## **POJO (Plain old Java object):**

Definizione:

Descrive una classe che non ha bisogno di essere una sottoclasse di qualcosa, o implementare interfacce specifiche.

NON Estende classi prespecificate

NON Implementa l'interfaccia prespecificata

NON Contiene annotazioni prespecificate

Ha proprietà (private), e metodi di accesso pubblici **getter e setter** per le rispettive proprietà.

Esempio:

Un JavaBean è un **POJO** che è serializzabile, ha un costruttore senza argomenti e consente l'accesso a proprietà (private) utilizzando metodi getter e setter public  
public class MioBean {

private String alcunaProprieta;

public String getAlcunaProprieta()

{return alcunaProprieta;}

public void setAlcunaProprieta(String alcunaProprieta)

{this.alcunaProprieta = alcunaProprieta;}

}

## **JSON (JS Object Notation):**

Definizione:

Javascript Object notation, si tratta di un formato testuale, molto utilizzato per la strutturazione e scambio dati in applicazioni client server.

È basato su JavaScript, ed è una valida alternativa al formato XML e sempre più Web Services mettono a disposizione entrambe le possibilità di integrazione

Json si è fatto largo tra i vari protocolli e formati per lo scambio dati per la semplicità di implementazione.

JSON permette di aggregare dati ed è composto da proprietà di vario tipo:

-stringhe (alfanumeriche) / -numeri (con o senza virgola) / -valori booleani ( true e false)

Più oggetti JSON possono essere inseriti in una sorta di collezione che prende il nome di array.

Esempio:

{

"results": [

{

"gender": "male",

"name": {

"title": "mr",

"first": "samuel",

"last": "ross"  
 },

"professione": "ing",

},

….

]  
 }

## **JPS :**

Definizione:

Java Virtual Machine Process Status Tool è stato introdotto con java 1.5

Elenca le Java Virtual Machine (JVM) attive sul sistema di corrente.

Se jps viene eseguito senza specificare un hostid, cercherà JVM attive sull'host locale.

Inoltre ha quasi sostituito l’uso  **ps -ef | grep java** quando si lavora con Java in Unix o Linux.

Elenco delle JVM prodotte dal comando jps può essere limitato dalle autorizzazioni concesse all'entità che esegue il comando.

Esempio:

Listing the instrumented JVMs on the local host:

**jps**

18027 Java2Demo.JAR

18032 jps

18005 jstat

Listing the instrumented JVMs on a remote host:

**jps -l remote.domain**

3002 /opt/jdk1.7.0/demo/jfc/Java2D/Java2Demo.JAR

2857 sun.tools.jstatd.jstatd

**jps -lm**

## **CDA:**

Definizione:

Class Dependency Analyzer (CDA): **jdeps** è un nuovo strumento da riga di comando introdotto in JDK 8 per comprendere le dipendenze statiche e le librerie dell'applicazione, ad esempio il comando **jdeps** mostra le dipendenze a livello di pacchetto o di classe dei file di classe Java. L'input per jdeps può essere un nome percorso file .class, un file JAR o può essere un nome di classe completo per analizzare tutti i file di classe. Ogni volta che forniamo input allo strumento da riga di comando jdeps, questo genera le dipendenze alla console di sistema. jdeps introdotto con varie opzioni che influenzano l'output. Come da opzione, lo strumento da riga di comando jdeps genera l'output.

Esempio:

**jdeps -dotoutput C:\Users\DubeyBis\Desktop\jdeps Geeks.class**

## **Mockito:**

Definizione:

Mockito è un framework che permette di creare oggetti mock a partire sia da una interfaccia che da un classe semplicemente dichiarandone il comportamento.

Lo scopo dei test unitari è quello di verificare il funzionamento di determinate porzioni di codice, che possono essere eseguiti in modo automatico e ripetuti nel tempo.

i costrutti fondamentali del framework:

**mock() o @Mock:** consentono la definizione di un oggetto mock al quale è poi possibile

associare un comportamento ad esempio utilizzando i metodi when();

**spy() o @Spy:** consentono di realizzare mock parziale dell’oggetto permettendo quindi

di invocare metodi reali;

**verify():** consente di verificare (testare) la corretta invocazione dei metodi.

Comparazione / Differenze con… :

Framework JUnit ( standard)

Esempio:

( tipi di test: unitari, integrazione, funzionali, regressione etc)

Per *mockare* il Dao ed eseguire comunque i metodi del service senza interagire con il db il codice necessario è il seguente:

DaoClass dao = Mockito.mock( DaoClass.class );

ServiceClass service = new ServiceClass(dao);

boolean result = service.query( "select \* from t" );

System.out.println( "Result: " + result );

## **Selenium:**

Definizione:

Selenium è il più diffuso tool open-source per l’automazione di browser permette quindi di specificare istruzioni per automatizzare le interazioni con il sito o l’applicazione web.

Principali componenti:

-**Selenium WebDriver**, l’ambiente di sviluppo che permette di registrare,

modificare e debuggare test per web application.

E’ come un driver per browser e non necessita di un server per eseguire i comandi.

-**Selenium Grid**, un server che permette ai test di utilizzare delle istanze di web browser su macchine remote, quindi lanciare test in parallelo.

## 

## **Funzioni LAMBDA:**

Definizione:

Le espressioni lambda sono una nuova e importante funzionalità inclusa in **Java SE 8**.

In poche parole, si tratta di un metodo senza una dichiarazione, una sorta di scorciatoia che consente di scrivere un metodo nello stesso posto dove ti serve.

Le lambda expressions sono particolarmente utili nei casi in cui serve definire una breve funzione che ha poche linee di codice e che verrà utilizzata una sola volta.

- Le classi anonime interne forniscono un modo per implementare classi che vengono utilizzate una sola volta

-Le interfacce funzionali (Java 8), per es. ActionListener, EventListener, Runnable, Comparator o FileFilter, sono caratterizzate dalla presenza di un solo metodo e si prestano bene per l'utilizzo delle Lambda expression.

**( Lista Parametri metodo interfaccia funzionale) ->**

**{ Blocco istruzioni metodo interfaccia funzionale }**

Esempio:

es. ordinamento lista interi usando **Comparator** (inferfaccia funzionale - compare() )

Normale:

.................

Collections.sort(numbers, new Comparator<Integer>(){

@Override

public int compare(Integer o1, Integer o2) {

return o1.compareTo(o2); }

Lambda expression:

…….

