
- Facilità di uso:** essenzialmente posso realizzare funzionalità complesse che si evolvono in maniera diversa e indipendente.
- Costo minore:** tutti i costi di manutenzione e gestione vengono eliminati. (tutti i distributori di Coca-Cola americani connessi hanno un costo di gestione di circa 20000 dollari su infrastruttura Amazon basata su MC, con infrastruttura SL invece il costo si abbassa a circa 1700 dollari l'anno).
- Scalabilità:** usare SL permette al provider di bilanciare il carico in modo molto più semplice e veloce, cosa che permette anche al provider di offrire servizi a costi minori. (Utilizza le macchine sempre al 100% visto che si ritrova a gestire tante piccole funzioni, che sono facilmente "incastrabili" **esempio del portabagagli (da non ripetere all'esame)**). In poche parole, l'utente paga le invocazioni invece delle macchine.
- Nessun costo di manutenzione dell'infrastruttura:** che ovviamente sono a carico dei provider.



| AWS Lambda | Azure Functions | Google Cloud Functions |
|--|--|--|
| First million requests a month free | First million requests a month free | First 2 million requests a month free |
| \$0.20 per million requests afterwards | \$0.20 per million requests afterwards | \$0.40 per million requests afterwards |
| \$0.00001667 for every GB-second used | \$0.000016 for every GB-second used | \$0.000025 for every GB-second used |

Limiti dell'architettura Server-Less:

- **Si perde il controllo dell'infrastruttura hardware:** ci si lega al provider sia economicamente che progettualmente (se ho sviluppato su Amazon e non posso spostarmi su Azure, e quindi se Amazon alza i costi, o resto e pago o riprogetto tutto e mi sposto).
- **Cold start:** Le applicazione SL hanno un “**avviamento freddo**” perché il sistema che gestisce e ottimizza queste funzioni se vede che non sono utilizzate per un tot di tempo le mette da parte, per risolvere questo problema si fa un **Ping** per riattivare la funzione ogni tanto, in modo da tenerla “sveglia” (ogni Ping ovviamente costa quindi se lo facciamo troppo spesso i costi lievitano).
- **Architettura condivisa:** Netflix e Disney per esempio potrebbero condividere l'infrastruttura (è possibile chiedere che siano separate, ma costa di più). Potrebbero succedere cose del tipo: se Netflix Fa cadere il servizio va giù anche Disney.

<< E così finisce la nostra avventura con il grande prof Scarano !!!>>

<< The End !!!>>

Domande per lezione:

Lezione 1:

- 1) Cosa è un sistema distribuito?
- 2) Quali sono le motivazioni tecnologiche ai sistemi distribuiti?
- 3) Quali sono le motivazioni economiche ai sistemi distribuiti?
- 4) Cosa è la "legge" di Moore?
- 5) In che maniera un sistema distribuito permette di rispondere efficacemente al progresso tecnologico preservando le risorse legacy?
- 6) Cosa sono le "leggi" di Sarnoff / Metcalfe / Reed?
- 7) Perché nelle domande precedenti, la parola "legge" viene messa tra virgolette? sorridente
- 8) Che cosa è il middleware e quale è il suo ruolo?
- 9) Perché le reti sono naturalmente (dal punto di vista economico) portate ad accorparsi, integrando i servizi in modo da fornirli agli utenti di tutte le reti che si stanno fondendo? E che cosa ha a che fare questo con le leggi di Metcalfe / Reed? E perché questo capita in maniera molto minore con la legge di Sarnoff (ma capita comunque)?

Lezione 2:

- 1) Cosa è un modello di riferimento (reference model) e a cosa/chi serve?
- 2) Cosa sono i requisiti non funzionali di un sistema?
- 3) Quale è la differenza sostanziale tra un sistema distribuito ed uno parallelo/concorrente?
- 4) Perché un sistema distribuito è concorrente, e perché la soluzione della concorrenza è più complessa rispetto ad un sistema centralizzato?
- 5) Che significa che in un sistema distribuito manca un clock globale, e che cosa comporta?
- 6) Cosa significa che un sistema distribuito può tollerare malfunzionamenti parziali?
- 7) Perché un sistema distribuito è eterogeneo?
- 8) Cosa significa che un sistema distribuito asseconda l'evoluzione aziendale?
- 9) Perché un sistema distribuito offre autonomia di gestione ai singoli nodi?
- 10) Cosa significa che un sistema distribuito deve:
 - 11) essere aperto
 - 12) essere integrato
 - 13) essere flessibile
 - 14) essere modulare
 - 15) supportare la federazione di sistemi
 - 16) essere facilmente gestibile
 - 17) essere scalabile
 - 18) essere trasparente
 - 19) Cosa è la trasparenza di accesso?
 - 20) Cosa è la trasparenza di locazione?
 - 21) Cosa è la trasparenza di migrazione / replica e perché dipende da trasparenza di locazione e di accesso?
 - 22) Cosa è la trasparenza alla persistenza e perché dipende dalla trasparenza di locazione
 - 23) Cosa è la trasparenza alle transazioni?
 - 24) Cosa è la trasparenza alla scalabilità e perché dipende da trasparenza di migrazione e di replica?
 - 25) Cosa è la trasparenza alle prestazioni e perché dipende da trasparenza di migrazione, di replica e di persistenza?
 - 26) Cosa è la trasparenza ai malfunzionamenti e perché dipende dalla trasparenza alle transazioni?

