



**Corso di Laurea in Informatica**  
**III Anno Triennale**  
**Programmazione Distribuita – classe 1**



# Context and Dependency Injection

Delfina Malandrino

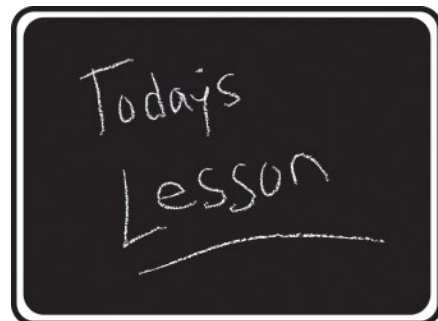
[dmalandrino@unisa.it](mailto:dmalandrino@unisa.it)

<http://www.unisa.it/docenti/delfinamalandrino>

1

## Organizzazione della lezione

- Introduzione
  - Dependency Injection
  - Life-cycle Management
  - Interception
  - Loose Coupling and Strong Typing
  - Deployment Descriptor
- Come scrivere un CDI Bean
  - Injection
  - Qualifiers
  - Producers/Disposers
  - Scope
- Interceptors
  - Classi Interceptor e Ciclo di vita
  - Interceptor multipli
- Decorators e Eventi
- Conclusioni



2

## Organizzazione della lezione

- Introduzione
  - Dependency Injection
  - Life-cycle Management
  - Interception
  - Loose Coupling and Strong Typing
  - Deployment Descriptor
- Come scrivere un CDI Bean
  - Injection
  - Qualifiers
  - Producers/Disposers
  - Scope
- Interceptors
  - Classi Interceptor e Ciclo di vita
  - Interceptor multipli
- Decorators e Eventi
- Conclusioni



3

## Inversion of control

- La prima versione di Java EE ha introdotto il concetto di **inversion of control (IoC)**, nel senso che:
  - il container prende il controllo del business code...
  - ... in modo da poter fornire servizi (come transazioni, sicurezza, ciclo di vita, etc) senza che il programmatore debba necessariamente scriverli
- Prendere il controllo significa gestire in toto il nostro programma, configurandone il contesto di esecuzione (l'ambiente): **Context**...
- ... e risolvendone le dipendenze con altre componenti: **Dependency Injection**
- Ideato da Martin Fowler nel 2004 ed alla base dello sviluppo del calcolo enterprise (application server e container)

4

## Understanding Beans

- Sono gli attori principali delle versioni di Java Enterprise dalle 6 in poi
- In pratica tutte le componenti JEE sono (o possono essere resi) CDI Managed beans
- Concetto di base: **“loose coupling, strong typing”**: unire i vantaggi di due mondi che si pensavano contrapposti e mutuamente esclusivi
- Il disaccoppiamento permette di poter inserire interceptors, decorators e gestione di eventi in tutta la piattaforma ...
- ... armonizzando in un unico ambiente il web layer e il back-end layer

5

## Understanding Beans

- Un pò di termini: “Plain Old Java Object” ⇒ POJO
  - Classi java eseguite in una JVM
- Java SE ha i suoi JavaBeans: POJOs eseguiti nella JVM che seguono delle convenzioni:
  - Naming, getter/setter, costruttore di default
- Java EE ha i suoi Enterprise JavaBeans (EJB): sono eseguiti in un container, contengono metadati, costruttore non final, e usufruiscono dei servizi del container (sicurezza, transazioni, etc.)