Collections.sort(numbers, (Integer o1, Integer o2) -> {

return o1.compareTo(o2); } );

-----------------------

**(Lista degli argomenti) -> Espressione** oppure **(Lista degli argomenti)->{ istruzioni; }**

Esempi:

// espressione che prende in input due interi e restituisce la somma

**(int x, int y) -> x + y**

// espressione che prende in input una stringa e restituisce la sua lunghezza

s -> s.length()

// espressione senza argomenti che restituisce il valore 50

() -> 50

// espressione che prende in input una stringa e non restituisce nulla

(String s) -> { System.out.println("Benvenuto ");

System.out.println(s);

}

-----------------

// Definizione di un'interfaccia funzionale

interface Operazione {

int calcola(int a, int b);

}

// Utilizzo di un'espressione lambda per implementare l'interfaccia Operazione

Operazione somma = (a, b) -> a + b;

// Utilizzo dell'espressione lambda per eseguire l'operazione di somma

int risultato = somma.calcola(5, 3); // risultato = 8

## **Parallel Stream:**

Definizione:

Il metodo **parallelStream**() ( contenuto nelle Stream API di java 8), ci semplifica la vita perché esegue le operazioni sugli elementi in parallelo.

Gli **Stream** non sono riusabili perchè sono pensati per poter eseguire una serie finita di operazioni sui loro elementi;

E’ possibile quindi istanziare uno Stream e usare la sua referenza con con tutte le intermediate operation, ma l’esecuzione di una terminal operation lo rende inacessibile.

Inoltre un parallel stream può essere riconvertito in sequenziale in ogni momento con il metodo **sequential()**.

Regola :

Le intermediate operation ( es: **filter(), distinct()**) vanno messe sempre prima delle operazioni e sono lazy, vengono eseguite solo se sono necessarie per l’esecuzione della terminal operation.

es. terminal operation listacolori.stream().sorted().anyMatch()

Esempio:

il seguente codice crea un parallel stream di numeri interi:

List<Integer> numeri = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

int somma = numeri.parallelStream()

.filter(n -> n % 2 == 0)

.mapToInt(Integer::intValue)

.sum();

In questo esempio, la lista di numeri viene trasformata in un parallel stream chiamando il metodo "parallelStream()". Successivamente, viene applicata una serie di operazioni di filtraggio e di mappatura utilizzando i metodi "filter()" e "mapToInt()", rispettivamente. Infine, viene calcolata la somma degli elementi del flusso utilizzando il metodo "sum()".

**La principale differenza tra un parallel stream e un normale stream** è che un parallel stream può eseguire le operazioni in parallelo, suddividendo il flusso di dati in sottoinsiemi e utilizzando più thread per elaborarli contemporaneamente. Tuttavia, l'utilizzo di un parallel stream può comportare dei costi aggiuntivi, come la sincronizzazione dei thread, il che significa che non sempre è la scelta migliore.

## **Thread / MultiTheread:**

Definizione:

Un Thread è un processo che appartiene ad un programma o ad un altro processo.

Due o più thread possono condividere, indipendentemente dai dati, il codice che essi eseguono.

Questo avviene quando tali thread eseguono il loro codice da istanze della stessa classe.

Impl Interfaccia Runnable

Esempio:

## **Session:**

Definizione:

Per sessione si intende un gruppo di interazioni con un sito web in un determinato arco di tempo.

Una sessione può essere considerata come un contenitore per le azioni intraprese da un utente sul sito.

Ad esempio, una singola sessione può contenere più visualizzazioni di pagina, eventi, interazioni ...etc

Un singolo utente può aprire più sessioni, che possono verificarsi nello stesso giorno o nell'arco di diversi giorni...

Per impostazione predefinita, una sessione termina dopo 30 minuti di inattività, ma è possibile modificare questo valore...etc

## **DNS:**

Definizione:

DNS (Domain Name System) è un sistema che consente di tradurre gli indirizzi web, cioè i nomi di dominio, in indirizzi IP, che sono i veri e propri indirizzi numerici utilizzati dai computer per comunicare su Internet.

Quando si digita un nome di dominio (ad esempio "google.com") nella barra degli indirizzi del browser, il computer invia una richiesta al server DNS per ottenere l'indirizzo IP associato a quel nome di dominio.

Il server DNS risponde fornendo l'indirizzo IP corrispondente, permettendo così al browser di stabilire la connessione con il server corretto e di caricare il sito web richiesto.

## **HTTP / HTTPS / Header:**

Definizione:

L’Hyper Text Transfer Protocol (HTTP) e’ un protocollo di comunicazione generalmente utilizzato per il trasferimento delle pagine web in internet.

Un server HTTP generalmente resta in ascolto delle richieste dei client sulla porta 80 usando il protocollo TCP a livello di trasporto.

Le specifiche del protocollo sono gestite dal World Wide Web Consortium (W3C).

L'HTTP è un protocollo che lavora con un'architettura di tipo client/server:

il client esegue una richiesta e il server restituisce la risposta mandata da un altro host.

Nell'uso comune il client corrisponde al browser ed il server la macchina su cui risiede il sito web.

**HTTP / HTTPS**

La principale differenza tra i due protocolli è che **HTTPS** utilizza un protocollo di sicurezza SSL/TLS (Secure Sockets Layer/Transport Layer Security) per cifrare i dati trasmessi tra il client e il server, garantendo così una maggiore sicurezza.

Mentre HTTP trasmette i dati in chiaro, rendendoli potenzialmente vulnerabili ad attacchi di hacker e malware, HTTPS cripta i dati durante la trasmissione, rendendoli molto più sicuri.

HTTP differisce da altri protocolli come per es. FTP, per il fatto che le connessioni vengono generalmente chiuse una volta che una particolare richiesta (o una serie di richieste correlate) è stata soddisfatta

Vi sono quindi due tipi di messaggi HTTP:

messaggi richiesta e messaggi risposta.

**Messaggio di richiesta:**

Il messaggio di richiesta è composto di quattro parti:

-riga di richiesta (request line);

-sezione header (informazioni aggiuntive);

-riga vuota (CRLF: i 2 caratteri carriage return e line feed);

-body (corpo del messaggio).

Il metodo di richiesta, per la versione 1.1, può essere uno dei seguenti:

GET

POST

HEAD

PUT

DELETE

I metodi HTTP più comuni sono GET, HEAD e POST.

