



Java Model 2, EJB e Spring

Home Page del corso: <http://lia.disi.unibo.it/Courses/twt2223-info/>
Versione elettronica: 4.01.Componenti.pdf
Versione elettronica: 4.01.Componenti-2p.pdf

Java Model 2

Nel progetto di applicazioni Web in Java, due modelli di ampio uso e di riferimento: **Model 1 e Model 2**

Come già detto, **Model 1 è un pattern semplice** in cui codice responsabile per presentazione contenuti è mescolato con logica di business

- Suggerito solo per piccole applicazioni (*sta diventando obsoleto nella pratica industriale*)

Model 2 come design pattern più complesso e articolato che **separa chiaramente livello presentazione dei contenuti dalla logica utilizzata per manipolare e processare contenuti stessi**

- Suggerito per applicazioni di medio-grandi dimensioni

Visto che Model 2 preme per separazione netta fra logica di business e di presentazione, **usualmente associato con paradigma Model-View-Controller (MVC)**

- Mai specificato in modo definitivo e “mandatory” come realizzare tecnicamente modello MVC in tecnologie Java
- Battaglia delle implementazioni
- Diverse soluzioni possibili (ad es. Java BluePrints suggerisce adozione di Enterprise Java Bean – EJB - per realizzare modello MVC)

Java Model 2

Il termine **Model 2** deriva da paper di Govind

(JavaWorld'99):

- Proposta di **separazione logica di business (servlet) da presentazione (JSP)**, con le due parti viste come "Controller" e "View" rispettivamente
- Parte di "**Model**" lasciata non specificata nell'architettura proposta da Govind (idea che ogni struttura dati possa essere adatta a realizzare modello: da Vector list a db relazionale)

Tecnologie alternative: ad esempio, **PHP Zend Framework, Ruby on Rails, Python Django, Python Zope, Java Spring, Java Struts**)

- altre alternative recenti, fra cui **Java Server Faces**

Spesso oggi due termini **Model 2 e MVC** vengono (impropriamente) usati come sinonimi fra gli sviluppatori

Architettura Model-View-Controller

Architettura adatta per applicazioni Web interattive (altre tecnologie e modelli più adatti per servizi asincroni, vedi Message Driven Bean e Java Messaging Service - JMS)

- **Model** – rappresenta livello dei dati, incluse operazioni per accesso e modifica. Model deve notificare view associate quando modello viene modificato e deve supportare:
 - possibilità per view di interrogare stato di model
 - possibilità per controller di accedere alle funzionalità incorporate da model
- **View** – si occupa di rendering dei contenuti di model. Accede ai dati tramite model e specifica come dati debbano essere presentati
 - aggiorna presentazione dati quando modello cambia
 - gira input utente verso controller
- **Controller** – definisce comportamento dell'applicazione
 - fa dispatching di richieste utente e seleziona view per presentazione
 - interpreta input utente e lo mappa su azioni che devono essere eseguite da model (in una GUI stand-alone, input come click e selezione menu; in una applicazione Web, richieste HTTP GET/POST)

Java Model 2

Mapping possibile su applicazioni Web Java-based

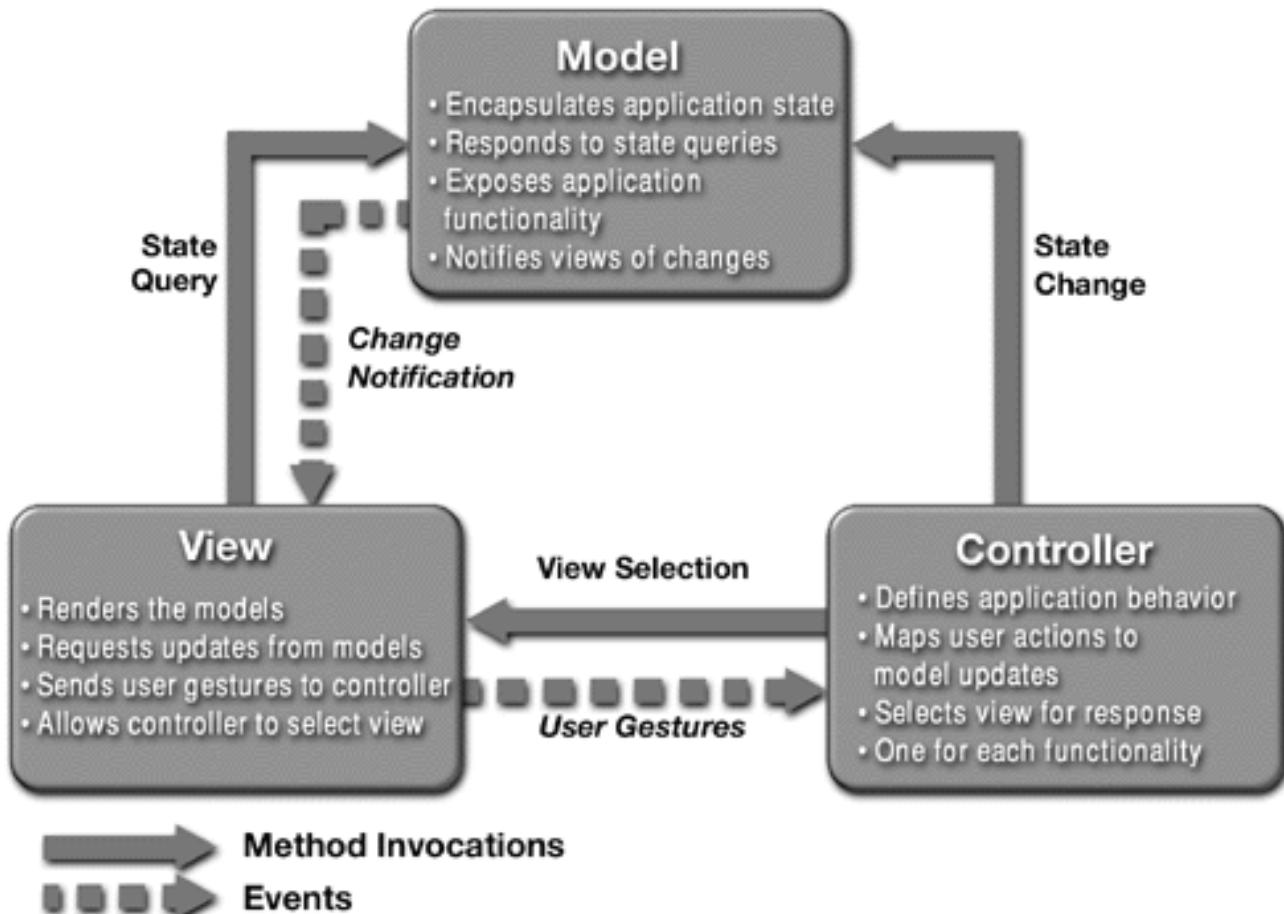
In applicazioni Web conformi a Model 2, richieste del browser cliente vengono passate a **controller** (usualmente implementato da **servlet** oppure **EJB Session Bean** oppure...)

- **Controller** si occupa di eseguire **logica business necessaria per ottenere il contenuto** da mostrare. Controller mette il contenuto (usualmente sotto forma di JavaBean o Plain Old Java Object - POJO) in messaggio e **decide a quale view** (usualmente implementata da **JSP**) passare la richiesta
- **View** si occupa del **rendering contenuto** (ad es. stampa dei valori contenuti in struttura dati o bean, ma anche operazioni più complesse come invocazione metodi per ottenere dati)

Architettura Model-View-Controller

Model-View-Controller (architettura generale)

- Eventi e invocazioni
- In Java Model 2, tipicamente controller come *EJB Session Bean o servlet*, e view come JSP



Aldilà di MVC, in generale, perché application server?

Vi ricordate discorsi su architetture multi-tier, vero?

- *Complessità del middle tier server*
- *Duplicazione dei servizi di sistema per la maggior parte delle applicazioni enterprise*
 - *Controllo concorrenza, transazioni*
 - *Load-balancing, sicurezza*
 - *Gestione risorse, connection pooling*
- *Come risolvere il problema?*
 - *Container condiviso che gestisce i servizi di sistema*
 - *Proprietario vs. basato su standard aperti?*

Soluzioni a Container: Proprietarie vs. Standard-based

Soluzioni proprietarie:

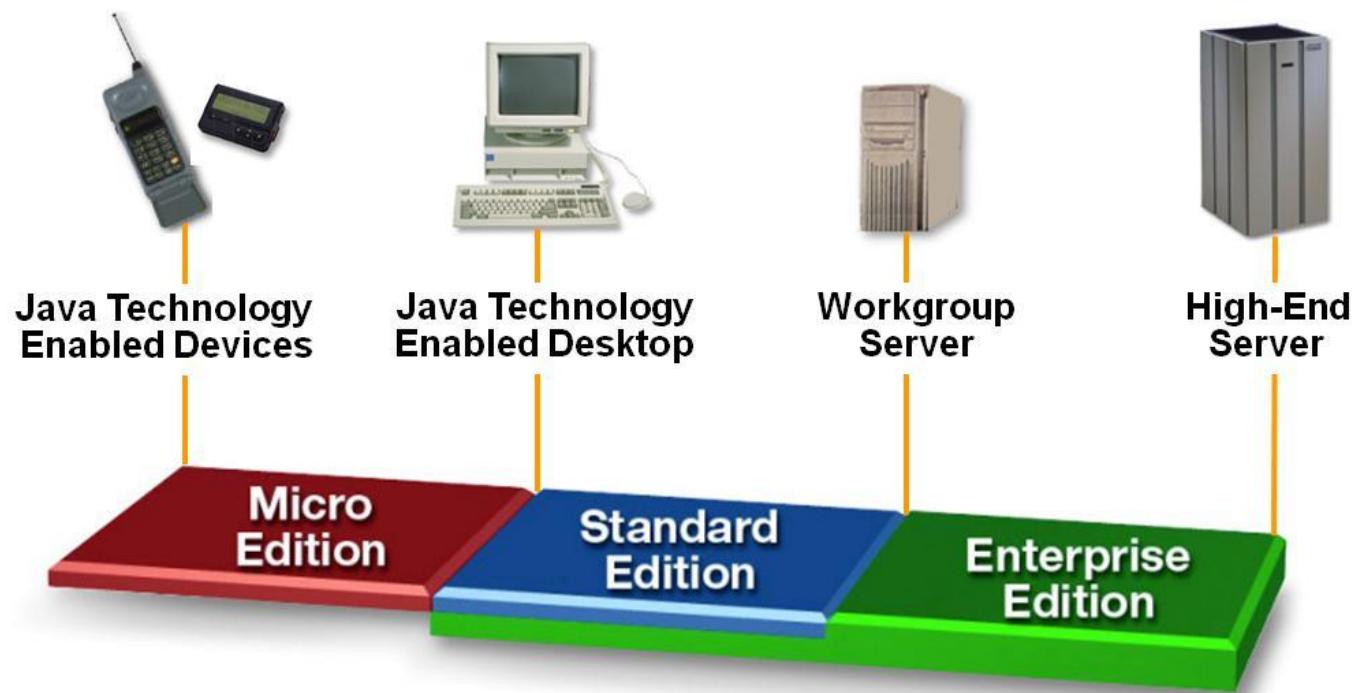
- Usano il modello componente-container
 - Componenti per la business logic
 - Container per fornire servizi di sistema
- Il contratto componenti-container è ben definito, ma in modo proprietario (problema di *vendor lock-in*)
 - Esempi: Tuxedo, .NET