6

## Organizzazione della lezione

- Introduzione
  - Dependency Injection
  - Life-cycle Management
  - Interception
  - Loose Coupling and Strong Typing
  - Deployment Descriptor
- Come scrivere un CDI Bean
  - Injection
  - Qualifiers
  - Producers/Disposers
  - Scope
- Interceptors
  - Classi Interceptor e Ciclo di vita
  - Interceptor multipli
- Decorators e Eventi
- Conclusioni



7

## Pattern dependency injection

- **Dependency Injection (DI)** è un design pattern che disaccoppia componenti dipendenti
- E' parte dell'inversion of control...
- ... dove l'inversion riguarda il processo di ottenere le dipendenze necessarie
- Termine coniato per la prima volta da Martin Fowler

8

## Pattern dependency injection

## Esempio

- Una maniera di vedere il pattern: inversione di Java Naming and Directory Interface (JNDI)
  - JNDI fornisce a richiesta un riferimento ad un certo oggetto
  - Il pattern fa sì che il container inietti la dipendenza nell'oggetto che ne ha bisogno

9

## Pattern dependency injection

Invece di avere un oggetto che cerca altri oggetti...  
... il container fa inject di questi oggetti dipendenti  
al posto nostro!!!

- Questo viene anche detto *Hollywood Principle*, "Don't call us?" (lookup objects), "we'll call you" (inject objects)

10

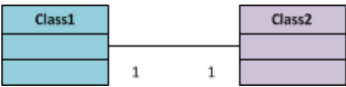
# Classi “Tightly coupled”

```
//pseudolinguaggio...
public class Class1
{
    public Class2 c;
}

public class Class2
{
    //...
}
```

La classe Class1 è strettamente accoppiata alla classe Class2

Two Tightly Coupled Classes



11

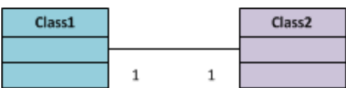
# Classi “Tightly coupled”

```
//pseudolinguaggio...
public class Class1
{
    public Class2 c;
}

public class Class2
{
    //...
}
```

La classe Class1 è strettamente accoppiata alla classe Class2  
... in quanto ha una variabile che fa riferimento ad una sua istanza

Two Tightly Coupled Classes



12

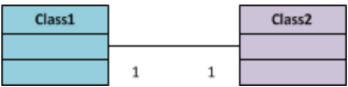
# Classi “Tightly coupled”

```
//pseudolinguaggio...
public class Class1
{
    public Class2 c;
}

public class Class2
{
    //...
```

- › La classe Class1 è strettamente accoppiata alla classe Class2
- › ... in quanto ha una variabile che fa riferimento ad una sua istanza
- › ...definita altrove

Two Tightly Coupled Classes



13

# Classi “Loosely coupled” (?)

```
//pseudolinguaggio...
public class Class1
{
    public IClass2 c;
}

public interface IClass2
{
    //...
}

public class Class2 implements IClass2
{
    //...
```

- › La classe Class1 sembra meno accoppiata alla classe Class2

Two Loosely Coupled Classes



14

# Classi "Loosely coupled" (?)

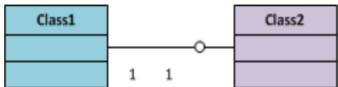
```
//pseudolinguaggio...
public class Class1
{
    public IClass2 c;
}

public interface IClass2
{
    //...
}

public class Class2 implements IClass2
{
    //...
}
```

- › La classe Class1 sembra meno accoppiata alla classe Class2
- › E in effetti sembra usare una interfaccia e non ha riferimenti alla classe Class2

Two Loosely Coupled Classes



15

# Classi "Loosely coupled" (?)

```
//pseudolinguaggio...
public class Class1
{
    public IClass2 c;
}

public interface IClass2
{
    //...
}

public class Class2 implements IClass2
{
    //...
}
```

- › La classe Class1 sembra meno accoppiata alla classe Class2
- › E in effetti sembra usare una interfaccia e non ha riferimenti alla Class2
- › Definizione di Class2

Two Loosely Coupled Classes



16



## Non sono "Loosely coupled"

```
//pseudolinguaggio...
public class Class1
{
    public IClass2 c;
    public Class1()
    {
        c = new Class2();
    }
}

public interface IClass2
{
    //...
}

public class Class2
{
    //...
}
```

Da qualche parte nel costruttore, deve necessariamente istanziare la classe Class2