* **Il metodo GET** è usato per ottenere il contenuto della risorsa indicata come URI (come può essere il contenuto di una pagina HTML).
* **Il metodo POST** è usato di norma per inviare informazioni al server (ad esempio i dati di un form). In questo caso l'URI indica che cosa si sta inviando e il body ne indica il contenuto.

Gli header di richiesta più comuni sono:

Host: nome del server a cui si riferisce l'URL. È obbligatorio nelle richieste conformi HTTP/1.1 perché permette l'uso dei virtual host basati sui nomi.

User-Agent: identificazione del tipo di client: tipo browser, produttore, versione...

Cookie: utilizzati dalle applicazioni

**Messaggio di risposta:**

Il messaggio di risposta è di tipo testuale ed è composto da quattro parti:

-riga di stato (status-line);

-sezione header;

-riga vuota (CRLF: i 2 caratteri carriage return e line feed);

-body (contenuto della risposta).

I codici di risposta più comuni sono:

**200 OK.** Il server ha fornito correttamente il contenuto nella sezione body.

**301 Moved Permanently.** La risorsa che abbiamo richiesto non è raggiungibile perché è stata spostata in modo permanente.

**302 Found**. La risorsa è raggiungibile con un altro URI indicato nel header Location. Di norma i browser eseguono la richiesta all'URI indicato in modo automatico senza interazione dell'utente.

**400 Bad Request**. La risorsa richiesta non è comprensibile al server.

**404 Not Found**. La risorsa richiesta non è stata trovata e non se ne conosce l'ubicazione. Di solito avviene quando l'URI è stato indicato in modo incorretto, oppure è stato rimosso il contenuto dal server.

**500 Internal Server Error**. Il server non è in grado di rispondere alla richiesta per un suo problema interno.

Esempio:

## **Services REST:**

Definizione:

Le applicazioni basate su REST, si definiscono RESTful e utilizzano le richieste HTTP per inviare i dati (creazione e/o aggiornamento), effettuare query, modificare e cancellare i dati. In definitiva, REST utilizza HTTP per tutte e quattro le operazioni CRUD (Create / Read / Update / Delete).

Proprio come un Web Services, un servizio REST è:

* Indipendente dalla piattaforma
* Indipendente dalla lingua
* Basati su standard (HTTP in primis)
* Può essere facilmente utilizzato in presenza di firewall

Una buona progettazione REST non contempla cookies: la “ST” di “REST” sta per “State Transfer”, e in effetti, una richiesta REST è del tutto autonoma e indipendente dalle altre, cioè, ogni richiesta porta con sé (durante la comunicazione client/server) tutte le informazioni che il server ha bisogno per completarla. Una richiesta non deve richiedere al server il recupero di un contesto o stato dell’applicazione.

Un client REST devo includere nelle intestazioni o nel corpo HTTP tutti i parametri, contesto e dati necessari al server per generare una risposta.

Comparazione / Differenze con… :

REST e Web Service, quali differenze?

REST non offre metodi per gestire sicurezza, crittografia, gestione delle sessioni, garanzie di QoS, proprio come un Web Services ma possono facilmente essere incluse nell’header HTTP

REST vs SOAP

Una bella analogia di paragone per REST vs SOAP è l’invio di una lettera:   
con SOAP, si utilizza una busta;   
con REST, una cartolina.   
Le cartoline sono più facili da maneggiare, sprecano meno carta (quindi, consumano meno larghezza di banda), e hanno in genere brevi contenuti.

(Naturalmente, le richieste non hanno limiti in lunghezza, se usano POST anziché GET).

REST è altrettanto sicuro come SOAP.

In particolare, una richiesta REST può essere effettuata su Secure Sockets

( protocollo HTTPS) e il contenuto può essere crittografato utilizzando qualsiasi meccanismo.

Senza crittografia, REST e SOAP sono entrambi insicuri.

Esempio:

## **WEB Services SOAP:**

Definizione:

SOAP è un protocollo per lo scambio di messaggi tra componenti software.

SOAP dunque è uno dei protocolli che abilitano i servizi Web.

SOAP può operare su differenti protocolli di rete, ma HTTP è il più comunemente utilizzato

SOAP si basa sul metalinguaggio XML e la sua struttura segue la configurazione head-body, analogamente ad HTML

Esempio:

## **SOA (Service-Oriented Architecture):**

Definizione:

Con la locuzione (SOA) si indica generalmente un'architettura software adatta a supportare l'uso di servizi Web per garantire l'interoperabilità tra diversi sistemi

così da consentire l'utilizzo delle singole applicazioni come componenti del processo di business e soddisfare le richieste degli utenti in modo integrato e trasparente;

cioè è il tipo di architettura fondata sull'applicazione dell'orientamento ai servizi.

## **Enterprice Service BUS :**

Definizione:

Un ESB ha la funzionalità di coordinare, orchestrare i vari applicativi per svolgere le funzioni di business.   
E’ un'infrastruttura software che fornisce servizi di supporto a SOA (service-oriented architecture) complesse.

Si basa su sistemi disparati, interconnessi con tecnologie eterogenee, e fornisce in maniera consistente servizi di coordinamento, sicurezza, messaggistica, instradamento intelligente e trasformazioni, agendo come una dorsale attraverso la quale viaggiano servizi software e componenti applicativi.  
I servizi SOA devono essere in grado di comunicare tra di loro attraverso un canale di comunicazione: il SOA bus.

Da un punto di vista architetturale il SOA bus è un layer che deve mettere a disposizione uno strato di comunicazione tra i servizi.

Scopo dell‘ESB è fornire un‘infrastruttura che centralizzi funzionalità quali supporto alla comunicazione sincrona ed asincrona basata su messaggi, intelligent routing, supporto alla trasformazione dei dati, supporto alla connettività verso EIS eterogenei, ...

**Non è legato ad una specifica tecnologia.**

Può essere realizzato usando una vasta gamma di tecnologie…

Esempio:

Ggli ESB sono contenitori di servizi: parliamo quindi di ESB container.

I Container forniscono le facilities nella gestione dei servizi (es: ciclo di vita), nelle modalità di comunicazione (es: gestione degli endpoint dei messaggi e modalità di inoltro) e di gestione delle risorse (es: resource pool); inoltre forniscono tipiche funzionalità infrastrutturali di middleware come monitoring, logging, audit.