Lezione 3:

- 1) Quale è la differenza tra un processo ed un thread? E quali le somiglianze?
- 2) Perché dire che un thread è un "processo light-weight" è tecnicamente scorretto, anche se molto diffuso nella prassi comune?
- 3) Quali sono esempi di applicazioni che hanno necessità di essere multithread?
- 4) Come usa un server il pool di thread per ottimizzare le prestazioni?
- 5) Illustrare il diagramma degli stati di un thread in Java
- 6) Come si può creare un thread in Java e quali sono i vantaggi e svantaggi delle due tecniche possibili?
- 7) Quando si invoca il metodo `.start()` su un thread cosa succede?
- 8) Quando si invoca il metodo `.run()` su un thread cosa succede?
- 9) Quando si invoca il metodo `.start()` su un thread, il thread viene eseguito: vero o falso?
- 10) Quando si invoca il metodo `.run()` su un thread, il thread viene eseguito: vero o falso?
- 11) Quale è la differenza tra un metodo `run()` e un metodo `start()` di un thread?
- 12) Cosa sono gli interrupt su un thread e quali metodi sono a disposizione per trattarli/gestirli?
- 13) Quali sono gli stati del diagramma di stato di un thread?
- 14) Un thread come passa dallo stato "New" a quello di "Runnable"?
- 15) Un thread come passa dallo stato "Runnable" a quello di "Timed Waiting" e come se ne esce?
- 16) Un thread come passa dallo stato "Runnable" a quello di "Waiting" e come se ne esce?
- 17) Un thread come passa dallo stato "Runnable" a quello di "Blocked" e come se ne esce? (maggiori dettagli vengono forniti nella lezione prossima con il concetto di lock di una risorsa)
- 18) Perchè lo stato "Runnable" è strutturato in due sottostati "Ready" e "Running" e come si passa dall'uno all'altro?
- 19) Quali sono i possibili tipi di errore che vengono generati dai thread?
- 20) Cosa è la interferenza tra thread? E come si risolve?
- 21) Cosa è la inconsistenza della memoria? E come si risolve?
- 22) Cosa è una race condition?
- 23) Perché i bug dovuti a race condition sono particolarmente complessi da trattare?
- 24) Cosa è la relazione "happens-before"?
- 25) Come faccio a stabilire una relazione "happens-before"?
- 26) Cosa significa che una variabile in memoria è dichiarata "volatile"?

Lezione 4:

- 1) Perché è necessaria la sincronizzazione?
- 2) Perché è necessaria la sincronizzazione efficiente?
- 3) Cosa sono i metodi sincronizzati?
- 4) Cosa sono i metodi statici sincronizzati?
- 5) Che relazione (dal punto di vista dell'accesso esclusivo) hanno un metodo sincronizzato dell'istanza e un metodo sincronizzato statico?
- 6) Rispondere sì/no e motivare la risposta per le seguenti domande:
 - 7) Un thread che è in esecuzione di un metodo sincronizzato dell'istanza blocca l'esecuzione di:
 - 8) un altro thread che esegue un metodo sincronizzato dell'istanza
 - 9) un altro thread che esegue un metodo statico sincronizzato
 - 10) un altro thread che esegue un metodo statico
 - 11) un altro thread che esegue un metodo dell'istanza
 - 12) tutti i thread in esecuzione sulla JVM
 - 13) tutti i thread in esecuzione su tutte le JVM in esecuzione sulla vostra macchina
 - 14) tutti i thread in esecuzione su tutte le JVM in esecuzione su tutti i vostri computer
 - 15) tutti i thread in esecuzione su tutte le JVM in esecuzione su tutti i computer dell'Università sorridente
 - 16) È possibile con i lock impliciti simulare i metodi sincronizzati di istanza
 - 17) È possibile con i lock impliciti simulare i metodi sincronizzati statici
 - 18) È possibile con i metodi sincronizzati di istanza simulare i lock

- 19) È possibile con i metodi sincronizzati statici simulare i lock
- 20) Cosa è una variabile volatile?
- 21) È possibile usare una variabile volatile per eliminare l'interferenza tra thread (race condition)
- 22) È possibile usare una variabile volatile per eliminare i problemi di inconsistenza della memoria
- 23) Se la variabile int a è volatile, allora la operazione a++ viene eseguita in mutua esclusione da parte di due thread: vero o falso. e perché?
- 24) Operazioni atomiche sono estremamente più efficienti delle operazioni non atomiche: vero/falso. e perché?
- 25) Cosa è il deadlock/Livelock/starvation?
- 26) Quale è la differenza tra deadlock e Livelock?