17

## Non sono "Loosely coupled"

```
//pseudolinguaggio...
public class Class1
{
    public IClass2 c;
    public Class1()
    {
        c = new Class2();
    }
}

public interface IClass2
{
    //...
}

public class Class2
{
    //...
}
```

Da qualche parte nel costruttore, deve necessariamente istanziare la Class2

Ad esempio qui

Tuttavia, questa implementazione di Loose coupling presenta un problema. Se **Class1** è responsabile della creazione di una nuova istanza di **Class2**, si ha solo l'illusione del loose coupling perché **Class1** deve comunque sapere di **Class2**

A causa del costruttore, **Class1** è ancora tightly coupled a **Class2**

18

## Non sono "Loosely coupled"

```
//pseudolinguaggio...
public class Class1
{
    public IClass2 c;
    public Class1()
    {
        c = new Class2();
    }
}

public interface IClass2
{
    //...
}

public class Class2
{
    //...
}
```

- › Da qualche parte nel costruttore, deve necessariamente istanziare la Class2
- › Ad esempio qui
- › Come si risolve questo problema?

Con la Dependency Injection!

19

## Non sono "Loosely coupled"

```
//pseudolinguaggio...
public class Class1
{
    public IClass2 c;
    public Class1()
    {
        c = new Class2();
    }
}

public interface IClass2
{
    //...
}

public class Class2
{
    //...
}
```

- › Da qualche parte nel costruttore, deve necessariamente istanziare la Class2
- › Ad esempio qui
- › Come si risolve questo problema? Con la Dependency Injection!

Una entità terza che risolve questo ultimo pezzettino di "accoppiamento" tra le classi

20

# L'iniezione di dipendenza

```
//pseudolinguaggio...
public class Class1
{
    public IClass2 c;
    public Class1()
    {
        c = DependencyFactory.Resolve<IClass2>();
    }
}

public interface IClass2
{
    //...
}
```

Nel costruttore viene richiesto l'aiuto di un'altra entità...

Two Loosely Coupled Classes



21

# L'iniezione di dipendenza

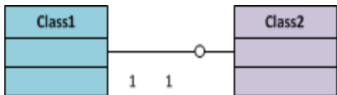
```
//pseudolinguaggio...
public class Class1
{
    public IClass2 c;
    public Class1()
    {
        c = DependencyFactory.Resolve<IClass2>();
    }
}

public interface IClass2
{
    //...
}
```

› Nel costruttore viene richiesto l'aiuto di un'altra entità. ..

... che istanzia un oggetto del giusto tipo, a seconda delle esigenze del "sistema"

Two Loosely Coupled Classes



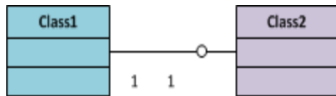
22

## L'iniezione di dipendenza

```
//pseudolinguaggio...
public class Class1
{
    public IClass2 c;
    public Class1()
    {
        c = DependencyFactory.Resolve<IClass2>();
    }
}

public interface IClass2
{
    //...
}
```

Two Loosely Coupled Classes



- › Nel costruttore viene richiesto l'aiuto di un'altra entità...
- ... che istanzia un oggetto del giusto tipo, a seconda delle esigenze del "sistema"
- › Ancora non abbiamo una vera indipendenza però!

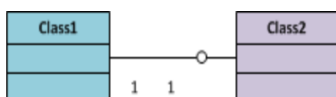
23

## L'iniezione di dipendenza

```
//pseudolinguaggio...
public class Class1
{
    public IClass2 c;
    public Class1()
    {
        c = DependencyFactory.Resolve<IClass2>();
    }
}

public interface IClass2
{
    //...
}
```