Gli ESB espongono tramite "standard" funzionalità

Esempi di standard ampiamente utilizzati sono: JDBC, JMS, JMX , WSDL, SOAP, JAX-RPC, JAXM, JAXR, SSAJ, XQuery, XPath, JCA, UDDI, JBI, BP4WLS,...

Centralizza le informazioni provenienti da sistemi differenti, che ciascuno espone attraverso un’interfaccia di qualche tipo (HTTP, JSON, AMQP, XML, SOAP, FTP, CSV, in effetti non importa molto quale).

## **APACHE CAMEL :**

Definizione:

*Apache Camel 2 è un Integration Framework, ovvero uno strumento che permette a sistemi software complessi di scambiarsi messaggi o utilizzare servizi anche se essi sono implementati su ambienti, piattaforme o linguaggi diversi tra loro.*

Perché questi questi servizi collaborino c’è bisogno di integrare un sistema di comunicazione che consenta lo scambio di messaggi che presentino:

* diverse forme (con determinati protocolli);
* diverse modalità (ad esempio sincrona o asincrona).

A risolvere questo problema (fornendo un contesto dove avviene l’interazione e una serie di servizi di ‘appoggio’) ci sono i cosiddetti **ESB** sia gratuiti (come Mule o Apache serviceMix), sia a pagamento (Bea Aqualogic Bus).

Pur non rientrando nella categoria di ESB, gli Integration Framework (come Apache Camel o Spring Integration Framework) rappresentano comunque strumenti efficaci nel risolvere il problema in questione.

**Apache Camel non è un ESB poichè non ha un proprio Container** dove viene eseguito, nè offre tutta una serie di servizi che tipicamente un ESB, offre eppure è altrettanto efficace per quanto riguarda il lavoro per il quale è stato realizzato che riguarda l’orchestrazione tra servizi, l’interazione attraverso lo scambio di messaggi , monitoring, creazione di componenti per la trasformazione dei messaggi ecc.

Fatto sta che Camel viene usato come strumento di routing da Apache serviceMix stesso (come motore interno per il routing).

Esempio:

## **JMS code:**

Definizione:

Java Message Service o JMS è un insieme di API che consentono lo scambio di messaggi tra applicazioni java distribuite sulla rete.

Con il termine messaging si intende un meccanismo che consente la comunicazione asincrona tra client remoti in modalità loosely coupled.

Le componenti caratteristiche di un sistema conforme alla specifica JMS sono:

**JMS Client:** programmi che si possono comportare come producer, inviano messaggi o consumer, ricevono messaggi;

***JMS provider:*** sistema di messaggistica che implementa JMS e realizza delle funzionalità aggiuntive per l’amministrazione e il controllo della comunicazione attraverso messaggi;

**Administered Object:** sono oggetti JMS preconfiguarti, creati da un amministratore ad uso dei client. Incapsulano la logica specifica del JMS provider nascondendola ai client, garantendo maggiore portabilità al sistema complessivo.

Possono essere:

**Connection Factory:** oggetto utilizzato da un client per realizzare la connessione con il

provider;

**Destination (queue/topic):** oggetto utilizzato da un client per specificare la destinazione

dove il messaggio è spedito/ricevuto.

**Il Modello di Messaggistica**

I possibili modelli di messaggistica disponibili in JMS sono due:

* point-to-point
* publish/subscribe.

Un JMS provider standalone può implementarne solamente uno o entrambe, ma uno conforme alla specifica J2EE deve implementarli entrambe.

**Point-to-point (PTP)**

Questa tipologia di comunicazione si fonda sul concetto di queue (coda) che il producer (chiamato sender) utilizza per indirizzare i propri messaggi.

Le code conservano tutti i messaggi inviati fino a quando il consumer (chiamato receiver) lo legge oppure il messaggio scade.

Un messaggio può essere consumato da un solo receiver, il quale può leggerlo in qualsiasi momento indipendentemente dallo stato che ha il produce al momento della lettura o dallo stato che esso aveva quando il producer ha inserito il messaggio nella coda.

Più producer e più consumer possono condividere la stessa coda, ma ogni messaggio può essere letto da un solo consumer il quale deve inviare la notifica dell’avvenuta ricezione e processamento del messaggio (acknowledgement).

I messaggi sono consegnati nell’ordine di invio ed il provider garantisce che siano consegnati una ed una sola volta.

Il modello PTP si rivela utile quando è necessario garantire che un messaggio arrivi ad un solo destinatario che notifichi la corretta ricezione.

**Publish/Subscribe (Pub/Sub)**

Questa tipologia di comunicazione si fonda sul concetto di topic, una sorta di bacheca.

I client, che sono generalmente anonimi, si distinguono in publisher (i producer) e subscriber (i consumer).

Il messaggio pubblicato dai publisher viene preso in carico dal sistema il quale si occupa di conservarlo fino a quando tutti i subscriber, che si siano dichiarati interessati, lo hanno ricevuto.

In questo tipo di comunicazione esiste una relazione temporale per cui un subscriber può ricevere solamente messaggi inviati dal publisher dopo la sottoscrizione, così come può continuare a riceverne solamente mantenendo attiva tale sottoscrizione.

Non vi è garanzia sull’ordine di ricezione dei messaggi, così come non vi è garanzia che il messaggio sia ricevuto una sola volta.

Esempio:

L’annotazione **@Resource** indica al container di iniettare il riferimento alla risorsa richiesta. L’attributo name identifica la risorsa attraverso il suo nome JNDI.

Nella classe è utilizzata in due occasioni:

per l’iniezione della coda JBQ identificata dal JNDI name java:/jms/queue/javaboss;

per l’iniezione della classe ConnectionFactory utilizzata per la creazione della connessione verso il provide JMS.

@Singleton

public class JBQWriter {

**@Resource(name = "java:/jms/queue/javaboss")**

private Queue jbq;

**@Resource(mappedName = "java:/JmsXA")**

private ConnectionFactory cf;

private Connection connection;

public void sendMessage(String txt) {

try {

connection = cf.createConnection();

Session session = connection.createSession(false, Session.AUTO\_ACKNOWLEDGE);

MessageProducer publisher = session.createProducer(jbq);

connection.start();

TextMessage message = session.createTextMessage(txt);

publisher.send(message);

}

catch (Exception exc) {

exc.printStackTrace();

}

finally {

if (connection != null) {

try {

connection.close();

} catch (JMSException e) {

e.printStackTrace();

}

}

}

}

}

## **Design Pattern:**

Definizione:

Un design pattern può essere definito "una soluzione progettuale consolidata generale ad un problema ricorrente in un dato contesto".