Soluzioni basate su standard aperti

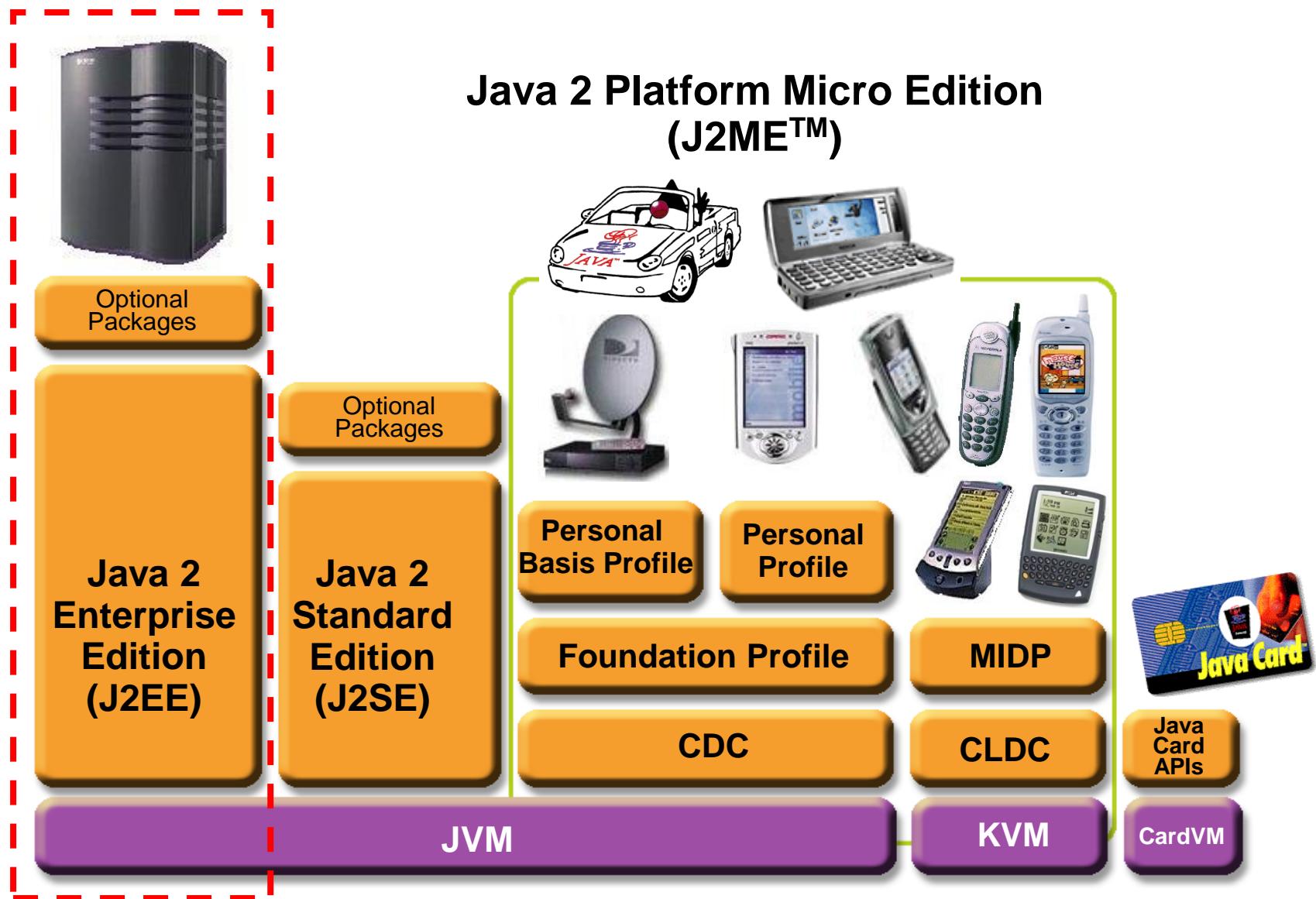
- Usano il modello componente-container e il container fornisce i servizi di sistema *in modo ben definito in accordo a standard industriali*
 - Ad es. J2EE e Java Specification Request (JSR)
(tra l'altro, anche supporto a portabilità di codice perché basato su bytecode Java e API di programmazione definite in standard aperti)

Che cos'è “application server” Java Enterprise Edition?

Piattaforma *open e standard* per lo sviluppo, il deployment e la gestione di applicazioni enterprise n-tier, Web-enabled, server-centric e basate su componenti



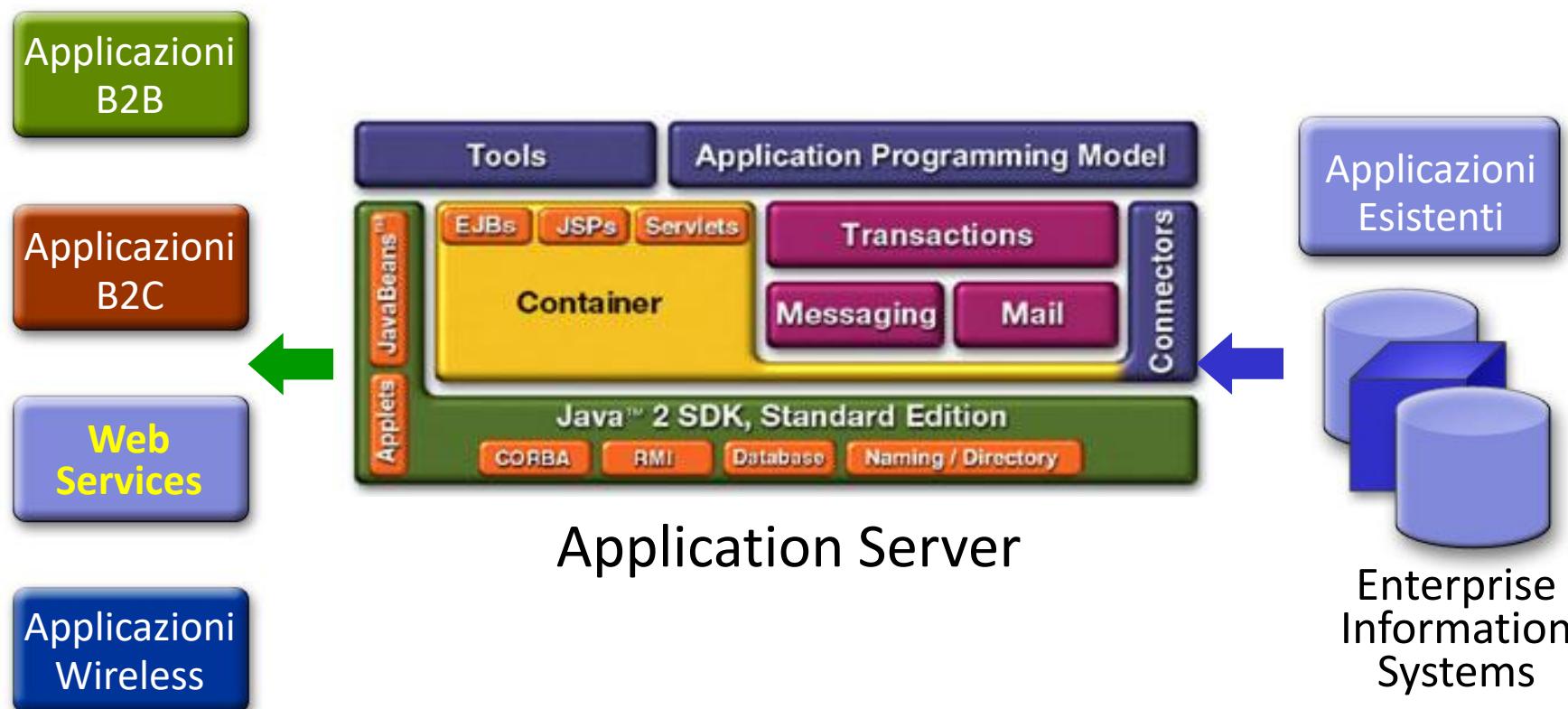
Varie “Edizioni” della Piattaforma Java



Architettura J2EE per Application Server

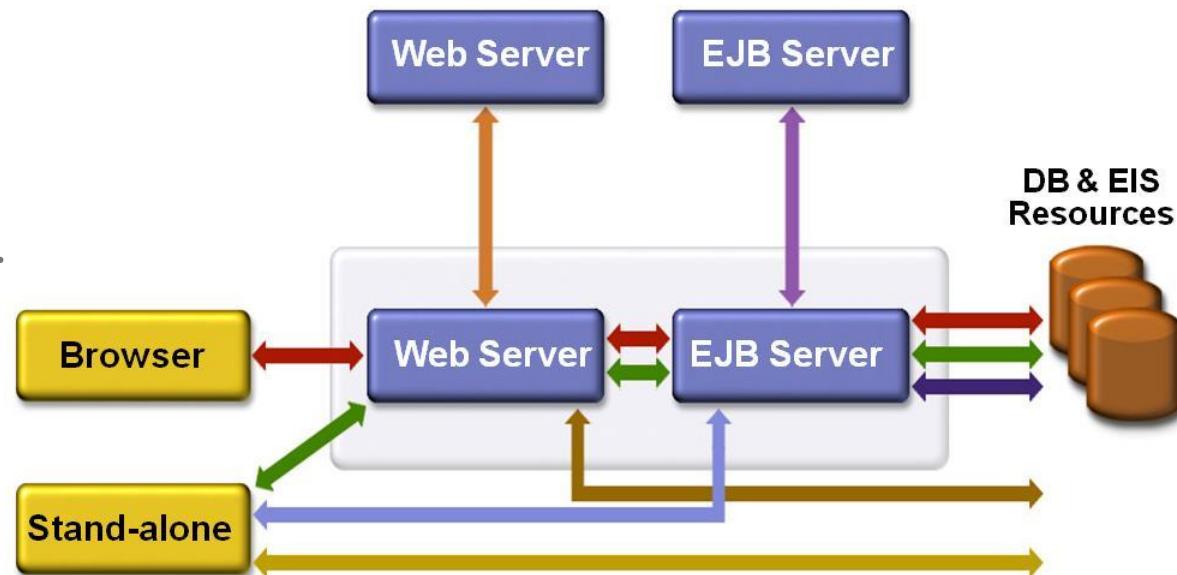
In questo corso non ci occuperemo dettagliatamente del container J2EE in termini generali (vedi corso di Sistemi Distribuiti M)

Ma per capire alcune direzioni di implementazione di Java Model 2, *di che cosa si occupa il container J2EE, e le differenze rispetto ad approcci a container leggero come Spring*, almeno dobbiamo avere parziale comprensione del modello...



J2EE per Applicazioni N-tier

- **Modello 4-tier e applicazioni J2EE**
 - Cliente HTML, JSP/Servlet, EJB, JDBC/Connector
- **Modello 3-tier e applicazioni J2EE**
 - Cliente HTML, JSP/Servlet, JDBC
- Modello 3-tier e applicazioni J2EE
 - Applicazioni standalone EJB client-side, EJB, JDBC/Connector
- Applicazioni enterprise B2B
 - Interazioni tra piattaforme J2EE tramite messaggi JMS o XML-based



Delega al Container

Il container può fornire “*automaticamente*” molte delle funzioni per supportare il servizio applicativo verso l’utente

- ***Supporto al ciclo di vita***

- Attivazione/deattivazione* del servitore

- Mantenimento dello *stato* (durata della sessione?)

- Persistenza trasparente* e recupero delle informazioni (interfaccia DB)

- ***Supporto al sistema dei nomi***

- Discovery* del servitore/servizio

- Federazione* con altri container

- ***Supporto alla qualità del servizio***

- Tolleranza ai *guasti*, selezione tra possibili *deployment*

- Controllo della *QoS richiesta e ottenuta*

- ***Sicurezza***

- ...

EJB in una slide...

- Tecnologia per **componenti server-side**
- Sviluppo e deployment semplificato di applicazioni Java:
 - *Distribuite, con supporto alle transazioni, multi-tier, portabili, scalabili, sicure, ...*
- Porta e amplifica i **benefici del modello a componenti** sul lato server
- Separazione fra **logica di business e codice di sistema**
 - **Container** per la fornitura dei servizi di sistema
- **Modello a container pesante** (contrapposto a possibili modelli alternativi, come container leggero Spring)
- Rende possibile (e semplice) la **configurazione a deployment-time**
 - **Deployment descriptor**

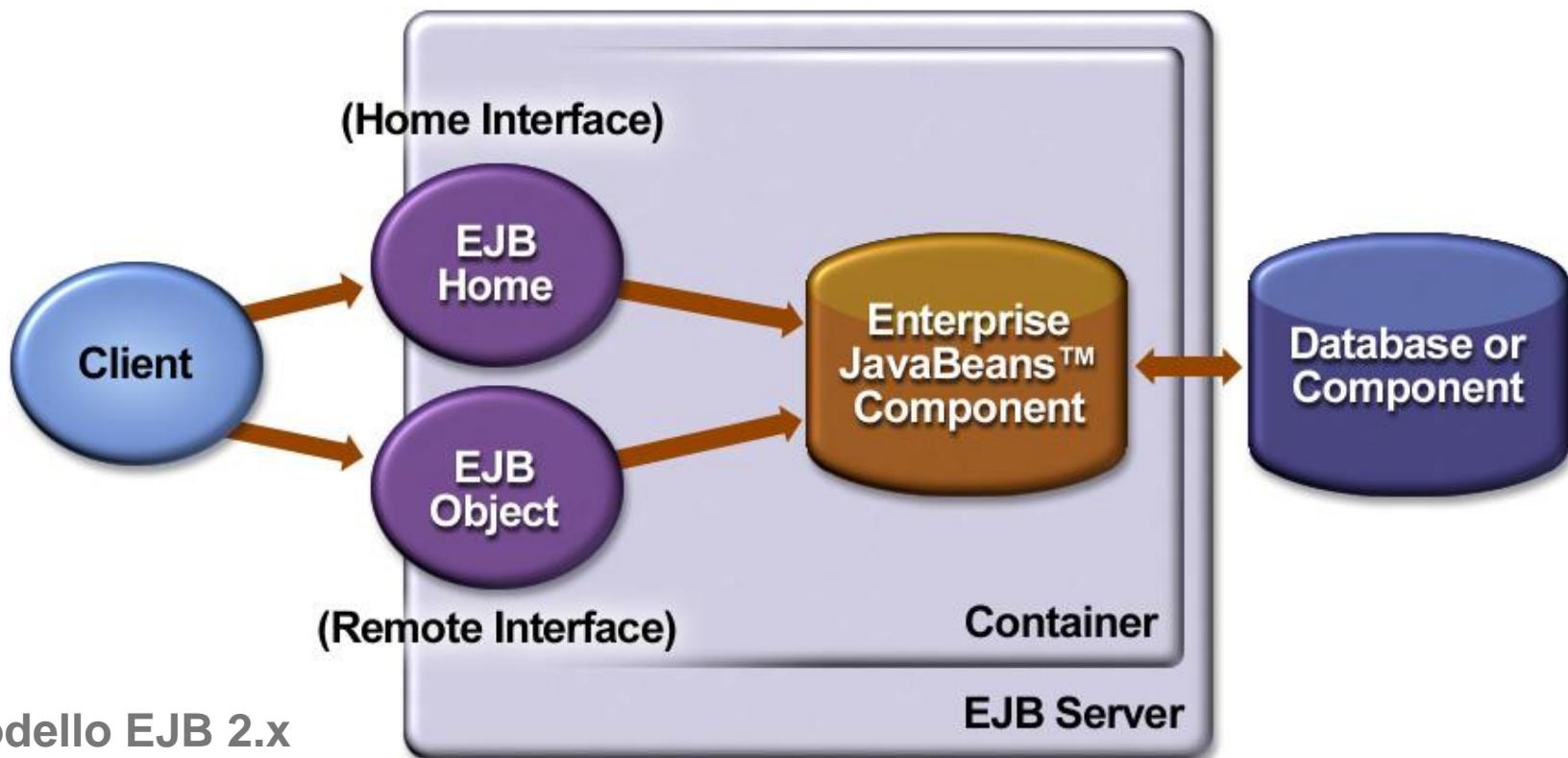
EJB: Principi di Design

- Applicazioni EJB e i loro componenti devono essere *debolmente accoppiati (loosely coupled)*
- Comportamento di EJB definito tramite *interfacce*
- Applicazioni EJB *NON* si occupano della *gestione delle risorse*
- *Applicazioni EJB sono N-tier*
 - *Session tier* come API verso l'applicazione
 - *Entity tier* come API verso le sorgenti dati