Two Loosely Coupled Classes



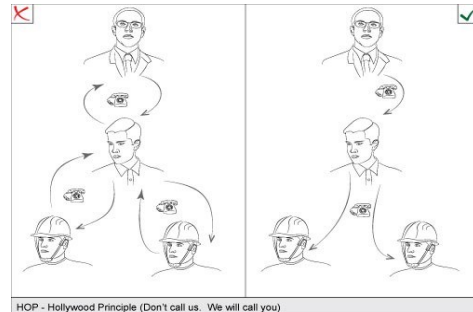
- › Nel costruttore viene richiesto l'aiuto di un'altra entità...
- ... che istanzia un oggetto del giusto tipo, a seconda delle esigenze del "sistema"
- › Ancora non abbiamo una vera indipendenza però!
- › Dobbiamo usare l'Hollywood Principle:  
"Don't call us. We'll call you"

24

## Invece di chiedere

E' il container che decide di iniettare la risorsa al posto giusto  
*"Don't call us, we'll call you"*

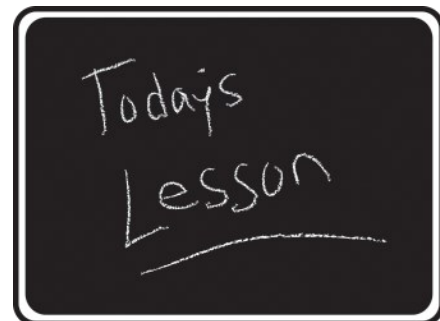
*Don't call us:* lookup objects  
*We'll call you:* inject objects



25

## Organizzazione della lezione

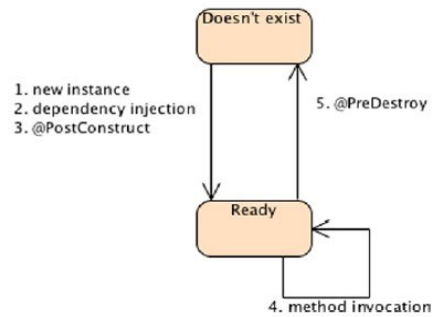
- Introduzione
  - Dependency Injection
  - Life-cycle Management
  - Interception
  - Loose Coupling and Strong Typing
  - Deployment Descriptor
- Come scrivere un CDI Bean
  - Injection
  - Qualifiers
  - Producers/Disposers
  - Scope
- Interceptors
  - Classi Interceptor e Ciclo di vita
  - Interceptor multipli
- Decorators e Eventi
- Conclusioni



26

## Life-Cycle Management

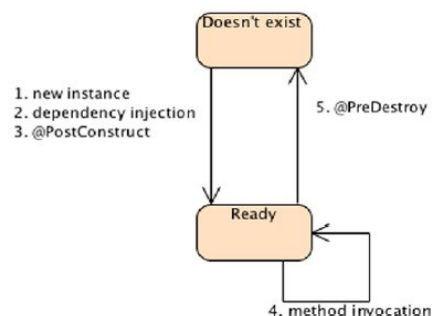
- In un POJO, il ciclo di vita è molto semplice: lo sviluppatore Java crea una nuova istanza con **new** e si attende che il Garbage Collector liberi memoria
- In un CDI Bean, all'interno di un container, non è possibile usare **new**



27

## Life-Cycle Management

- E' il container che è responsabile del ciclo di vita di un bean e quindi della creazione (usando **new**) ...
- ... e poi risolve le dipendenze necessarie e invoca i metodi annotati con:
  - **@PostConstruct** che vengono eseguiti dopo l'istanziamento del bean, dopo che sono state verificate tutte le dipendenze e prima della prima invocazione di metodi di business
  - ... e prima della deallocazione chiama i metodi annotati con **@Pre-Destroy**



28

## Scope di un Bean

- I Beans CDI possono essere stateful, il che significa che sono in esecuzione in uno “scope” (il contesto) ben definito
  - Scope predefiniti: request, session, application e conversation
- Ad esempio:
  - Un session context ed i suoi beans esistono durante il lifetime di una sessione HTTP
  - Durante questo lifetime, i riferimenti iniettati in questi beans saranno consapevoli del contesto
    - Vale a dire, l'intera catena di dependencies di un bean è contestuale. Il container gestisce tutti i beans all'interno dello scope automaticamente al posto del programmatore
    - Al termine della sessione li distruggerà