Si tratta di una descrizione o modello logico da applicare per la risoluzione di un problema che può presentarsi in un dato contesto.

Classificazione: Pattern creazionali, Pattern strutturali

**- Pattern creazionali:**

( I pattern creazionali risolvono problematiche inerenti l'istanziazione degli oggetti )

-Lazy initialization

è la tattica di istanziare un oggetto solo nel momento in cui deve essere usato per la prima volta. È utilizzato spesso insieme al pattern factory method.

- Il Prototype pattern

permette di creare nuovi oggetti clonando un oggetto iniziale, o prototipo.

**- Il Singleton**

ha lo scopo di assicurare che di una classe possa essere creata una sola istanza.

**- Pattern strutturali:**  
 (pattern strutturali risolvono problematiche inerenti la struttura delle classi e degli oggetti)

- L'Adapter: converte l'interfaccia di una classe in una interfaccia diversa.

- Il Container: offre una soluzione alla rottura dell'incapsulamento per via dell'uso   
 dell'ereditarietà.

-Il Decorator: consente di aggiungere metodi a classi esistenti durante il run-time (cioè durante lo svolgimento del programma), permettendo una maggior flessibilità nell'aggiungere delle funzionalità agli oggetti.

-Proxy: consente di posticipare l'accesso o creazione al momento in cui sia davvero richiesto.

Esempio:

MVC, IoC

## **Spring Framework:**

Definizione:

Spring è un framework Java utilizzato per lo sviluppo di applicazioni enterprise;

è nato come alternativa flessibile e “leggera” alla specifica Java EE.

L’obiettivo originario del framework era la semplificazione.

Spring dà la possibilità di creare applicazioni complesse senza la necessità di rispettare pienamente la specifica EJB, ma usando semplici oggetti Java (POJO).

Il cuore di Spring è un motore di Inversion of Control che consente d’implementare applicazioni solide basate su componenti estremamente testabili e riutilizzabili.

Main Projects:

**Spring IO platform**

**Spring Framework**

**Spring Cloud**

**Spring Boot**

**Spring data**

**Spring for Android**

**Spring Mobile**

**Spring web services**

**IoC** invece inverte il control flow facendo in modo che non sia più lo sviluppatore a doversi preoccupare di questi aspetti, ma il framework, che reagendo a qualche “stimolo” se ne occuperà per suo conto.

Esempio:

## **IoC / MVC:**

Definizione:

**Model-View-Controller** (abbreviato spesso in MVC), che consiste nel separare i componenti software che implementano il modello delle funzionalità di business (model),

dai componenti che implementano la logica di presentazione (view) e da quelli di controllo che tali funzionalità utilizzano (controller).

Model-View-ViewModel (spesso abbreviato in MVVM)

Comparazione / Differenze con… :

Esempio:

## **Dependency Injection (DI):**

Definizione:

Dependency Injection (DI) per riferirsi ad una specifica implementazione dello IoC rivolta ad invertire il processo di risoluzione delle dipendenze, facendo in modo che queste vengano iniettate dall’esterno.

Quindi assegnare ad un oggetto le sue variabili di instanza, banalmente la classe A si dice dipendente dalla classe B se ne usa in qualche punto i servizi offerti.

Perchè?

disaccoppiamento delle classi configurando dall'esterno ....

decidere a runtime quali sono le classi che implementano poi una richiesta del sistema.

L’idea alla base della Dependency Injection è quella di avere un componente esterno (assembler) che si occupi della creazione degli oggetti e delle loro relative dipendenze e di assemblarle mediante l’utilizzo dell’injection.

In particolare esistono tre forme di injection:

1) Constructor Injection, dove la dipendenza viene iniettata tramite l’argomento del costruttore

2) Setter Injection, dove la dipendenza viene iniettata attraverso un metodo “set”

3) Interface Injection che si basa sul mapping tra interfaccia e relativa implementazione (non utilizzato in Spring)

Esempio:

**@Required**

L'annotazione **@Required** si applica ai metodi di impostazione delle proprietà del bean.

**@Autowired**

L'annotazione @Autowired può essere applicata a metodi setter di proprietà del bean, metodi non setter, costruttore e proprietà.

**@Qualifier**

L'annotazione @Qualifier insieme a @Autowired può essere utilizzata per rimuovere la confusione specificando quale bean esatto verrà cablato.

**@Configuration**

Questa annotazione viene utilizzata sulle classi che definiscono i bean.

@Configuration è un analogo per il file di configurazione XML: è la configurazione che utilizza la classe Java.

La classe Java annotata con @Configuration è una configurazione a sé stante e avrà metodi per istanziare e configurare le dipendenze.

**@Bean**

Questa annotazione viene utilizzata a livello di metodo.

L'annotazione **@Bean** funziona con @Configuration per creare bean Spring.

Come accennato in precedenza, @Configuration avrà metodi per istanziare e configurare le dipendenze.

Tali metodi verranno annotati con **@Bean**.

Il metodo annotato con questa annotazione funziona come **ID bean** e crea e restituisce il bean effettivo.

**@Controller**

L'annotazione @Controller viene utilizzata per indicare che la classe è un controller Spring.

Questa annotazione può essere utilizzata per identificare i controller per Spring MVC o Spring WebFlux.

**@Service**

Questa annotazione viene utilizzata su una classe.

@Service contrassegna una classe Java che esegue alcuni servizi, come eseguire la logica di business, eseguire calcoli e chiamare API esterne.

Questa annotazione è una forma specializzata dell'annotazione @Component destinata ad essere utilizzata nel livello di servizio.

**@RestController**

Questa annotazione viene utilizzata a livello di classe.

L'annotazione @RestController contrassegna la classe come controller in cui ogni metodo restituisce un oggetto di dominio anziché una vista.

Annotando una classe con questa annotazione non è più necessario aggiungere @ResponseBody a tutto il metodo RequestMapping.

Significa che non usi più view-resolver né invii html in risposta.

Devi solo inviare l'oggetto dominio come risposta HTTP nel formato che è compreso dai consumatori come JSON.