Lezione 5:

- 1) Sull'esempio di SimpleThread e Example, completare lo studio del comportamento per metodi istanza/statici, synchronized/no-synchronized, invocazione dei due thread dello stesso metodo o di metodi diversi
- 2) A cosa serve il design pattern del Singleton?
- 3) Cosa è la lazy allocation?
- 4) La soluzione al Singleton con il metodo getInstance() sincronizzato
- 5) è corretta? Se no, perché? Se sì, allora perché ne studiamo delle altre?
- 6) Perché la soluzione al Singleton con un blocco synchronized subito dopo l'if non funziona?
- 7) Cosa è il double-checked locking? Perché non è corretto? Come si può modificare per renderlo corretto?

Lezione 7:

- 1) Come funzionano i socket TCP in Java?
- 2) Quale è la differenza tra le classi ServerSocket e Socket in Java?
- 3) Cosa sono ed a cosa servono gli stream?
- 4) Cosa significa che gli stream vengono usati tipicamente come wrapper di altri stream?
- 5) Quali sono i metodi più importanti (e cosa fanno) di InputStream e di OutputStream?
- 6) Perché di solito i programmati non usano InputStream o OutputStream direttamente?
- 7) Come si fa a usare gli stream che sono associati ad un socket?
- 8) Quali sono i metodi offerti da ObjectInputStream (e ObjectOutputStream) per leggere (scrivere) un oggetto dallo (sullo) stream?
- 9) Descrivere il funzionamento del codice di un HelloWorld per socket TCP
- 10) Modificare il server HelloWorld rendendolo iterativo (inserendo cioè accept e risposta nel ciclo)
- 11) Modificare il server HelloWorld rendendolo concorrente: ad ogni accept si fa partire un thread, che si occupa di far partire la risposta al client, in modo che il server possa tornare a fare la accept subito
- 12) Descrivere il funzionamento del codice di Registro con i socket TCP
- 13) Cosa si deve fare per poter separare l'esempio di Registro in due progetti diversi, uno per il client e l'altro per il server?
- 14) Cosa si deve fare per rendere il server di Registro concorrente?

Lezione 8:

- 1) Come funziona il meccanismo stub-skeleton dell'esempio?
- 2) Quale è il ruolo della interfaccia remota nell'esempio?
- 3) Quale è il semplice "protocollo" di comunicazione tra stub e skeleton e che cosa "trascura" nella sua semplicità, che dovrebbe essere offerto per permettere la programmazione ad oggetti remoti?
- 4) Cosa manca all'esempio descritto per essere un vero e proprio strato di comunicazione software per oggetti distribuiti?
- 5) Perché usare i socket TCP (senza usare nessun meccanismo di invocazione remota), come ad esempio nell'esempio di Registro, non permette di riconoscere errori nel passaggio di parametri a tempo di compilazione? E quando (se) vengono riconosciuti? E che cosa succede?

- 6) Perché il problema degli errori nel passaggio di parametri, invece, nella invocazione di metodi (locali, ma anche remoti come vedremo) vengono riconosciuti a tempo di compilazione? E quale è il vantaggio per il programmatore?

Lezione 10:

- 1) In che maniera la esperienza di Jim Waldo all'interno della progettazione di CORBA ha influito su alcuni requisiti di progettazione di Java RMI?
- 2) Quale è il motivo dell'obiettivo di RMI come ambiente semplice? e come ambiente familiare al programmatore Java?
- 3) Quali sono i vantaggi dell'integrare il modello distribuito all'interno di un linguaggio di programmazione?
- 4) Perché la garbage collection rappresenta un utile strumento per la semplificazione dei compiti del programmatore?
- 5) Quali sono i rischi di memory leak? E perché sono problemi che sono particolarmente critici per i server?
- 6) Quali sono i vantaggi e gli svantaggi di un ambiente in cui la gestione della memoria è a carico del programmatore? e quando invece è a carico del sistema con un meccanismo di garbage collection?
- 7) Cosa sono le invocazioni unicast, multicast, di oggetti attivabili?
- 8) A cosa può servire la invocazione multicast?
- 9) A cosa può servire avere oggetti attivabili?
- 10) Perché è importante poter sostituire livelli di trasporto (in RMI ma anche in altri sistemi) senza dover modificare gli strati superiori, fino all'applicazione?
- 11) Cosa significa che una applicazione Java viene eseguita in una sandbox?
- 12) Quali sono i 4 livelli di sicurezza forniti da Java?
- 13) Per ciascuno dei 4 livelli di sicurezza, fornire l'obiettivo e alcuni esempi di funzionamento
- 14) Perché è necessario che il Bytecode Verifier "ripeta" i controlli che (ovviamente?) il compilatore ha già effettuato?
- 15) Quali sono i package in cui si struttura Java RMI e quali sono le loro funzioni?
- 16) Cosa è una interfaccia Remota?
- 17) Come vengono distinti i metodi remoti?
- 18) Cosa significa che la interfaccia remota aggiunge un ulteriore modificatore di accesso ai tradizionali valori in Java di public, private, etc. ?
- 19) Come devono essere i parametri di un metodo remoto?
- 20) Cosa si intende quando si dice che un oggetto locale è passato per copia invece che per riferimento?
- 21) Cosa garantisce la integrità referenziale?
- 22) Dare un esempio di integrità referenziale
- 23) Come si fa a localizzare un oggetto remoto?
- 24) Perché è importante che il compilatore forzi la gestione della eccezione di RemoteException per i metodi remoti?
- 25) Cosa è un metodo idempotente? e perché è utile come metodo remoto?

Lezione 11:

- 1) Perché gli oggetti remoti è bene che siano (quanto più possibile) simili agli oggetti locali?
- 2) Perché non è possibile la totale trasparenza, tra oggetti locali e remoti?
- 3) Cosa fa la classe RemoteObject? Cosa ridefinisce di Object?
- 4) Quale è la differenza nel passaggio di parametri remoti?
- 5) Descrivere la architettura a layer di Java RMI con i compiti di ciascun layer e la maniera in cui comunicano tra di loro
- 6) Quali sono i vantaggi di una architettura a layer? e quali gli svantaggi?
- 7) Cosa fa lo Stub&Skeleton/Remote-Reference/Transport layer?
- 8) In che maniera il marshalling è diverso dalla serializzazione?
- 9) Cosa specializza di ObjectOutputStream lo stream di Marshal?

- 10) Quali sono i passi che permettono di creare un oggetto server con Java RMI?
- 11) Quali sono i passi che permettono di creare un oggetto client con Java RMI?
- 12) Quali passi della creazione di un oggetto server devono essere effettuati prima di quella di un oggetto client?
- 13) A cosa serve rmic?
- 14) Una volta creata una applicazione client-server in Java RMI, quali sono i passi strettamente necessari se (1) si modifica la implementazione del server; (2) si modifica la implementazione del client; (3) si modifica la interfaccia?
- 15) Se client e server vengono scritti in due progetti diversi, indipendenti, quali sono i file che devono essere comuni tra di essi?
- 16) [E**] Provare a modificare il policy file in maniera da trovare il minimo insieme di regole di accesso che permette la esecuzione di HelloWorld
- 17) [P*] Scrivere l'esempio di HelloWorld senza il meccanismo del riuso della implementazione remota (subclassing di UnicastRemoteObject) ma con la classe di implementazione locale (uso del metodo exportObject()) (vedi cap. 3.3.1)
- 18) [P*] Usare due progetti separati per HelloWorld, usando la modalità di generazione di stub e skeleton con rmic

Lezione 14-15:

- 1) Qual è l'idea alla base del design pattern inversion of control?
- 2) Quali sono i vantaggi del "loose coupling, strong typing"?
- 3) In che modo il ciclo di vita di un bean differisce da quello di un POJO?
- 4) Quali sono i vantaggi derivanti dall'uso degli Interceptor?
- 5) In che modo è possibile definire una sorta di priorità nell'esecuzione di una catena di Interceptor?
- 6) In che modo è possibile realizzare disaccoppiamento nelle applicazioni Java enterprise?
- 7) Qual è il meccanismo che permette di scegliere fra due diverse implementazioni di uno specifico bean?
- 8) Perchè è stato introdotto il concetto di Interceptor Binding?
- 9) Qual è il vantaggio derivante dall'uso dei Decorator?
- 10) Qual è il vantaggio derivante dall'uso degli Eventi?

Lezione 17-18:

- 1) Qual è la differenza fra una entità ed un oggetto?
- 2) A cosa serve l'annotazione @GeneratedValue?
- 3) Qual è l'elemento discriminante per distinguere una entità da un POJO?
- 4) Qual è l'API fondamentale per la gestione delle operazioni sulle entità?
- 5) Quali sono le caratteristiche e le funzionalità più importanti della persistence unit?
- 6) Descrivere il ciclo di vita di una entità
- 7) Descrivere i tipi di relazioni in un database relazionale
- 8) Definizione e funzionalità di un Persistence Context
- 9) Descrivere i vari tipi di query definiti da JPQL