29

## Scope di un Bean

- Al contrario di componenti stateless (e.g., stateless session beans) o singleton (e.g., Servlets or singletons), client diversi di un bean stateful possono vedere questo bean in stati differenti, perchè lo “scope” può dettare che ognuno veda un bean diverso
  - Client eseguiti nello stesso contesto vedranno lo stesso stato del bean
  - Client in diversi contesti vedranno una diversa istanza
- Tutto gestito automaticamente dal container (nessun controllo da parte del client)

30

## Organizzazione della lezione

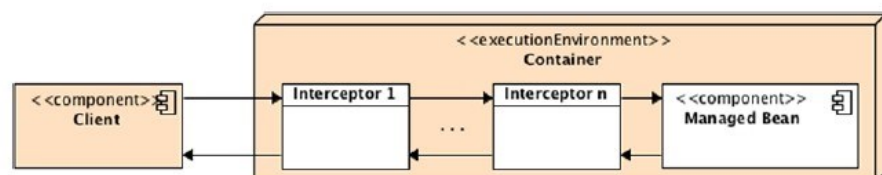
- Introduzione
  - Dependency Injection
  - Life-cycle Management
  - **Interception**
  - Loose Coupling and Strong Typing
  - Deployment Descriptor
- Come scrivere un CDI Bean
  - Injection
  - Qualifiers
  - Producers/Disposers
  - Scope
- Interceptors
  - Classi Interceptor e Ciclo di vita
  - Interceptor multipli
- Decorators e Eventi
- Conclusioni



31

## Interception

- Si frappongono tra invocazioni di metodi di business
- Utili alla Aspect-oriented Programming (AOP), un paradigma che separa i "cross-cutting concerns" di una architettura dal codice di business
- Tipici esempi:
  - Technical concerns: Log ingresso/uscita da un metodo, log della durata della invocazione di un metodo, memorizzazione di statistiche relative all'uso di un metodo, ecc.
  - Business concerns: check extra se un cliente spende più di 10.000 euro, inviare un nuovo ordine di acquisto se una merce in magazzino è ad un livello minimo, ecc.

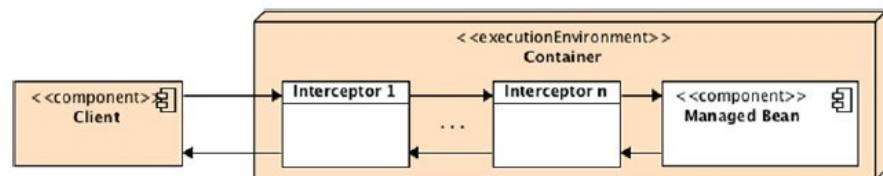


32



## Interception

- CDI Beans supportano questa funzionalità (AOP) offrendo la possibilità di intercettare le invocazioni di metodo con gli interceptor
- Il container si occupa di chiamare gli interceptor prima/dopo l'invocazione del metodo



33

## La potenza degli Interceptor

- Gli interceptor disaccoppiano molto efficacemente i cross-cutting concerns di natura tecnica dalla logica di business
- La maniera in cui il container assicura i servizi agli EJB è attraverso una catena configurabile di interceptors
- L'esecuzione degli interceptor non è a conoscenza alcuna da chi scrive il POJO o l'EJB

34

## Organizzazione della lezione

- Introduzione
  - Dependency Injection
  - Life-cycle Management
  - Interception
  - Loose Coupling and Strong Typing
  - Deployment Descriptor
- Come scrivere un CDI Bean
  - Injection
  - Qualifiers
  - Producers/Disposers
  - Scope
- Interceptors
  - Classi Interceptor e Ciclo di vita
  - Interceptor multipli
- Decorators e Eventi
- Conclusioni