**@SessionAttribute**

Questa annotazione viene utilizzata a livello di parametro del metodo.

L'annotazione @SessionAttribute viene utilizzata per associare il parametro del metodo a un attributo di sessione.

Questa annotazione fornisce un comodo accesso agli attributi di sessione esistenti o permanenti.

**@SessionAttributes**

Questa annotazione viene applicata a livello di tipo per un gestore specifico.

L'annotazione @SessionAtrributes viene utilizzata quando si desidera aggiungere un oggetto JavaBean in una sessione.

Viene utilizzato quando si desidera mantenere l'oggetto in sessione per un breve periodo. @SessionAttributes viene utilizzato insieme a @ModelAttribute.

Considera questo esempio.

**@Lazy**

Questa annotazione viene utilizzata sulle classi di componenti.

Per impostazione predefinita, tutte le dipendenze cablate automaticamente vengono create e configurate all'avvio.

Ma se vuoi inizializzare un bean pigramente, puoi usare l'annotazione @Lazy sulla classe.

Ciò significa che il bean verrà creato e inizializzato solo quando viene richiesto per la prima volta.

È inoltre possibile utilizzare questa annotazione sulle classi @Configuration.

Ciò indica che tutti i metodi @Bean all'interno di quella @Configuration dovrebbero essere inizializzati lentamente.

**@Value**

Questa annotazione viene utilizzata a livello di campo, parametro costruttore e parametro metodo.

L'annotazione @Value indica un'espressione di valore predefinito per il campo o il parametro con cui inizializzare la proprietà.

Poiché l'annotazione @Autowired dice a Spring di iniettare un oggetto in un altro quando carica il contesto dell'applicazione,

puoi anche utilizzare l'annotazione @Value per iniettare valori da un file di proprietà nell'attributo di un bean.

Supporta both #{...} and ${...} placeholders.

## **J2EE / EJB:**

Definizione:

J2EE sta per Java 2 Enterprise Edition

Un’applicazione di tipo Enterprise è incentrata fortemente sulla risoluzione di problemi di business cioè una Enterprise Application è un’applicazione che soddisfa requisiti di sicurezza, performance e affidabilità.  
Si basano su Application Server, ovvero fornisce dei servizi di fondamentale importanza, ad esempio la facilitazione della comunicazione, la gestione delle connessioni ad un database e quella delle transazioni.

Esempio:

## **Garbage Collector:**

Definizione:

è un thread Daemon che sta in esecuzione in background.

Libera la memoria heap distruggendo gli oggetti non più raggiungibili.

Gli oggetti non raggiungibili sono quelli a cui non fa più riferimento alcuna parte del programma

E’ un meccanismo che tiene traccia delle locazioni di memoria utilizzate e le libera solo quando non sono effettivamente più necessarie.

Principali fasi: marking, normal deletion, compacting

Tipi di Garbage Collector:

**Serial GC**

**Parallel / Throughput GC**

**Concurrent Mark Sweep (CMS) Collector**

**G1 Collector**

forzare il GC:

**System.gc ();**

**System.runFinalization ();**

che è semplicemente un suggerimento per il garbage collector che vuoi che faccia una raccolta ovvero forzarlo ad usare il metodo finalize() di oggetto non raggiungibile liberando la memoria assegnata e facendo ciò che dice il metodo finalize().

## **JSP / Servlet:**

Definizione:

JSP - Sono costituite da una combinazione di testo e tag: il testo rappresenta il contenuto informativo mentre i tag partecipano alla costruzione delle pagine dinamiche decidendone ad esempio la formattazione, eventuali controlli etc.

È possibile includere all’interno di una JSP delle parti di codice attraverso le cosiddette scriplet.

Il ciclo di vita di una JSP è piuttosto semplice: quando viene richiesta una pagina jsp al container web, questo provvede a convertire la pagina in una servlet e ad autorizzare la servlet a processare la richiesta utente, una volta che la pagina viene trasformata in una servlet, la stessa servlet verrà utilizzata per processare successive richieste.

È opportuno ricordare che la compilazione della jsp in servlet avviene tutte le volte che la jsp subisce delle modifiche.

## **JSF:**

Definizione:

Nelle applicazioni Java EE, in JSF la logica, solo apparentemente più complessa, facilita di gran lunga la costruzione del layer di presentazione.

In JSF ad ogni pagina (view) corrisponda un managed bean (che è un POJO), attraverso un particolare tipo di **Expression Language.**

JSF mette inoltre a disposizione la logica per la conversione dei tipi e la validazione attraverso l’utilizzo di convertitori e validatori.

Il ciclo di vita di Java Server Faces è d**i 6 diverse fasi:**

**Restore View:** è la fase di inizializzazione della view. Durante questa fase si

procede alla costruzione della view della pagina,

dei gestori degli eventi, dei validatori e dei componenti presenti

nella vista.

**Apply Request Values:** durante questa fase vengono estratti i valori dalla request

e convertiti nei tipi previsti dal managed bean.

**Process Validation:** è la fase di validazione dei valori inseriti dall’utente secondo le

logiche di validazione definite.

**Update Model Values:** vengono aggiornate le proprietà del managed bean con i

valori inseriti dall’utente se questi hanno superato la fase di

validazione.

**Invoke Application:** in questa fase vengono gestiti gli eventi applicative (quali ad

esempio il click su un bottone) .

**Render Response:** viene renderizzata la view.

Esempio:

## **Classe Astratta:**

Definizione:

Una classe astratta è una **classe che non può essere instanziata** e che serve solo per essere derivata, definendo al suo interno metodi e proprietà tutti o in parte anch'essi astratti.

ha almeno un metodo astratto, è obbligatoria la sua implementazione e non può essere istanziata.

Si usa una classe astratta per condividere codice fra più classi, se più classi hanno in comune metodi e campi o se si vogliono dichiarare metodi comuni che non siano necessariamente campi static e final

Le classi astratte permettono di implementare il concetto di polimorfismo, uno dei capisaldi della programmazione orientata agli oggetti.

Il polimorfismo consente di assegnare comportamenti differenti a instanze di classi differenti derivanti tutte dalla stessa classe base, in questo caso la classe astratta.