Architettura EJB

Idea di base: *container pesante attivo all'interno di un EJB Server (Application Server)*

Cliente può interagire *remotamente* con componente EJB tramite *interfacce ben definite passando SEMPRE attraverso container*

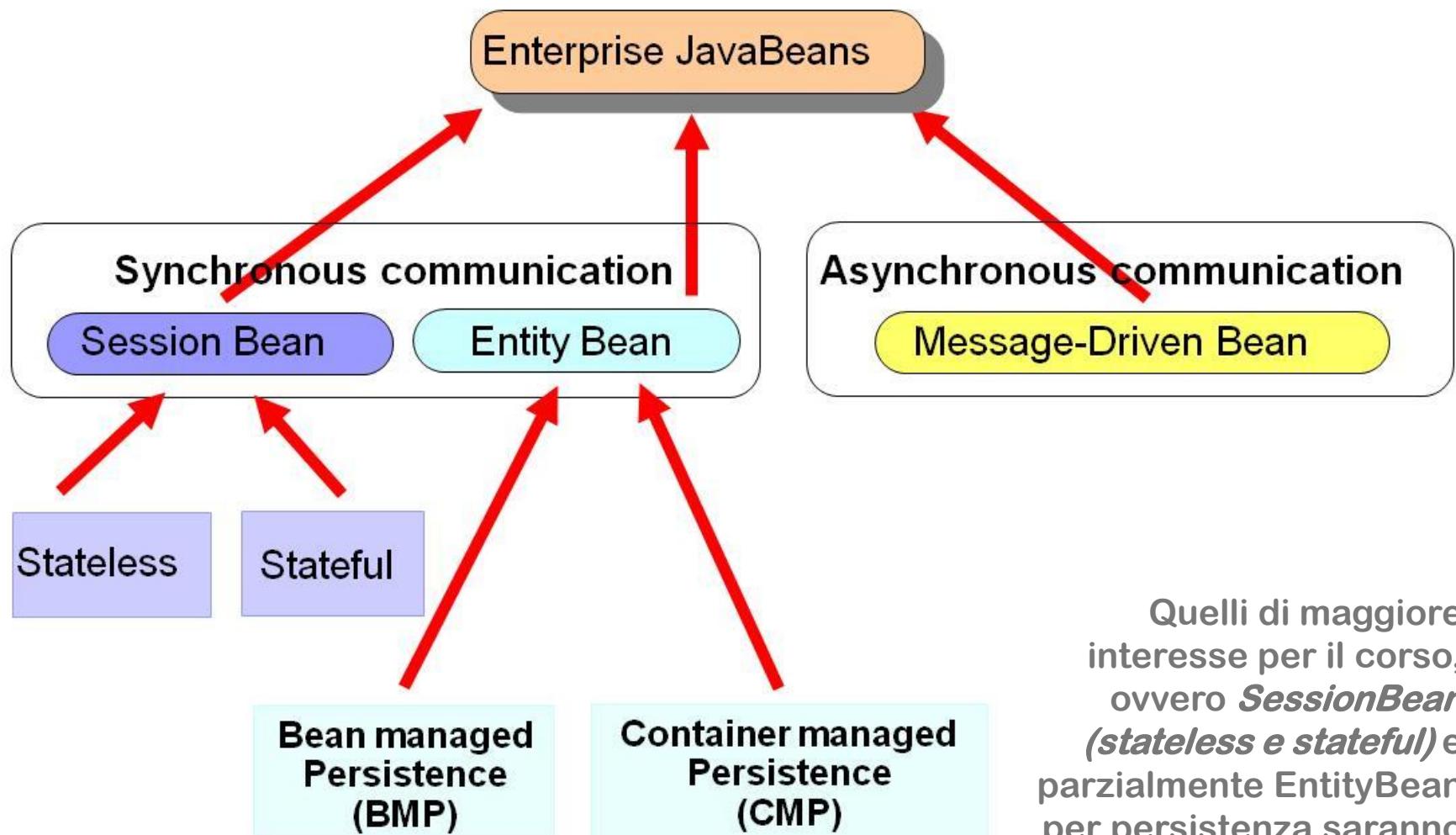


Modello EJB 2.x

Descrittori di Deployment

- Forniscono *istruzioni al container* su come gestire e controllare il comportamento (anche runtime) di componenti J2EE
 - Transazioni
 - Sicurezza
 - Persistenza
 - ...
- Permettono la *personalizzazione tramite specifica dichiarativa* (NO personalizzazione tramite programmazione)
- Semplificano *portabilità* del codice
- Sostituiti o sostituibili con *annotazioni* a partire da Java5

Principali Componenti EJB



Quelli di maggiore
interesse per il corso,
ovvero *SessionBean*
(*stateless* e *stateful*) e
parzialmente *EntityBean*
per persistenza saranno
descritti in seguito....

Tipologie di Componenti Bean

- ***Session Bean***
 - *Stateful session bean*
 - *Stateless session bean*
- ***Entity Bean***
 - Bean Managed Persistence (BMP)
 - Container Managed Persistence (CMP)
- ***Message Driven Bean***
 - Per Java Message Service (JMS)
 - Per Java API for XML Messaging (JAXM)

Session Bean

In questo corso ci concentreremo principalmente su Session Bean per realizzazione di controller

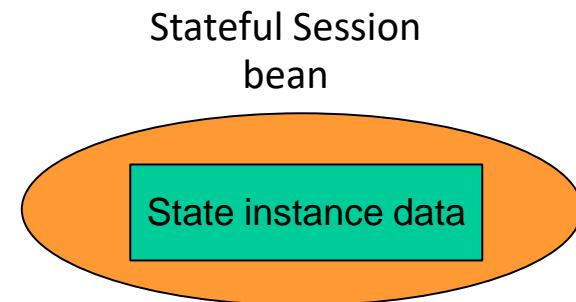
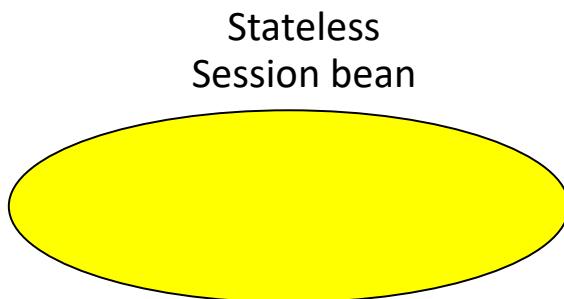
- *Lavorano tipicamente per un singolo cliente*
- *Non sono persistenti* (vita media relativamente breve)
 - Persi in caso di failure di EJB server
- *Non rappresentano dati in un DB*, anche se possono accedere/modificare questi dati
- In EJB2.x classe Bean deve implementare interfaccia `javax.ejb.SessionBean`; in EJB3.x solo uso di annotazioni

Quando usare i Session Bean?

- Per modellare oggetti di processo o di controllo *specifici per un particolare cliente*
- Per modellare workflow o attività di gestione e per coordinare interazioni fra bean
- Per muovere la *logica applicativa di business* dal lato cliente a quello servitore

Session Bean

- **Stateless:** esegue una richiesta e restituisce risultato *senza salvare alcuna informazione di stato relativa al cliente*
 - transienti
 - elemento temporaneo di business logic necessario per uno specifico cliente per un intervallo di tempo limitato
- **Stateful:** può mantenere *stato specifico per un cliente*



Come gestire il *pooling* di questi SB?
Fa differenza se si tratta di SB con stato o senza stato?

Entity Bean

- Forniscono una **vista ad oggetti** dei dati mantenuti in un database
 - Tempo di vita *non connesso alla durata delle interazioni con i clienti*
 - Componenti permangono nel sistema fino a che i dati esistono nel database - *long lived*
 - Nella maggior parte dei casi, **componenti sincronizzati con relativi database relazionali**
- **Accesso condiviso** per clienti differenti
- In EJB2.x classe Bean deve implementare interfaccia **javax.ejb.EntityBean**; in EJB3.x supporto alla persistenza simile a quanto vedrete per Hibernate

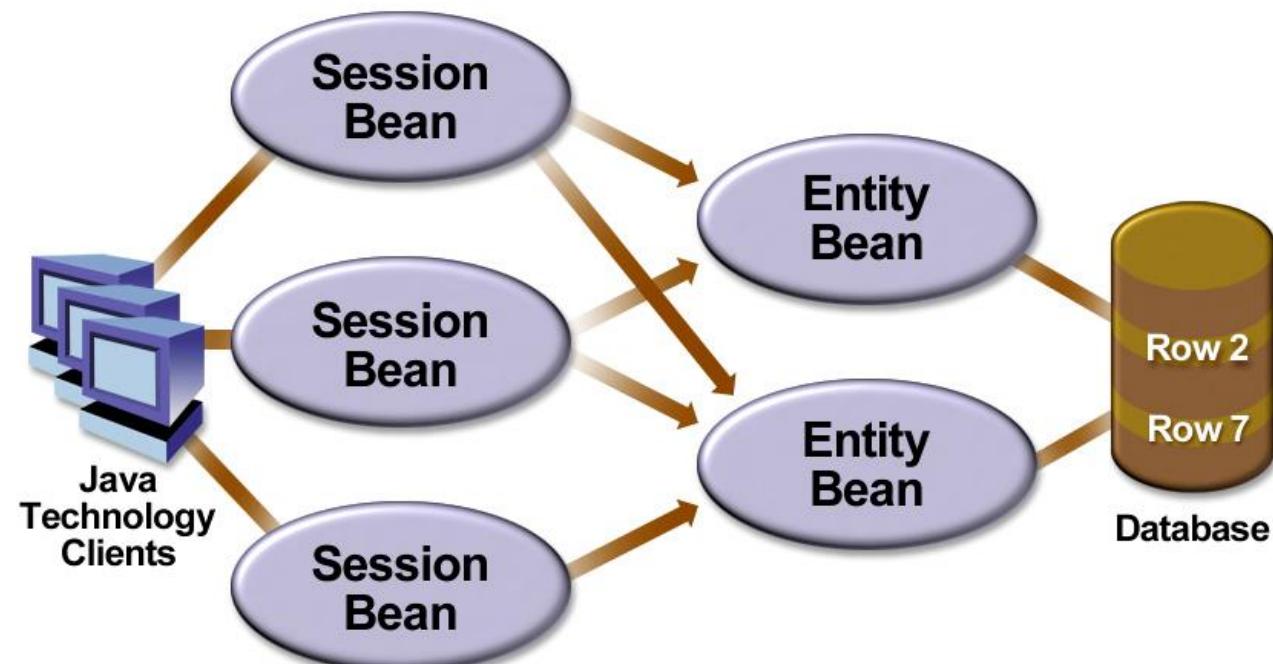
Session Bean

- Rappresenta un processo di business
- Una istanza per cliente
- **Short-lived:** vita del bean pari alla vita cliente
- Transient
- Non sopravvive a crash del server
- Può avere proprietà transazionali

Entity Bean

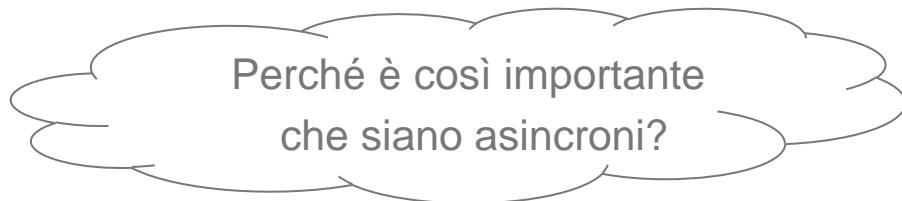
- Rappresenta dati di business
- Istanza condivisa fra clienti multipli
- **Long-lived:** vita del bean pari a quella dei dati nel database
- Persistente
- Sopravvive a crash del server
- Sempre transazionale

Session ed Entity Bean



Message-Driven Bean (MDB)

- Svolgono il ruolo di **consumatori di messaggi asincroni**
- **Non possono essere invocati direttamente dai clienti**
 - Attivati in seguito all'arrivo di un messaggio
- I clienti possono interagire con MDB tramite l'invio di messaggi verso le **code o i topic per i quali questi componenti sono in ascolto (listener)**
- **Privi di stato**



Perché è così importante che siano asincroni?