35

## Loose Coupling and Strong Typing

- Gli interceptor rappresentano un mezzo potente per disaccoppiare dettagli tecnici dalla logica di business
- La gestione del ciclo di vita di un bean disaccoppia il bean stesso dalla gestione del suo ciclo di vita
- Esistono altri modi per disaccoppiare
  - I beans possono usare event notifications per disaccoppiare event producer da event consumer

In altre parole... loosely coupling è il  
DNA su cui CDI è stato costruito

36

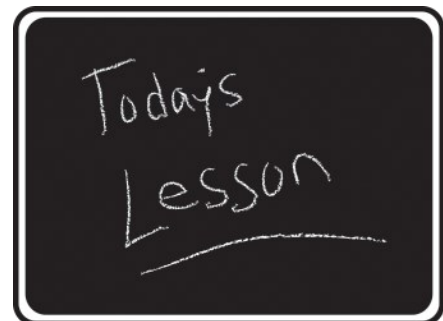
## Loose Coupling and Strong Typing

- Eppure, questo viene ottenuto mantenendo la forte tipizzazione, usando le annotazioni con parametri, per poter legare insieme in maniera “safe” i beans
- L’uso di identificatori “String-based” (cioè XML) viene limitato moltissimo a pochi casi specifici per il deployment

37

## Organizzazione della lezione

- Introduzione
  - Dependency Injection
  - Life-cycle Management
  - Interception
  - Loose Coupling and Strong Typing
  - Deployment Descriptor
- Come scrivere un CDI Bean
  - Injection
  - Qualifiers
  - Producers/Disposers
  - Scope
- Interceptors
  - Classi Interceptor e Ciclo di vita
  - Interceptor multipli
- Decorators e Eventi
- Conclusioni



38

## Deployment Descriptor

- Ogni Java EE specification ha un **xml** deployment descriptor opzionale
  - Per descrivere come una componente o una applicazione deve essere configurata
  - Nei CDI è obbligatorio (**beans.xml**). Usato per configurare alcune funzionalità (interceptors, decorators, alterantives), ma necessario per abilitare CDI
    - Ecco perché CDI ha necessità di identificare i beans nel classpath (beans discovery)
  - Durante la fase di beans discovery che accade la magia
    - Cioè quando CDI trasforma un POJO in un CDI Bean
- Durante la fase di deployment, CDI controlla tutti gli application `jar` e i `war` file, ed ogni volta che incontra un `beans.xml` deployment descriptor, gestisce tutti i POJO che diventeranno CDI Beans

39

## Deployment Descriptor

- Senza un `beans.xml` nel classpath (`META-INF` o `WEB-INF` directory), CDI non sarà capace di usare injection, interception, decoration, ecc.
- **Senza un `beans.xml` CDI non funzionerà!!!**
- Se una Web application contiene diversi `jar` files e si desidera avere CDI abilitato attraverso l'intera applicazione, ogni `jar` avrà necessità del proprio `beans.xml` per attivare CDI ed il bean discovery per ogni `jar`

40

## Organizzazione della lezione

- Introduzione
  - Dependency Injection
  - Life-cycle Management
  - Interception
  - Loose Coupling and Strong Typing
  - Deployment Descriptor
- Come scrivere un CDI Bean
  - Injection
  - Qualifiers
  - Producers/Disposers
  - Scope
- Interceptors
  - Classi Interceptor e Ciclo di vita
  - Interceptor multipli
- Decorators e Eventi
- Conclusioni

41

Lo vedremo  
nella prossima  
lezione



42

## Come scrivere un CDI Bean

- Un CDI Bean può essere ogni classe con una logica di business
- Può essere chiamato direttamente da codice Java attraverso injection oppure può essere invocato da una JSF page

43

## Partiamo da un esempio

Listing 2-1

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

    @Inject
    private EntityManager em;
    private Date instantiationDate;

    @PostConstruct
    private void initDate()
    {
        instantiationDate = new Date();
    }