Per implementare il polimorfismo però i metodi e le proprietà del tipo base devono essere dichiarati come astratti o virtuali, affinchè questi possano essere riscritti (override) dal tipo derivato che ne instanzierà un particolare oggetto differente da un'altro ma con caratteristiche simili poichè derivanti entrambi dalla stessa classe madre, che ne definisce i comportamenti comuni tramite le signature dei metodi astratti.

I metodi astratti vengono solamente definiti nella classe astratta ed ognuno di esso deve essere necessariamente implementato nella classe derivata.

Comparazione / Differenze con… :

Entrambe non possono essere instanziate e possono dichiarare al loro interno metodi con o senza implementazione.

per esempio le classi astratte possono dichiarare campi che sono **non static e final,** e dichiarare metodi public, protected e private.

Invece con le interfacce tutti i campi sono automaticamente public, static e final e tutti i metodi che vengono dichiarati o definiti sono public.

Esempio:

Un esempio di utilizzo di classi astratte nella JDK è rappresentato dalla “**AbstractMap**“, ....

## 

## **Interfaccia:**

Definizione:

interfaccia solo firma dei metodi

Le interfacce sono molto simili alle classi astratte, in quanto anch'essa definisce metodi e proprietà astratte.

Nelle interfacce non troveremo l'implementazione di alcun metodo o proprietà, come per le classi astratte pure.

Si decide di utilizzare una interfaccia se ci si trova nella situazione in cui alcune classi (assolutamente non legate fra di loro) si trovano a condividere i metodi di una interfaccia, se si vuole specificare il comportamento di un certo tipo di dato (ma non implementarne il comportamento) o se si vuole avere la possibilità di sfruttare la “multiple inheritance”.

Comparazione / Differenze con… :

Entrambe non possono essere instanziate e possono dichiarare al loro interno metodi con o senza implementazione.

per esempio le classi astratte possono dichiarare campi che sono non static e final, e dichiarare metodi public, protected e private.

Invece con le interfacce tutti i campi sono automaticamente public, static e final e tutti i metodi che vengono dichiarati o definiti sono public.

Esempio:

Un esempio di classe che implementa molte interfacce nella JDK, invece, è rappresentato da **HashMap**, che implementa le interfacce Serializable, Clonable

## **Ereditarietà:**

Definizione:

Comparazione / Differenze con… :

Esempio:

## **Private / Public / Protected:**

Definizione:

Modificatore Stessa Classe Stesso Package Sottoclasse Dappertutto

public si si si si

protected si si si no

(default) si si no no

private si no no no

## **Final / Static :**

Definizione:

Final: questo modificatore è applicabile alle variabili, ai metodi e alle classi.  
 • una variabile dichiarata final diviene una costante   
 • un metodo dichiarato final non può essere riscritto in una sottoclasse (non è possibile   
 applicare l’override)

• una classe dichiarata final non può essere estesa

Static: questo modificatore può essere applicato a metodi e a variabili di una classe.

Un membro statico è condiviso da tutte le istanze della classe, quindi può essere

utilizzato anche senza istanziare la classe.   
 Il suo utilizzo provocherà il caricamento in memoria della classe contenente il   
 membro in questione e vivrà fino al termine del programma.

Esempio:

metodo sqrt() della classe Math, ovvero i: Math.sqrt(numero);

## **Polimorfismo:**

Definizione:

Il polimorfismo consente di assegnare comportamenti differenti a instanze di classi differenti derivanti tutte dalla stessa classe base, in questo caso la classe astratta.

Esempio:

## 

## **Generics:**

Definizione:

I Generics sono stati introdotti a partire dalla versione J2SE 5, il termine ha un significato di tipo parametrizzato.  
Aggiungono al linguaggio java la possibilità di 'parametrizzare' i tipi di dati gestiti ad esempio nei contenitori (Collection, Map, Set etc..).

Quindi drastica riduzione dell'uso dei casting espliciti ad opera dell'utente.

Esempio:

## **Archit Multi-tier / J2EE:**

Definizione:

In J2EE uno dei punti cardine è, certamente, quello dell’architettura a più strati (tiers).

In generale, se si pensa alla struttura di un’applicazione è possibile suddividerla logicamente in tre macro aree:

1. Visualizzazione dei dati ( interfaccia utente o presentation)

- web client: browser (Jsp / Servlet) → web tier → business  
 - application client; GUI Swing

1. Area di business gestisce il processamento dell’informazione.

- Il business tier: EJB: Entity Beans, i Session Beans e i Message Driven Beans.

1. Accesso ai dati (data access): salvataggio dell’informazione ed il suo reperimento.

Il Data Access Layer (EIS layer): RDBMS (Oracle)

J2EE fornisce dei container ad hoc sia per i componenti web che per quelli di business, quindi sono definiti dei container per gli Enterprise Java Beans, per le pagine JSP, per le Servlet e per i client J2EE.

Principali utilizzate:

**JDBC** (Java Database Connectivity) API: L’insieme di interface che consente alle applicazioni di interagire con i più svariati database.

**RMI** (Remote Method Invocation) : Ne abbiamo parlato in un articolo specifico. Consente di invocare metodi di oggetti remoti.

**EJB** (Enterprise Java Beans): Rappresentano i componenti il cui compito è quello di gestire la logica di business (ovvero il business tier).

**Servlet e JSP** (Java Server Pages) : I componenti, probabilmente, più noti ed utilizzati nelle applicazioni J2EE.

**JMS** (Java Message Service): Un insieme di API che fornisce un sistema di messaggistica destinato allo scambio di informazioni (messaggi) tra più componenti J2EE.

**JTA** (Java Transaction API): Consente ai componenti J2EE di eseguire transazioni distribuite.

**JNDI** (Java Naming and Directory Interface): È un insieme di API che consentono di interfacciarsi con i servizi di naming e directory.

Un servizio di naming e directory si occupa di fornire una soluzione per trovare ed identificare ed associare nomi con dati, servizi e risorse.

**JavaMail**: Un insieme di API che consente ai componenti J2EE di inviare e ricevere messaggi di posta elettronica in modo indipendente dalla piattaforma.

## **CRUD:**

Definizione:

CRUD (Create / Read / Update / Delete).

Esempio:

## **CAMEL:**

Definizione:

Esempio:

## **Hibernate:**

Definizione:

## **JPA (Java Persistence API):**

Definizione:

Le Java Persistence API, sono un framework per il linguaggio di programmazione Java che si occupa della gestione della persistenza dei dati di un DBMS relazionale nelle applicazioni che usano le piattaforme Java Platform, Standard Edition e Java Enterprise Edition.