Nel caso di **utilizzo di JMS**

- MDB corrispondente deve implementare l'interfaccia javax.jms.MessageListener interface
- L'implementazione del metodo onMessage() deve contenere la business logic
- Il bean viene configurato come **listener per queue o topic JMS**

Tipologie di Bean EJB3.0

- In EJB 3.0, i bean di tipo sessione e message-driven sono classi Java ordinarie (Plain Old Java Object - POJO)
 - Rimossi i requisiti di interfaccia
 - Tipo di bean specificato da una annotation (o da un descrittore)
 - Annotation principali
 - @Stateless, @Stateful, @MessageDriven
 - Specificati nella classe del bean
- Gli entity bean di EJB2.x non sono stati modificati e possono continuare a essere utilizzati
 - Ma Java Persistence API supporta nuove funzionalità
 - @Entity si applica solo alle nuove entità relative a Java Persistence API

Esempio

```
// EJB2.1 Stateless Session Bean: Bean Class

public class PayrollBean implements javax.ejb.SessionBean {
    SessionContext ctx;
    public void setSessionContext(SessionContext ctx) {
        this.ctx = ctx;
    }
    public void ejbCreate() {...}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}

    public void setTaxDeductions(int empId, int deductions) {
        ...
    }
}
```

```
// EJB3.0 Stateless Session Bean: Bean Class

@Stateless
public class PayrollBean implements Payroll {

    public void setTaxDeductions(int empId,int deductions) {
        ...
    }
}
```

Esempio

Accesso locale o remoto

- *Locale a default*
- *Remoto tramite annotation o descrittore di deployment*

```
// EJB 3.0 Stateless Session Bean: Interfaccia remota

@Remote
public interface Payroll {

    public void setTaxDeductions(int empId, int deductions);

}
```

```
// EJB 3.0 Stateless Session Bean:
// Alternativa: @Remote annotation applicata direttamente alla
// classe bean

@Stateless @Remote
public class PayrollBean implements Payroll {

    public void setTaxDeductions(int empId,int deductions) {
        ...
    }
}
```

Oltre ai dettagli di programmazione, ancora più rilevante è capire *quali servizi di supporto/sistema e come vengano supportati in un modello a container pesante:*

- **Pooling e concorrenza** 
- **Transazionalità**
- **Gestione delle connessioni a risorse**
- **Persistenza** (vedi Java Persistence API – JPA - e supporto Hibernate ORM)
- **Messaggistica** (vedi Java Messaging System – JMS)
- **(Sicurezza)**

1) Gestione della Concorrenza

Migliaia fino a milioni di oggetti in uso simultaneo

Come gestire relazione fra numero clienti e numero oggetti distribuiti richiesti per servire richieste cliente?

Discussion: voi come realizzereste *instance pooling*?

- **Resource Pooling**
 - Pooling dei componenti server-side da parte di EJB container (*instance pooling*). Idea base è di *evitare di mantenere una istanza separata di ogni EJB per ogni cliente. Si applica a stateless session bean e message-driven bean* (che cos'è un cliente per un MDB?)
 - Anche *pooling dei connector* (vedremo in seguito)
- **Activation**
 - *Utilizzata da stateful session bean per risparmiare risorse*

Stateless Session Bean

Ogni EJB container mantiene ***un insieme di istanze del bean*** pronte per servire richieste cliente

Non esiste stato di sessione da mantenere fra richieste successive; ***ogni invocazione di metodo è indipendente*** dalle precedenti

Implementazione delle strategie di instance pooling demandate ai vendor di EJB container, ma analoghi principi

Ciclo di vita di uno stateless session bean:

- ***No state*** (non istanziato; stato iniziale e terminale del ciclo di vita)
 - ***Pooled state*** (istanziato ma non ancora associato ad alcuna richiesta cliente)
 - ***Ready state*** (già associato con una richiesta EJB e pronto a rispondere ad una invocazione di metodo)
-

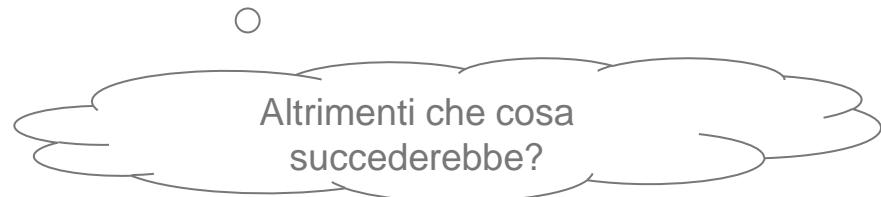
Stateless Session Bean

Istanza del bean nel pool riceve un riferimento a **javax.ejb.EJBContext** (in caso di richiesta di injection nel codice tramite apposita annotation)

EJBContext fornisce un'interfaccia per il bean per comunicare con l'ambiente EJB

Quando il bean passa in *stato ready*, *EJBContext* contiene anche informazioni sul *cliente che sta utilizzando il bean*. Inoltre contiene il riferimento al proprio *EJB stub*, utile per passare riferimenti ad altri bean

Ricordiamo che il fatto di essere *stateless* è indicato semplicemente tramite annotation **@javax.ejb.Stateless** (NOTA: *variabili di istanza non possono essere usate per mantenere stato della sessione*)



Stateful Session Bean: Activation

Usata nel caso di *stateful session bean*

Gestione della coppia oggetto EJB + istanza di bean stateful

- **Passivation:** *disassociazione fra stateful bean instance e suo oggetto EJB*, con salvataggio dell'istanza su memoria (*serializzazione*). Processo del tutto trasparente per cliente
- **Activation:** recupero dalla memoria (*deserializzazione*) dello stato dell'istanza e riassociazione con oggetto EJB

Nella specifica J2EE, non richiesto che la classe di uno stateful session bean sia *serializzabile*. Quindi?

Dipendenza dall'implementazione dello specifico vendor e attenzione al trattamento dei transient...

Stateful Session Bean: Activation

La procedura di activation può essere associata anche
all'invocazione di metodi di callback sui *cambi di stato nel ciclo di vita* di uno *stateful session bean*

Ad esempio, l'annotation **@javax.ejb.PostActivate** associa
l'invocazione del metodo a cui si applica *immediatamente dopo l'attivazione* di un'istanza

Similmente, **@javax.ejb.PrePassivate** (prima dell'azione di passivation)

Ad esempio, utilizzati spesso per la *chiusura/apertura di connessioni a risorse per gestione più efficiente*

Gestione Concorrenza

Per definizione, **session bean non possono essere concorrenti**, nel senso che **una singola istanza è associata ad un singolo cliente**

Vietato l'utilizzo di thread a livello applicativo e, ad esempio, della keyword synchronized

Come nel caso di stateless session bean, **Message Driven Bean non mantengono stato della sessione** e quindi il container può effettuare pooling in modo relativamente semplice

Strategie di pooling analoghe alle precedenti. Unica differenza che **ogni EJB container contiene molti pool, ciascuno dei quali è composto di istanze con la stessa destination JMS**

Un semplice esempio per Session Bean...

Immaginiamo il solito sito di e-commerce, con homepage a JSP...

Control/Business Logic tramite Session Bean

- **OperationsBean:** *Stateless SB* contenente la definizione di somma, divisione...
- **CalculatorBean:** *Stateful SB* che
 1. seleziona operazione da svolgere in base a parametri forniti
 2. effettua operazione richiesta (non direttamente ma demandando ad altro componente)
 3. mantiene il risultato parziale delle operazioni



Spring

Introduzione a Spring

Che cos'è Spring?

- Implementazione di *modello a container leggero* per la costruzione di applicazioni Java SE e Java EE

Molti dei concetti chiave alla base di Spring sono stati di successo così rilevante da essere diventati linee guida per l'evoluzione di EJB3.0

Funzionalità chiave:

- *Inversion of Control (IoC) e Dependency injection*
- Supporto alla persistenza
- Integrazione con Web tier
- Aspect Oriented Programming (AOP)

Funzionalità Chiave: Dependency Injection (e Persistenza)

Dependency Injection

- Gestione della configurazione dei componenti applica principi di *Inversion-of-Control* e utilizza *Dependency Injection*
 - Eliminazione della necessità di binding “manuale” fra componenti
- *Idea fondamentale di una factory per componenti (BeanFactory) utilizzabile globalmente. Si occupa fondamentalmente del ritrovamento di oggetti per nome e della gestione delle relazioni fra oggetti (configuration management)*

Persistenza

- *Livello di astrazione generico* per la gestione delle transazioni con DB (senza essere forzati a lavorare dentro un EJB container)
- Integrazione con framework di persistenza come Hibernate, JDO, JPA
Lo vedrete nel dettaglio nelle prossime lezioni del corso...

Funzionalità Chiave: Web tier e AOP

Integrazione con Web tier

- *Framework MVC per applicazioni Web*, costruito sulle funzionalità base di Spring, con supporto per diverse tecnologie per la *generazione di viste*, ad es. JSP, FreeMarker, Velocity, Tiles, iText e POI (Java API per l'accesso a file in formato MS)
- *Web Flow* per navigazione a grana fine

Supporto a Aspect Oriented Programming

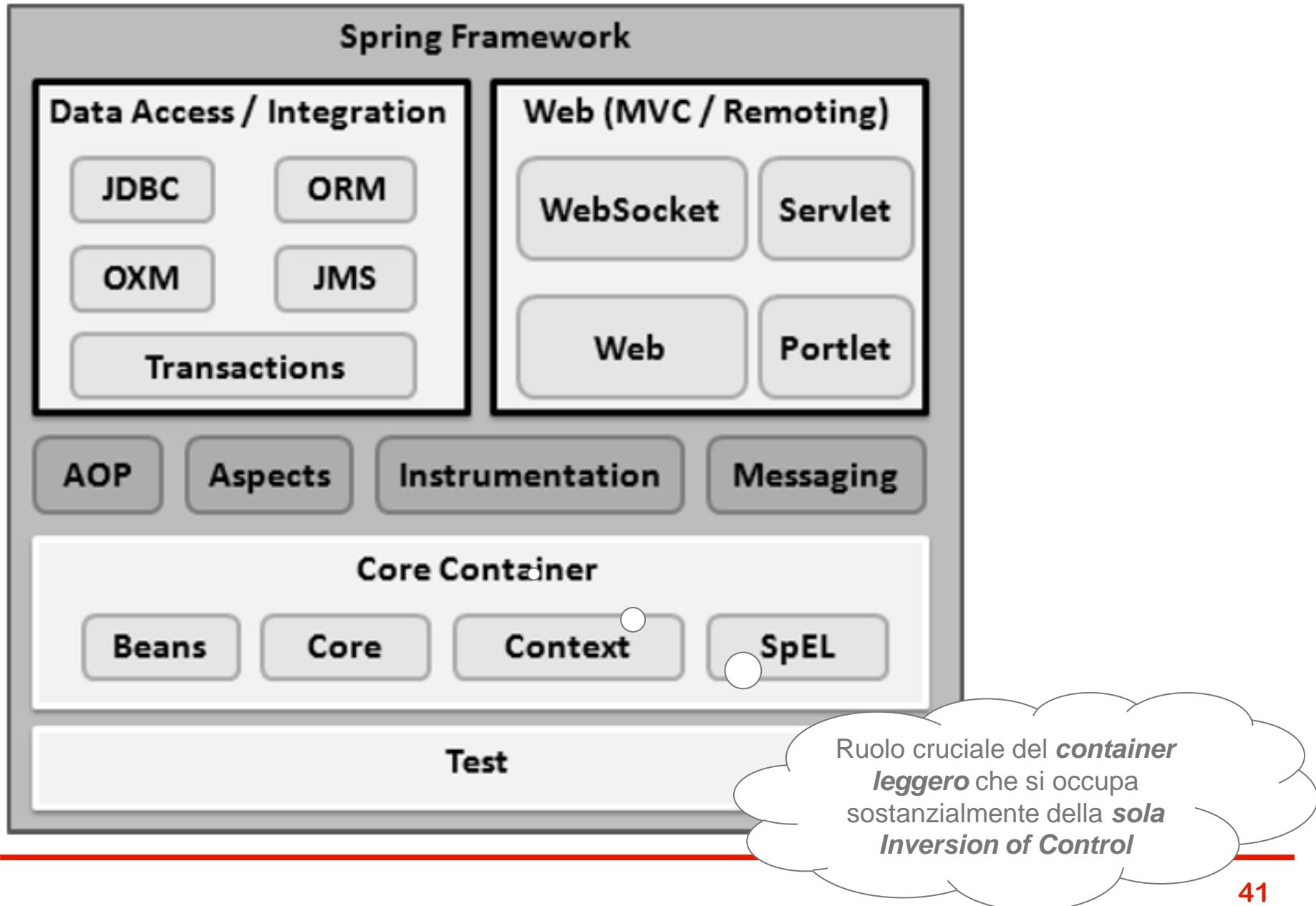
- Framework di supporto a servizi di sistema, come gestione delle transazioni, tramite *tecniche AOP*
 - Miglioramento soprattutto in termini di *modularità*
 - Parzialmente correlata anche la facilità di testing

Yet Another Container/Framework?

No, Spring rappresenta un approccio piuttosto unico (che ha fortemente influenzato i container successivi, *verso tecnologie a microcontainer* – Spring 1.2 è datato Maggio 2005). In altre parole, **proprietà originali**:

- **Spring come framework modulare. Architettura a layer, possibilità di utilizzare anche solo alcune parti in isolamento**
 - Anche possibilità di introdurre Spring *incrementalmente in progetti esistenti* e di imparare ad utilizzare la tecnologia “pezzo per pezzo”
- **Supporto a importanti aree non coperte da altri framework diffusi, come la gestione degli oggetti di business**
- **Tecnologia di integrazione di soluzioni esistenti**
- **Facilità di testing**

Architettura di Spring



Architettura di Spring – i soli moduli di interesse centrale per il corso

Core Package

- ❑ Parte fondamentale del framework. Consiste in un *container leggero* che si occupa di *Inversion of Control* o, per dirla alla Fowler, *Dependency Injection*
- ❑ L'elemento fondamentale è *BeanFactory*, che fornisce una implementazione estesa del pattern factory ed *elimina la necessità di gestione di singleton* a livello di programmazione, permettendo di disaccoppiare configurazione e dipendenze dalla logica applicativa

Vi ricordate bene, vero,
che cos'è un singleton?

MVC Package

- ❑ Ampio supporto a progettazione e sviluppo secondo architettura *Model-View-Controller (MVC)* per applicazioni Web

Vedremo solo qualche dettaglio in seguito (per ulteriori dettagli, vedi esercitazione opzionale da svolgersi in autonomia su sito Web del corso)

Dependency Injection in Spring

- *Applicazione più nota e di maggiore successo del principio di Inversion of Control*
 - “Hollywood Principle”
 - Don't call me, I'll call you
- Container (in realtà il container leggero di Spring) si occupa di *risolvere (injection)* le dipendenze dei componenti attraverso l'opportuna *configurazione dell'implementazione dell'oggetto (push)*
- Opposta ai pattern più classici di *istanziazione di componenti* o *Service Locator*, dove è il componente che deve determinare l'implementazione della risorsa desiderata (*pull*)
 - Martin Fowler chiamò per primo *Dependency Injection* questo tipo di IoC

Potenziali Benefici di Dependency Injection

Stessi benefici sono propri di molte altre tecnologie correnti, tutte con dependency injection, come EJB3.0

Flessibilità

- Eliminazione di codice di lookup nella logica di business

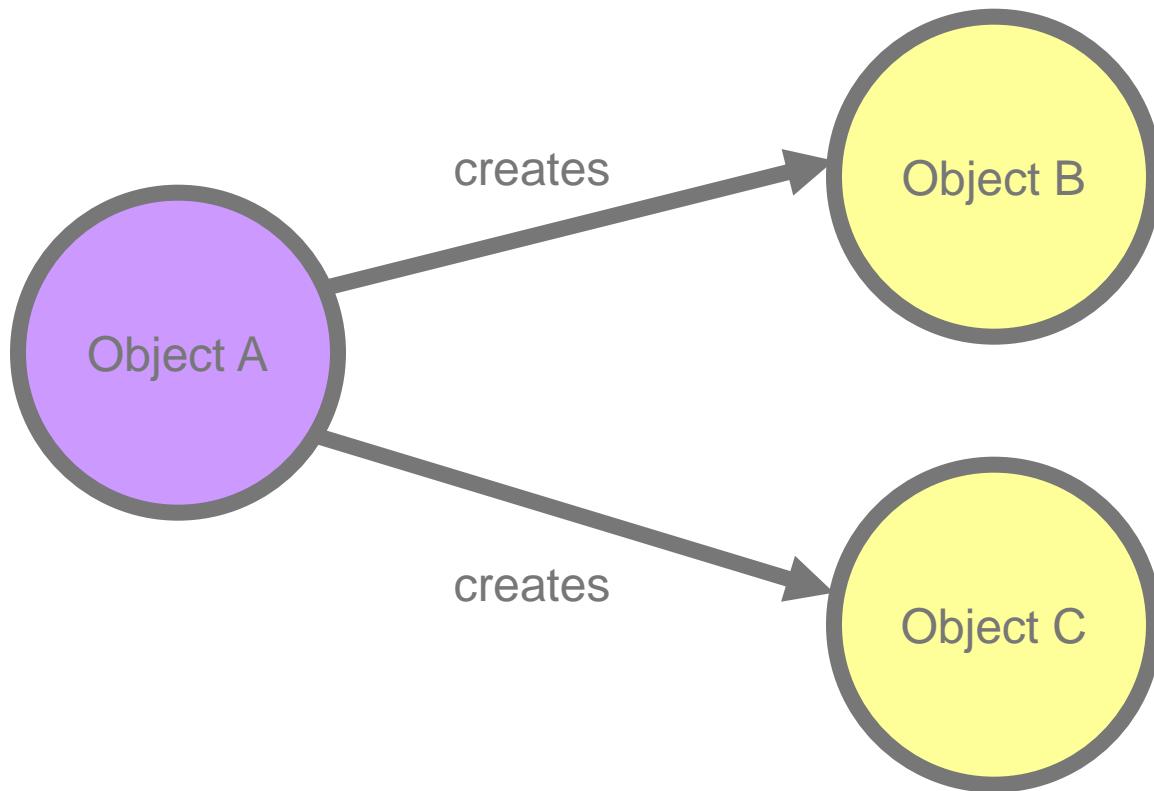
Possibilità e facilità di testing

- Nessun bisogno di *dipendere da risorse esterne o da container in fase di testing*
- Possibilità di abilitare *testing automatico*

Manutenibilità

- Permette *riutilizzo in diversi ambienti applicativi* cambiando semplicemente i file di configurazione (o in generale le specifiche di dependency injection) e *non il codice*

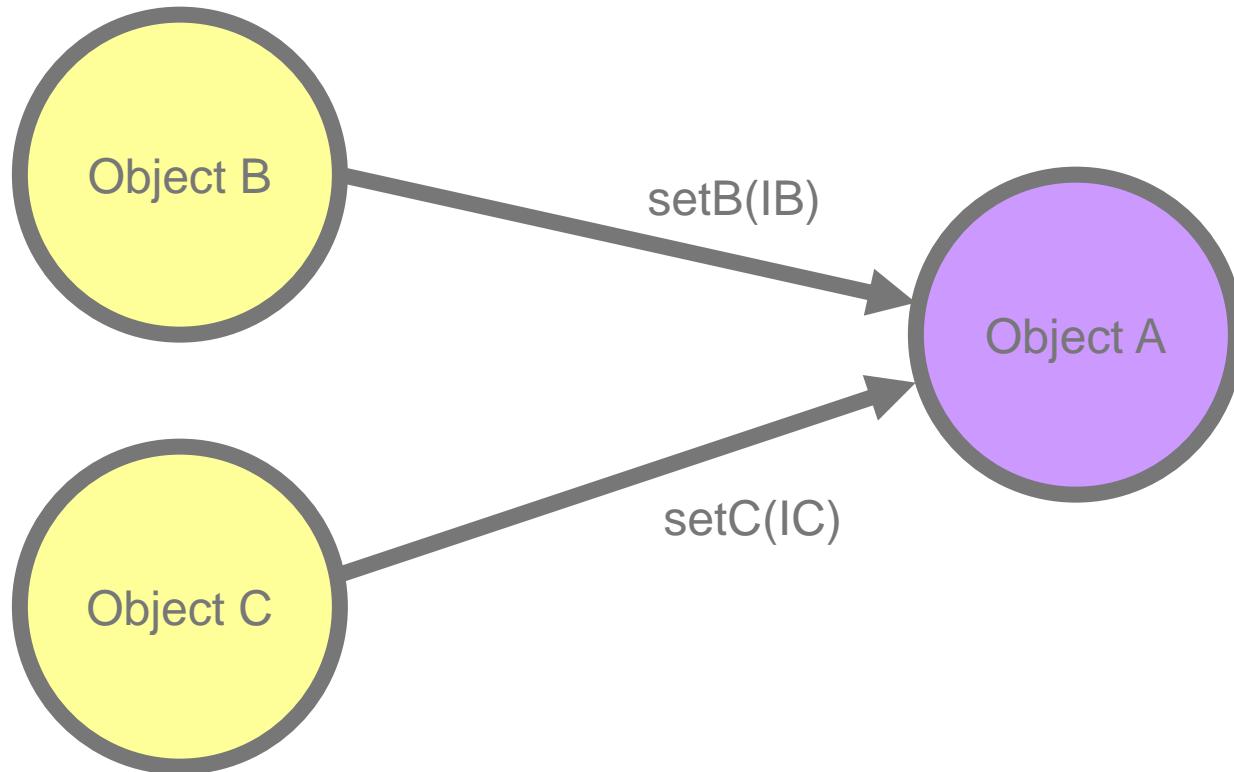
Senza Dependency Injection



Un oggetto/componente deve esplicitamente istanziare gli oggetti/componenti di cui ha necessità (sue dipendenze)

Accoppiamento stretto tra oggetti/componenti

Con Supporto a Dependency Injection



Supporto a dependency injection si occupa di creare oggetti/componenti quando necessario e di passarli automaticamente agli oggetti/componenti che li devono utilizzare

Idea base per implementazione: costruttori in oggetto A che accettano B e C come parametri di ingresso, oppure A contiene metodi setter che accettano interfacce B e C come parametri di ingresso

Due Varianti per Dependency Injection in Spring

□ Dependency injection *a livello di costruttore*

Dipendenze fornite attraverso i costruttori dei componenti

```
public class ConstructorInjection {  
    private Dependency dep;  
    public ConstructorInjection(Dependency dep) {  
        this.dep = dep;    } }
```

□ Dependency injection *a livello di metodi “setter”*

- Dipendenze fornite attraverso i *metodi di configurazione* (*metodi setter in stile JavaBean*) dei componenti
- Più frequentemente utilizzata nella comunità degli sviluppatori

```
public class SetterInjection {  
    private Dependency dep;  
    public void setMyDependency(Dependency dep) {  
        this.dep = dep;    } }
```

BeanFactory

- L'oggetto **BeanFactory** è responsabile della *gestione dei bean che usano Spring e delle loro dipendenze*
- Ogni applicazione interagisce con la dependency injection di Spring (*IoC container*) tramite l'interfaccia **BeanFactory**
 - Oggetto **BeanFactory** viene creato dall'applicazione, tipicamente nella forma di **XmlBeanFactory**
 - Una volta creato, l'oggetto **BeanFactory** legge un *file di configurazione e si occupa di fare l'injection (wiring)*
- **XmlBeanFactory** è estensione di **DefaultListableBeanFactory** per leggere definizioni di bean da un documento XML

```
public class XmlConfigWithBeanFactory {  
  
    public static void main(String[] args) {  
        XmlBeanFactory factory = new XmlBeanFactory(new  
            FileSystemResource("beans.xml"));  
        SomeBeanInterface b = (SomeBeanInterface) factory.  
            getBean("nameOftheBean"); } }
```

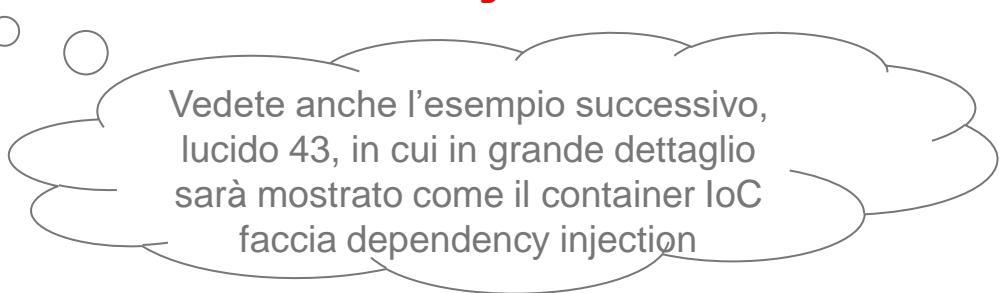
File di Configurazione

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
"http://www.springframework.org/dtd/spring-beans.dtd">  
<beans>  
    <bean id="renderer" class="StandardOutMessageRenderer">  
        <property name="messageProvider">  
            <ref local="provider"/>  
        </property>  
    </bean>  
    <bean id="provider" class="HelloWorldMessageProvider"/>  
</beans>
```

Oppure a livello di costruttore

...