JPA 2 è una specifica che semplifica l’interazione con i database a supporto della persistenza in Java.

JPA 2.0 è una specifica Java per l’accesso, la persistenza e la gestione dei dati tra

oggetti Java e database relazionali.

JPA fornisce un modello di persistenza per POJO, ed è utilizzabile da applicazioni Web o client, o anche all’esterno della piattaforma Java EE

JPA non è un prodotto, ma una specifica Java per mappare un POJO su un database, pertanto richiede un’implementazione:

ci sono diversi progetti che ne hanno implementato la specifica, alcuni commerciali e altri open source.

Il vantaggio offerto da JPA consiste nell’abilitare un mapping oggetti-relazioni attraverso annotazioni o XML standard, definendo come avviene il mapping tra classi Java e tabelle di un database relazionale.

JPA definisce inoltre le API di un EntityManager, il cui scopo è la gestione a runtime di queries e transazioni su oggetti resi persistenti.

Attualmente la maggior parte degli attori presenti nel campo della persistenza hanno rilasciato implementazioni delle specifiche JPA, inclusi Hibernate, TopLink, Kodo.

In scenari complessi diventa molto utile l’impiego dell’Object-Relational Mapping (**ORM**), ovvero la possibilità, tramite strumenti software, di associare le tabelle presenti nel DB a corrispondenti oggetti di dominio. Il mapping, inoltre, ha il vantaggio di riportare le relazioni logiche esistenti tra le tabelle anche a livello applicativo.

La specifica JPA (Java Persistence API), definisce proprio un’interfaccia applicativa utile a costruire sistemi ORM basati su oggetti Java.

Esistono molte librerie e framework che implementano la specifica JPA e che ci permettono di effettuare facilmente il mapping delle nostre tabelle su entità;

il più celebre è sicuramente il progetto Hibernate ORM.

Uno dei principali progetti “spin off” derivati da Spring è **Spring Data, framework** nato con l’obiettivo di fornire un modello standard per l’implementazione del data access layer all’interno di un progetto Spring.

JPA 2.0 version

Le principali caratteristiche aggiunte riguardano:

l’estensione delle funzionalità di mapping (ad esempio il supporto alla collezione di oggetti embedded e liste ordinate),

il supporto alla validazione,

Query Criteria API.

Il mapping in JPA può avvenire o tramite annotazioni o file XML.

Quello che consente JPA è ridurre la quantità di informazioni da fornire, in quanto è richiesto alle implementazioni di prevedere valori di default per quasi tutti gli aspetti del mapping.

Le altre informazioni saranno mappate di default, incluse il nome della tabella, i nomi delle colonne, cardinalità e relazioni, logica di persistenza e di accesso agli oggetti.

L’entità è dunque un oggetto del dominio della persistenza tipicamente associato a una tabella di un database relazionale e le istanze dell’entità solitamente corrispondono alle tuple della tabella.

Tra i programmi in esecuzione e il database si trova la cache:

Un PersistenceContext è in pratica una cache con la propria connessione (non condivisa) al database.

Un EntityManager rappresenta un PersistenceContext (e quindi una cache) e compito dell’EntityManagerFactory è creare un EntityManager (in definitiva una cache).

Il pattern **Repository** è un modello consolidato nelle applicazioni Enterprise.

Una classe Repository contiene tutto il codice relativo alla persistenza ma nessuna logica di business. Fornisce metodi per inserire, aggiornare e rimuovere un’entità e metodi che istanziano ed eseguono query specifiche.

L‘obiettivo di questo pattern è quello di separare il codice per la persistenza dal codice di business e a migliorare la riusabilità del codice relativo alla persistenza

Esempio:  
Il file **persistence.xml** va inserito in una cartella denominata META-INF, posta nella cartella src. Questo file è un file di configurazione standard

Le associazioni tra tabelle si traducono nel nostro Domain Model in annotazioni di tipo @ManyToMany, @ManyToOne oppure @OneToMany (in base al tipo di relazione) applicati ad attributi di tipo List o Set del tipo dell’entità associate.

## **Javascript /Ajax/JQuery:**

Definizione:

Esempio:

## **BootStrap:**

Definizione:

Esempio:

## **Serializable (interfaccia):**

Definizione:

L'interfaccia Serializable viene comunemente utilizzata dai programmatori Java per creare oggetti persistenti.

Normalmente infatti, un oggetto esiste fin tanto che la Java Virtual Machine (JVM) è attiva.

Il meccanismo di persistenza, attivato dalla serializzazione, permette invece di creare oggetti che esistono indipendentemente dal fatto che la JVM sia o meno in esecuzione.

## **JAR e WAR (diff):**

Definizione:

In sintesi, un file **JAR** è utilizzato per distribuire librerie e classi Java, mentre un file **WAR** è utilizzato per distribuire un'intera applicazione web basata su Java.

## **DB - 1 a 1 :**

Definizione:

Esempio:

## **DB - 1 a N / N a N:**

Definizione:

Esempio:

## **PL/SQL :**

Definizione:

Esempio:

## **SQL / For nkey /Unique Co :**

Definizione:

Esempio:

## **Maven / Gradle:**

Definizione:

Esempio:

## **TDD Agile:**

Definizione:

TDD: TEST DRIVEN DEVELOPMENT

Il Test Driven Development è una componente della filosofia Agile e permette di risolvere problemi comuni che nascono man mano che il progetto aumenta di dimensione.

Il TDD comprende 4 fasi:

* analisi
* design
* sviluppo
* testing.

In altre parole, significa comprendere il dominio del problema e analizzarlo, progettare un’applicazione per la soluzione del problema, svilupparla e testarla.

Per essere chiari TDD non significa solo creare unit test, ma è un termine che si riferisce all’intero processo di sviluppo.

Nel TDD si usa una sorta di pattern: RED – GREEN – REFACTOR

Vantaggi:

Riutilizzo del codice

Aumentare la qualità del software

Esempio:

**Scrivi un unit test destinato a fallire:** Deve essere il più semplice e chiaro possibile e deve essere

scritto con il proposito di fallire.

**Esegui il test:** Se il test fallisce, è il momento di scrivere il codice minimo per far passare il test

**Migliora il test:** Se credi che sia possibile migliorare il codice che hai scritto.