```
<beans>  
    <bean id="provider" class="ConfigurableMessageProvider">  
        <constructor-arg>  
            <value> Questo è il messaggio configurabile</value>  
        </constructor-arg>  
    </bean> </beans>
```

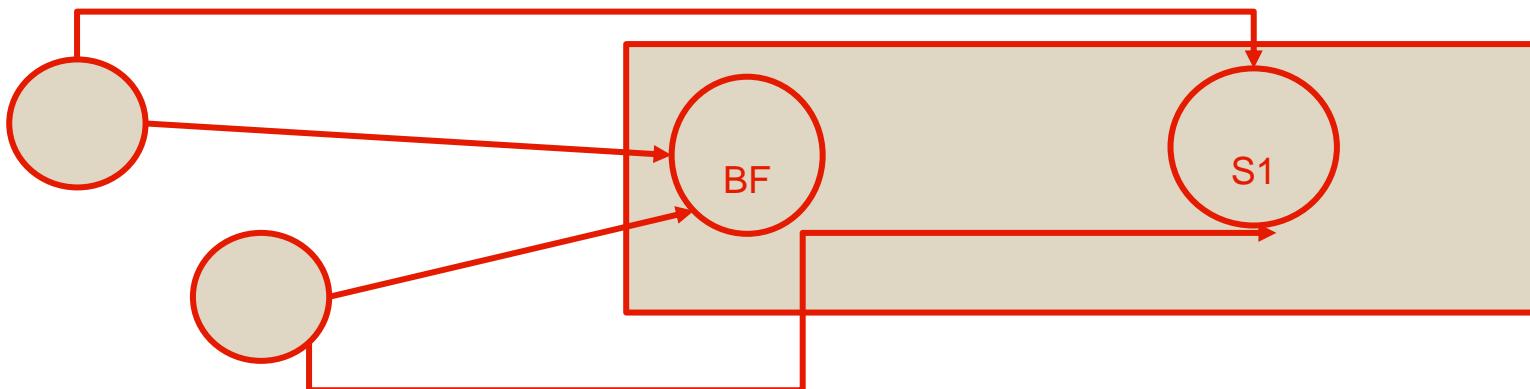
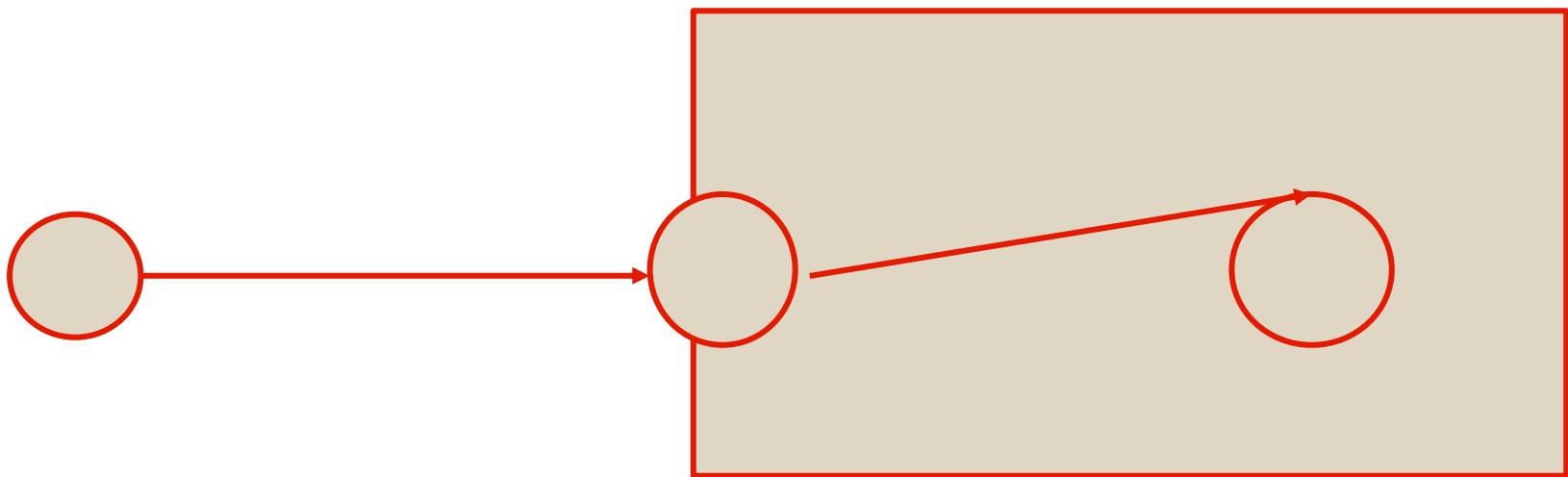


Vedete anche l'esempio successivo,
lucido 43, in cui in grande dettaglio
sarà mostrato come il container IoC
faccia dependency injection

Uso della Dependency Injection

```
public class ConfigurableMessageProvider implements  
    MessageProvider {  
  
    private String message;  
    // usa dependency injection per config. del messaggio  
    public ConfigurableMessageProvider(String message) {  
        this.message = message;  
    }  
  
    public String getMessage() {  
        return message;  
    }  
}
```

Container pesante vs. leggero



Tipi di Parametri di Injection

- Spring supporta *diversi tipi di parametri* con cui fare injection
 - 1. *Valori semplici*
 - 2. *Bean all'interno della stessa factory*
 - 3. *Bean anche in diverse factory*
 - 4. *Collezioni (collection)*
 - 5. *Proprietà definite esternamente*
- Tutti questi tipi possono essere usati sia per *injection sui costruttori che sui metodi setter*

Ad esempio, injection di valori semplici

```
<beans>
    <bean id="injectSimple" class="InjectSimple">
        <property name="name"> <value>John Smith</value></property>
        <property name="age"> <value>35</value> </property>
        <property name="height"> <value>1.78</value> </property>
    </bean>
</beans>
```

Esempio: Injection di Bean della stessa Factory

- Usata quando è necessario fare *injection di un bean all'interno di un altro (target bean)*
- Uso del tag <ref> in <property> o <constructor-arg> del target bean
- *Controllo lasco sul tipo del bean “iniettato” rispetto a quanto definito nel target*
 - Se il tipo definito nel target è *un’interfaccia*, il bean injected deve essere *un’implementazione di tale interfaccia*
 - Se il tipo definito nel target è una *classe*, il bean injected deve essere della *stessa classe o di una sottoclasse*

```
<beans>
    <bean id="injectRef" class="InjectRef">
        <property name="oracle">
            <ref local="oracle"/>
        </property>
    </bean>
</beans>
```

Naming dei Componenti Spring

Come fa BeanFactory a trovare il bean richiesto (*pattern singleton come comportamento di default*)?

- Ogni bean deve avere un *nome unico all'interno della BeanFactory contenente*
- Procedura di risoluzione dei nomi
 - Se un tag <bean> ha un attributo di nome **id**, il valore di questo attributo viene usato come nome
 - Se non c'è attributo **id**, Spring cerca un attributo di nome **name**
 - Se non è definito né **id** né **name**, Spring usa il nome della classe del bean come suo nome

Supporto a MVC in Spring

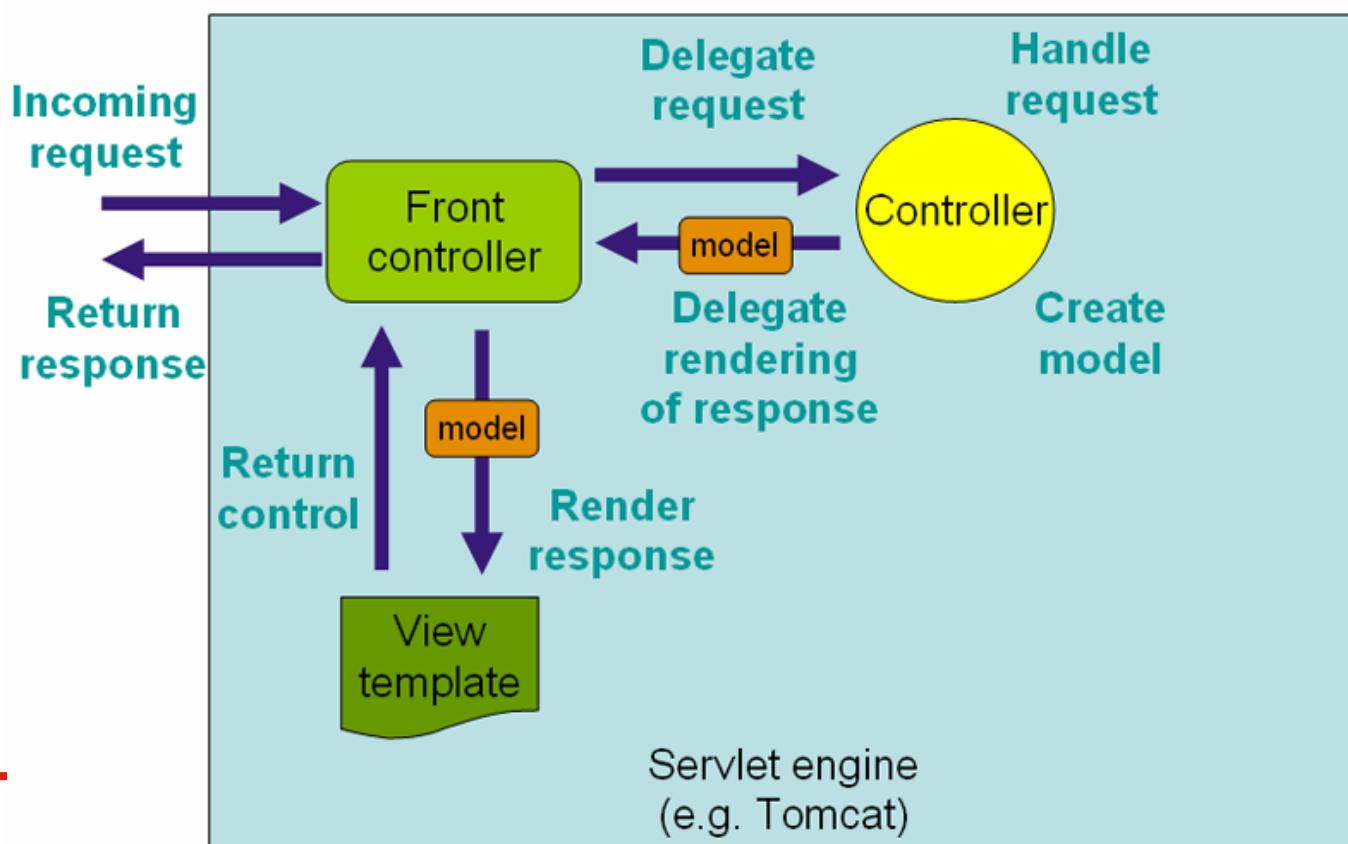
Supporto a *componenti “controller”*, responsabili per interpretare richiesta utente e interagire con business object applicazione

Una volta che il controller ha ottenuto i *risultati (parte “model”)*, decide a quale *“view” fare forwarding* del model; view *utilizza i dati in model* per creare presentazione verso l’utente

- *Chiara separazione ruoli*: controller, validator, *oggetti command/form/model*, *DispatcherServlet*, handler mapping, *view resolver*, ...
- *Adattabilità e riutilizzabilità*: possibilità di utilizzare qualsiasi classe per controller purché implementi interfaccia predefinita
- *Flessibilità nel trasferimento model*: via un Map nome/valore, quindi possibilità di integrazione con svariate tecnologie per view
- *Facilità di configurazione*: grazie al meccanismo standard di *dependency injection* di Spring
- *Handler mapping e view resolution configurabili*
- *Potente libreria di JSP tag* (Spring tag library): supporto a varie possibilità di temi (*theme*)
- Componenti con ciclo di vita scoped e associati automaticamente a HTTPRequest o HttpSession (grazie a WebApplicationContext di Spring)

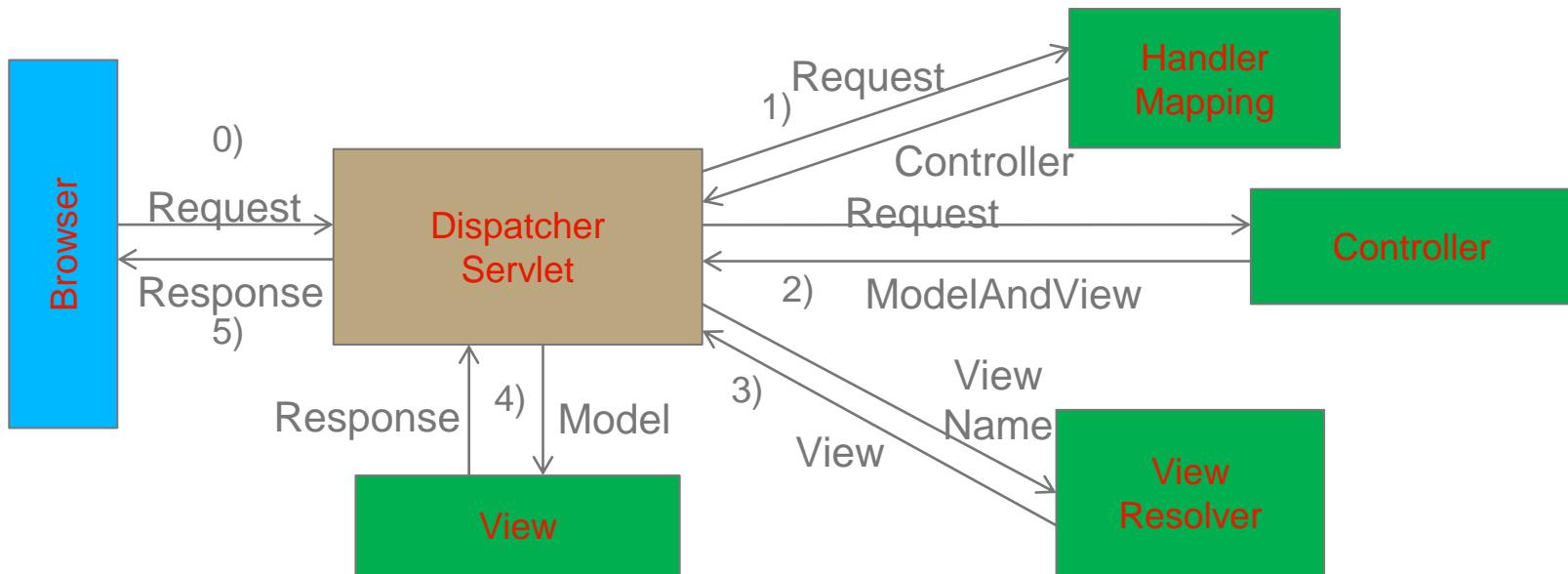
Spring DispatcherServlet

- Progettato attorno a una *servlet centrale che fa da dispatcher* delle richieste (*DispatcherServlet*)
- *Completamente integrata con IoC container* di Spring
- DispatcherServlet come “Front Controller”
- DispatcherServlet è una normale servlet e quindi serve URL mapping tramite web.xml



DispatcherServlet

- Intercetta le HTTP Request in ingresso che giungono al Web container
- Cerca un controller che sappia gestire la richiesta
- Invoca il controller ricevendo un model (output business logic) e una view (come visualizzare output)
- Cerca un View Resolver opportuno tramite cui scegliere View e creare HTTP Response



Spring DispatcherServlet

Azioni svolte da DispatcherServlet:

- WebApplicationContext è associato alla richiesta (controller e altri componenti potranno utilizzarlo)
`DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE`
- “*locale*” resolver associato alla richiesta (può servire nel processamento richiesta e view rendering)
- *theme resolver* associato alla richiesta (view potranno determinare quale theme usare)
- Viene cercato un *handler appropriato*. Se trovato, viene configurata una *catena di esecuzione associata all'handler (preprocessori, postprocessori e controller); risultato è preparazione di model*
- Se restituito un model, viene girato alla view associata (perché un model potrebbe non essere ricevuto?)

Spring Controller

Spring supporta la nozione di controller in modo astratto permettendo *creazione di ampia varietà di controller*: form-specific controller, command-based controller, controller che eseguono come wizard, ...

Base di partenza: interfaccia

```
org.springframework.web.servlet.mvc.Controller  
public interface Controller {  
    ModelAndView handleRequest( HttpServletRequest request,  
                               HttpServletResponse response) throws Exception; }
```

Molte implementazioni dell'interfaccia già disponibili. Ad es:

□ ***AbstractController***

classe che offre già supporto per caching. Quando la si utilizza come baseclass, necessita di fare overriding del metodo
handleRequestInternal(HttpServletRequest, HttpServletResponse)

```
public class SampleController extends AbstractController {  
    public ModelAndView handleRequestInternal( HttpServletRequest request,  
                                              HttpServletResponse response) throws Exception {  
        ModelAndView mav = new ModelAndView("hello");  
        mav.addObject("message", "Hello World!");  
        return mav; } }
```

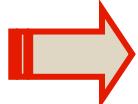
Spring Controller

AbstractController opera automaticamente direttive di caching verso il cliente per obbligarlo a caching locale (2 minuti nell'esempio di config XML mostrato sotto)

```
<bean id="sampleController" class="samples.SampleController">
    <property name="cacheSeconds" value="120"/> </bean>
```

- **ParameterizableViewController**
simile all'esempio precedente, *con possibilità di specificare nome view in Web application context* (senza bisogno di farne hard-code nella classe Java del controllore)
- **UrlFilenameViewController**
esamina URL passato, estraе filename del file richiesto e lo *usa automaticamente come nome di view*. Ad es:

`http://www.springframework.org/index.html`

 `view name = index`

Command Controller

Command controller permettono di associare dinamicamente parametri di HttpServletRequest verso oggetti dati specificati (simili a Struts ActionForm, ma senza bisogno di implementare una specifica interfaccia)

Alcuni controller disponibili:

- ***AbstractCommandController***
nessuna funzionalità form, solo consente di specificare che fare con oggetto command (JavaBean) popolato automaticamente coi parametri richiesta
- ***AbstractFormController***
offre supporto per form; dopo che utente ha compilato form, mette i campi in oggetto command. Programmatore deve specificare metodi per determinare quali view utilizzare per presentazione form
- ***SimpleFormController***
oltre al precedente, anche nome view per form, nome view per pagina da mostrare all'utente quando form submission completata con successo, ...
- ***AbstractWizardFormController***
programmatore deve implementare metodi (abstract) validatePage(), processFinish() e processCancel(); altri metodi di cui fare overriding sono referenceData (per passare model ad una vista sotto forma di oggetto Map); getTargetPage (se wizard vuole cambiare ordine pagine o omettere pagine dinamicamente), onBindAndValidate(se si desidera overriding del flusso usuale di associazione valori e validazione form)

Spring Handler

Funzionalità base: *fornitura di HandlerExecutionChain*, che contiene *un handler* per la richiesta e può contenere una *lista di handler interceptor* da applicare alla richiesta, prima o dopo esecuzione handler

All'arrivo di una richiesta, *DispatcherServlet* la gira a handler per ottenere un HandlerExecutionChain appropriato; poi DispatcherServlet esegue i vari passi specificati nella chain

Concetto potente e molto generale: pensate a handler custom che determina una specifica catena non solo sulla base dell'URL della richiesta ma anche dello stato di sessione associata

Diverse possibilità per handler mapping in Spring. Maggior parte estendono *AbstractHandlerMapping* e condividono le proprietà seguenti:
interceptors: lista degli intercettori
defaultHandler: default da utilizzare quando no matching specifico possibile
order: basandosi su questa proprietà (`org.springframework.core.Ordered`), Spring ordina tutti handler mapping disponibile e sceglie il primo in lista

...

Spring View

Spring mette a disposizione anche componenti detti “*view resolver*” per *semplificare rendering di un model* su browser, senza legarsi a una specifica tecnologia per view (ad es. collegandosi a JSP, Velocity template, ...)

Due interfacce fondamentali sono *ViewResolver* e *View*:

- *ViewResolver* per effettuare mapping fra nomi view e reale implementazione di view
- *View* per preparazione richieste e gestione richiesta verso una tecnologia di view

Possibili ViewResolver:

- *AbstractCachingViewResolver*
realizza trasparentemente caching di view per ridurre tempi preparazione
- *XmlViewResolver*
Accetta file di configurazione XML con stessa DTD della classica bean factory Spring. File di configurazione /WEB-INF/views.xml
- *UrlBasedViewResolver*
Risolve direttamente nomi simbolici di view verso URL
- ...

Spring tag e gestione eccezioni

Spring tag library

- Ampio set di tag specifici per Spring per gestire elementi di form quando si utilizzano JSP e Spring MVC
 - Accesso a oggetto command
 - Accesso a dati su cui lavora controller

Libreria di tag contenuta in spring.jar, descrittore chiamato spring-form.tld, per utilizzarla:

```
<%@ taglib prefix="form"
    uri="http://www.springframework.org/tags/form" %>
```

dove form è prefisso che si vuole utilizzare per indicare tag nella libreria

Gestione eccezioni

HandlerExceptionResolver per semplificare gestione di **eccezioni inattese durante esecuzione controller**; eccezioni contengono info su handler che stava eseguendo al momento dell'eccezione

- Implementazione di interfaccia HandlerExceptionResolver (metodo `resolveException (Exception, Handler)` e restituzione di oggetto `ModelAndView`)
- Possibilità di uso di classe `SimpleMappingExceptionResolver`, con mapping automatico fra nome classe eccezione e nome view

Convenzioni e configurazione di default

In molti casi è sufficiente usare *convenzioni pre-stabilite e ragionevoli default per fare mapping senza bisogno di configurazione*

Convention-over-configuration

Riduzione quantità di configurazioni necessarie per configurare handler mapping, view resolver, istanze ModelAndView, ...

Vantaggi soprattutto in termini di prototipazione rapida

Suggerimento di studio ☺: distribuzione Spring ufficiale contiene una applicazione Web che mostra caso pratico delle differenze convention-over-configuration (sia per model che per view che per controller): nella cartella '`samples/showcases/mvc-convention`'

Convenzioni sulla configurazione di default

Ad esempio, per *Controller*, la classe *ControllerClassNameHandlerMapping* usa convenzione per determinare mapping fra URL e istanze controller. Ad es.:

```
public class ViewShoppingCartController implements Controller {  
    public ModelAndView handleRequest(HttpServletRequest request,  
        HttpServletResponse response) {...} }  
  
<bean  
    class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>  
<bean id="viewShoppingCart"  
    class="x.y.z.ViewShoppingCartController"> ... </bean>
```

ControllerClassNameHandlerMapping trova vari bean controller definiti e toglie “Controller” dal nome per definire automaticamente mapping. Ad es:
WelcomeController si mappa su URL '/welcome'*
HomeController si mappa su URL '/home'*
IndexController si mappa su URL '/index'*

Per un esempio completo e semplice di utilizzo Spring MVC, vedi esercitazione opzionale...

Appendice 1 (opzionale)

- **Introduzione ad Aspect Oriented Programming (AOP)**
- **Supporto AOP in Spring**

Aspect Oriented Programming (AOP)

Una brevissima introduzione non esaustiva a tecniche di AOP

- Aspect Oriented programming (AOP) come approccio di design e tecnica per *semplificare l'applicazione di cross-cutting concern (problematiche trasversali alla logica applicativa)*
- Esempi di cross-cutting concern
 - Logging
 - Locking
 - Gestione degli eventi
 - Gestione delle transazioni
 - Sicurezza e auditing
- Concetti rilevanti per AOP:
 - *Joinpoint*
 - *Advice*
 - *Pointcut e Aspect*
 - *Weaving e Target*
 - *Introduction*

AOP: Joinpoint & Advice

Joinpoint

- Punto ben definito del codice applicativo, anche determinato a runtime, *dove può essere inserita logica addizionale*
- Esempi di joinpoint
 - *Invocazione di metodi*
 - Inizializzazione di classi
 - Inizializzazione di oggetti (creazione di istanze)

Advice

- *Codice con logica addizionale* che deve essere eseguito ad un determinato joinpoint
- Tipi di Advice
 - *before advice* eseguono *prima* del joinpoint
 - *after advice* eseguono *dopo* il joinpoint
 - *around advice* eseguono *attorno (around)* al joinpoint

AOP: Pointcut & Aspect

Pointcut

- ❑ *Insieme di joinpoint usati per definire quando eseguire un advice*
- ❑ Controllo fine e flessibile su come applicare advice al codice applicativo
- ❑ Ad esempio:
 - Invocazione di metodo è un tipico joinpoint
 - Un tipico pointcut è l'insieme di tutte le invocazioni di metodo in una classe determinata
- ❑ *Pointcut possono essere composti in relazioni anche complesse per vincolare il momento di esecuzione dell'advice corrispondente*

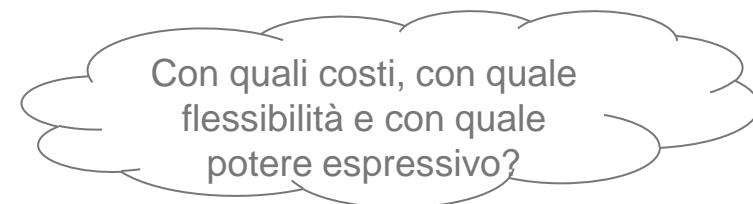
Aspect

- ❑ Aspect come *combinazione di advice e pointcut*

AOP: Weaving & Target

Weaving

- Processo *dell'effettivo inserimento di aspect dentro il codice applicativo* nel punto appropriato
- Tipi di weaving
 - A tempo di compilazione
 - Runtime



Target

- Un *oggetto* il cui *flusso di esecuzione viene modificato* da qualche processo AOP
- Viene anche indicato qualche volta come *oggetto con advice (advised object)*

A questo punto, se voi dovreste implementare AOP in Spring (in ambiente Java in generale), che tipo di approccio usereste?

Spring realizza AOP sulla base **dell'utilizzo di proxy**

- Se si desidera creare una classe advised, occorre utilizzare la classe **ProxyFactory** per *creare un proxy per un'istanza di quella classe*, fornendo a **ProxyFactory** tutti gli aspect con cui si desidera informare il proxy

HelloWorld usando Spring AOP

- Gli *advice* Spring sono scritti in Java (nessun linguaggio AOP-specific)
- Pointcut tipicamente specificati in file XML di configurazione
- Spring supporta **solo joinpoint a livello di metodo** (ad esempio, impossibile associare advice alla modifica di un campo di un oggetto)

```
public class MessageDecorator implements MethodInterceptor {  
  
    public Object invoke(MethodInvocation invocation)  
        throws Throwable {  
        System.out.print("Hello ");  
        Object retVal = invocation.proceed();  
        System.out.println("!");  
        return retVal;  
    }  
  
}
```

HelloWorld usando Spring AOP

- Uso della classe *ProxyFactory* per creare proxy dell'oggetto target
- Anche modalità più di base, *tramite uso di possibilità predeterminate e file XML, senza istanziare uno specifico proxy per AOP*

```
public static void main(String[] args) {  
    MessageWriter target = new MessageWriter();  
    ProxyFactory pf = new ProxyFactory();  
    // aggiunge advice alla coda della catena dell'advice  
    pf.addAdvice(new MessageDecorator());  
    // configura l'oggetto dato come target  
    pf.setTarget(target);  
    // crea un nuovo proxy in accordo con le configurazioni  
    // della factory  
    MessageWriter proxy = (MessageWriter) pf.getProxy();  
    proxy.writeMessage();  
    // Come farei invece a supportare lo stesso comportamento  
    // con chiamata diretta al metodo dell'oggetto target?  
    ... } }
```

Appendice 2 (opzionale)

- Parentesi su Dependency Injection in Spring e piccolo esempio
- Come realizzare un supporto per dependency injection molto semplice in Java?

Parentesi finale: Dependency Injection e HelloWorld

Per fissare le idee in modo molto concreto, proviamo a vedere con il *più semplice degli esempi* se a questo punto del corso capiamo fino in fondo che cosa si intende per *dependency injection*, quali vantaggi produce e che tipo di supporto è necessario per realizzarla

```
public class HelloWorld {  
  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

Quali problemi?

Necessità di cambiare il codice (e di ricompilare) per imporre una modifica del messaggio

Codice *non estensibile e modificabile*

Con Argomenti a Linea di Comando

```
public class HelloWorldWithCommandLine {  
    public static void main(String[] args) {  
        if(args.length > 0) {  
            System.out.println(args[0]);  
        } else { System.out.println("Hello World!"); }  
    } }
```

In questo modo si **“esternalizza” il contenuto del messaggio**, che viene letto a runtime dagli argomenti a linea di comando

Problemi?

- Codice responsabile del rendering del messaggio (**println**) si *occupa anche di ottenere il messaggio*
 - Cambiare come il messaggio deve essere ottenuto obbliga a cambiare il codice del renderer
- Il renderer *non può essere modificato facilmente* (messaggio verso stderr? O in tag HTML invece che plain text?)

Disaccoppiamento

- 1) Disaccoppiamento dell'*implementazione della logica del message provider* rispetto al resto del codice tramite *creazione di una classe separata*

```
public class HelloWorldMessageProvider {  
  
    public String getMessage() {  
        return "Hello World!"; } }
```

- 2) Disaccoppiamento *dell'implementazione della logica di message rendering* dal resto del codice

La logica di message rendering è data all'oggetto

HelloWorldMessageProvider *da qualcun altro* – questo è ciò che si intende con Dependency Injection

```
public class StandardOutMessageRenderer {  
  
    private HelloWorldMessageProvider messageProvider = null;  
  
    public void render() {  
  
        if (messageProvider == null) {  
  
            throw new RuntimeException("You must set property messageProvider of  
            class:" + StandardOutMessageRenderer.class.getName()); } }
```

Disaccoppiamento

```
// continua
System.out.println(messageProvider.getMessage()); }

// dependency injection tramite metodo setter
public void setMessageProvider(HelloWorldMessageProvider provider)
    { this.messageProvider = provider;
    }

public HelloWorldMessageProvider getMessageProvider() {
    return this.messageProvider;
}
}
```

HelloWorld con Disaccoppiamento

```
public class HelloWorldDecoupled {  
  
    public static void main(String[] args) {  
        StandardOutMessageRenderer mr =  
            new StandardOutMessageRenderer();  
        HelloWorldMessageProvider mp =  
            new HelloWorldMessageProvider();  
        mr.setMessageProvider(mp);  
        mr.render();  
    } }  
}
```

A questo punto, la logica del message provider e quella del message renderer sono separate dal resto del codice

Quali problemi ancora?

- Implementazioni specifiche di MessageRenderer e di MessageProvider sono hard-coded nella logica applicativa (in questo launcher in questo lucido)
- Aumentiamo il disaccoppiamento ***tramite interfacce***

HelloWorld con Disaccoppiamento: Interfacce

```
public interface MessageProvider {
    public String getMessage(); }

public class HelloWorldMessageProvider
    implements MessageProvider {
    public String getMessage() {
        return "Hello World!";
    }
}

public interface MessageRenderer {
    public void render();
    public void setMessageProvider(MessageProvider provider);
    public MessageProvider getMessageProvider();
}
```

HelloWorld con Disaccoppiamento: Interfacce

```
public class StandardOutMessageRenderer
        implements MessageRenderer {
    // MessageProvider è una interfaccia Java ora
    private MessageProvider messageProvider = null;
    public void render() {
        if (messageProvider == null) {
            throw new RuntimeException("You must set property messageProvider
of class:" + StandardOutMessageRenderer.class.getName());      }
        System.out.println(messageProvider.getMessage());  }
    ...
    public void setMessageProvider(MessageProvider provider) {
        this.messageProvider = provider;      }

    public MessageProvider getMessageProvider() {
        return this.messageProvider;  }
}
```

HelloWorld con Disaccoppiamento: Interfacce

- Rimane responsabilità del launcher di effettuare la “dependency injection”

```
public class HelloWorldDecoupled {  
  
    public static void main(String[] args) {  
        MessageRenderer mr = new StandardOutMessageRenderer();  
        MessageProvider mp = new HelloWorldMessageProvider();  
        mr.setMessageProvider(mp);  
        mr.render();  
    }  
}
```

- Ora è possibile modificare la logica di message rendering senza alcun impatto sulla logica di message provider
- Allo stesso modo, è possibile cambiare la logica di message provider senza bisogno di modificare la logica di message rendering

HelloWorld con Disaccoppiamento: Interfacce

Quali problemi ancora?

- L'uso di differenti implementazioni delle interfacce **MessageRenderer** o **MessageProvider** necessita comunque di una *modifica (limitata) del codice della logica di business logic (launcher)*
- => Creare una *semplice classe factory* che legga i nomi delle classi desiderate per le implementazioni delle interfacce da un file (*property file*) e le istanzi a runtime, facendo le veci dell'applicazione

HelloWorld con Classe Factory

```
public class MessageSupportFactory {  
    private static MessageSupportFactory instance = null;  
    private Properties props = null;  
    private MessageRenderer renderer = null;  
    private MessageProvider provider = null;  
  
    private MessageSupportFactory() {  
        props = new Properties();  
        try {  
            props.load(new FileInputStream("msf.properties"));  
            // ottiene i nomi delle classi per le interfacce  
            String rendererClass = props.getProperty("renderer.class");  
            String providerClass = props.getProperty("provider.class");  
            renderer = (MessageRenderer) Class.forName(rendererClass).  
                newInstance();  
            provider = (MessageProvider) Class.forName(providerClass).  
                newInstance();  
        } catch (Exception ex) { ex.printStackTrace(); }  
    }  
}
```

HelloWorld con Classe Factory

```
static { instance = new MessageSupportFactory(); }

public static MessageSupportFactory getInstance() {
    return instance;
}

public MessageRenderer getMessageRenderer() {
    return renderer;
}

public MessageProvider getMessageProvider() {
    return provider;
}

public class HelloWorldDecoupledWithFactory {

    public static void main(String[] args) {
        MessageRenderer mr = MessageSupportFactory.getInstance().
            getMessageRenderer();
        MessageProvider mp = MessageSupportFactory.getInstance().
            getMessageProvider();
        mr.setMessageProvider(mp);
        mr.render();
    }
}
```

HelloWorld con Classe Factory

File di proprietà

```
# msf.properties  
renderer.class=StandardOutMessageRenderer  
provider.class=HelloWorldMessageProvider
```

- Ora le implementazioni di message provider e message renderer *possono essere modificate tramite semplice modifica del file di proprietà*

Quali problemi ancora?

- Necessità di *scrivere molto “glue code”* per mettere insieme l'applicazione
- Necessità di scrivere una *classe MessageSupportFactory*
- L'istanza di *MessageProvider* deve essere ancora *iniettata manualmente* nell'implementazione di MessageRenderer

HelloWorld usando Spring

```
public class HelloWorldSpring {  
  
    public static void main(String[] args) throws Exception {  
        // ottiene il riferimento a bean factory  
        BeanFactory factory = getBeanFactory();  
        MessageRenderer mr = (MessageRenderer) factory.  
            getBean("renderer");  
        MessageProvider mp = (MessageProvider) factory.  
            getBean("provider");  
        mr.setMessageProvider(mp);  
        mr.render();  
    }  
  
    // continua...
```

HelloWorld usando Spring

```
// Possibilità di scrivere il proprio metodo getBeanFactory()
// a partire da Spring DefaultListableBeanFactory class

private static BeanFactory getBeanFactory() throws Exception {
    DefaultListableBeanFactory factory = new
        DefaultListableBeanFactory();
    // creare un proprio lettore delle definizioni
    PropertiesBeanDefinitionReader rdr = new
        PropertiesBeanDefinitionReader(factory);
    // caricare le opzioni di configurazione
    Properties props = new Properties();
    props.load(new FileInputStream("beans.properties"));
    rdr.registerBeanDefinitions(props);
    return factory;
}
```

HelloWorld con Spring: Quali Problemi?

Quali vantaggi già raggiunti in questo modo?

- Eliminata la necessità di produrre glue code (*MessageSupportFactory*)
- Migliore gestione degli errori e meccanismo di configurazione completamente disaccoppiato*

Quali problemi ancora?

- Il codice di startup deve avere conoscenza delle dipendenze di MessageRenderer, deve ottenerle e deve passarle a MessageRenderer*
 - In questo caso *Spring agirebbe come non più di una classe factory sofisticata*
 - Rimarrebbe al programmatore il compito di fornire il proprio metodo **getBeanFactory()** usando le API di basso livello del framework Spring

HelloWorld con Spring Dependency Injection

Finalmente ☺, utilizzo della Dependency Injection del framework Spring

File di configurazione

```
#Message renderer
renderer.class=StandardOutMessageRenderer
# Chiede a Spring di assegnare l'effettivo provider alla
# proprietà MessageProvider del bean Message renderer
renderer.messageProvider(ref)=provider
```

```
#Message provider
provider.class=HelloWorldMessageProvider
```

HelloWorld con Spring Dependency Injection

```
public class HelloWorldSpringWithDI {  
    public static void main(String[] args) throws Exception {  
        BeanFactory factory = getBeanFactory();  
        MessageRenderer mr = (MessageRenderer) factory.  
            getBean("renderer");  
        // Nota che non è più necessaria nessuna injection manuale  
        // del message provider al message renderer  
        mr.render();    }  
  
    private static BeanFactory getBeanFactory() throws Exception {  
        DefaultListableBeanFactory factory = new  
            DefaultListableBeanFactory();  
        PropertiesBeanDefinitionReader rdr = new  
            PropertiesBeanDefinitionReader(factory);  
        Properties props = new Properties();  
        props.load(new FileInputStream("beans.properties"));  
        rdr.registerBeanDefinitions(props);  
        return factory; }  
}
```

HelloWorld con Spring Dependency Injection: Ultime Osservazioni

- Il metodo **main()** deve semplicemente ottenere il bean **MessageRenderer** e richiamare **render()**
 - Non deve ottenere prima il MessageProvider e configurare la proprietà MessageProvider del bean MessageRenderer
 - *“wiring” realizzato automaticamente dalla Dependency Injection di Spring*
- Nota che non serve **nessuna modifica** alle classi da collegare insieme tramite dependency injection
- Queste classi **NON fanno alcun riferimento a Spring**
 - *Nessun bisogno di implementare interfacce* del framework Spring
 - *Nessun bisogno di estendere classi* del framework Spring
- **Classi come POJO puri** che possono essere sottoposte a *testing senza alcuna dipendenza da Spring*

Spring Dependency Injection con file XML

Più usualmente, dipendenze dei bean sono specificate tramite un file XML

```
<beans>
    <bean id="renderer"
        class="StandardOutMessageRenderer">
        <property name="messageProvider">
            <ref local="provider"/>
        </property>
    </bean>
    <bean id="provider"
        class="HelloWorldMessageProvider"/>
</beans>
```

Spring Dependency Injection con file XML

```
public class HelloWorldSpringWithDIXMLFile {  
  
    public static void main(String[] args) throws Exception {  
        BeanFactory factory = getBeanFactory();  
        MessageRenderer mr = (MessageRenderer) factory.  
            getBean("renderer");  
        mr.render();  
    }  
  
    private static BeanFactory getBeanFactory() throws Exception  
{  
        BeanFactory factory = new XmlBeanFactory(new  
            FileSystemResource("beans.xml"));  
        return factory;  
    }  
}
```

Spring DI con file XML: Costruttore per MessageProvider

```
<beans>
    <bean id="renderer" class="StandardOutMessageRenderer">
        <property name="messageProvider">
            <ref local="provider"/>
        </property>
    </bean>
    <bean id="provider" class="ConfigurableMessageProvider">
        <constructor-arg>
            <value>Questo è il messaggio configurabile</value>
        </constructor-arg>
    </bean>
</beans>
```

Spring DI con file XML: Costruttore per MessageProvider

```
public class ConfigurableMessageProvider
    implements MessageProvider {

    private String message;
    public ConfigurableMessageProvider(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

}
```

HelloWorld usando Spring AOP

Infine, se volessimo scrivere a video “Hello World!” sfruttando AOP

```
public class MessageWriter implements IMessageWriter{  
  
    public void writeMessage() {  
        System.out.print("World");  
    }  
  
}
```

- *joinpoint* è l’invocazione del metodo **writeMessage()**
- *Necessità di un “around advice”*

HelloWorld usando Spring AOP

- Gli *advice* Spring sono scritti in Java (nessun linguaggio AOP-specific)
- Pointcut tipicamente specificati in file XML di configurazione
- Spring supporta *solo joinpoint a livello di metodo* (ad esempio, impossibile associare advice alla modifica di un campo di un oggetto)

```
public class MessageDecorator implements MethodInterceptor {  
  
    public Object invoke(MethodInvocation invocation)  
        throws Throwable {  
        System.out.print("Hello ");  
        Object retVal = invocation.proceed();  
        System.out.println("!");  
        return retVal;  
    }  
}
```

HelloWorld usando Spring AOP

- Uso della classe *ProxyFactory* per creare il proxy dell'oggetto target
- Anche modalità più di base, *tramite uso di possibilità predeterminate e file XML, senza istanziare uno specifico proxy per AOP*

```
public static void main(String[] args) {  
    MessageWriter target = new MessageWriter();  
    ProxyFactory pf = new ProxyFactory();  
    // aggiunge advice alla coda della catena dell'advice  
    pf.addAdvice(new MessageDecorator());  
    // configura l'oggetto dato come target  
    pf.setTarget(target);  
    // crea un nuovo proxy in accordo con le configurazioni factory  
    MessageWriter proxy = (MessageWriter) pf.getProxy();  
    proxy.writeMessage();  
    // Come farei invece a supportare lo stesso comportamento  
    // con chiamata diretta al metodo dell'oggetto target?  
    ... } }
```