    @Transactional
    public Book createBook(String title, Float price,
        String description) {
        Book book = new Book(title, price, description);
        book.setIsbn(numberGenerator.generateNumber());
        book.setInstanciationDate(instantiationDate);
        em.persist(book);
        return book;
    }
}
```

↳ Un POJO

44

## Partiamo da un esempio

Listing 2-1

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

    @Inject
    private EntityManager em;
    private Date instantiationDate;

    @PostConstruct
    private void initDate()
    {
        instantiationDate = new Date();
    }

    @Transactional
    public Book createBook(String title, Float price,
        String description) {
        Book book = new Book(title, price, description);
        book.setIsbn(numberGenerator.generateNumber());
        book.setInstantiationDate(instantiationDate);
        em.persist(book);
        return book;
    }
}
```

Un POJO

Dipendenze: il container  
> inietterà prima un  
NumberGenerator

Inject di  
references ad  
altri beans

45

## Partiamo da un esempio

Listing 2-1

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

    @Inject
    private EntityManager em;
    private Date instantiationDate;

    @PostConstruct
    private void initDate()
    {
        instantiationDate = new Date();
    }

    @Transactional
    public Book createBook(String title, Float price,
        String description) {
        Book book = new Book(title, price, description);
        book.setIsbn(numberGenerator.generateNumber());
        book.setInstantiationDate(instantiationDate);
        em.persist(book);
        return book;
    }
}
```

Un POJO

Dipendenze: il container  
inietterà prima un  
NumberGenerator

...e un EntityManager

Inject di  
references ad  
altri beans

46

# Partiamo da un esempio

Listing 2-1

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

    @Inject
    private EntityManager em;
    private Date instantiationDate;

    @PostConstruct
    private void initDate()
    {
        instantiationDate = new Date();
    }

    @Transactional
    public Book createBook(String title, Float price,
        String description) {
        Book book = new Book(title, price, description);
        book.setIsbn(numberGenerator.generateNumber());
        book.setInstantiationDate(instantiationDate);
        em.persist(book);
        return book;
    }
}
```

Un POJO

Dipendenze: il container  
inietterà prima un  
NumberGenerator

... e un EntityManager

Metodo dopo il costruttore  
(invocato dal container)

# Partiamo da un esempio

Listing 2-1

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

    @Inject
    private EntityManager em;
    private Date instantiationDate;

    @PostConstruct
    private void initDate()
    {
        instantiationDate = new Date();
    }

    @Transactional
    public Book createBook(String title, Float price,
        String description) {
        Book book = new Book(title, price, description);
        book.setIsbn(numberGenerator.generateNumber());
        book.setInstantiationDate(instantiationDate);
        em.persist(book);
        return book;
    }
}
```

Un POJO

Dipendenze: il container inietterà  
prima un NumberGenerator  
... e un EntityManager

Metodo dopo il costruttore (invocato  
dal container)

Servizio transazionale (Interceptor  
binding)

Li vedremo  
nella prossima  
lezione



## Organizzazione della lezione

- Introduzione
  - Dependency Injection
  - Life-cycle Management
  - Interception
  - Loose Coupling and Strong Typing
  - Deployment Descriptor
- Come scrivere un CDI Bean
  - Injection
  - Qualifiers
  - Producers/Disposers
  - Scope
- Interceptors
  - Classi Interceptor e Ciclo di vita
  - Interceptor multipli
- Decorators e Eventi
- Conclusioni

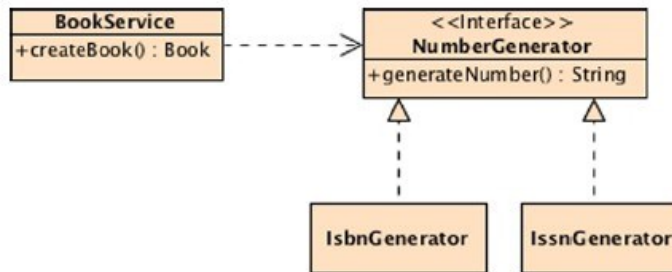
50

## Un esempio di riferimento

- Un sistema per la gestione di libri: insieme di oggetti **Book**, acceduti da un **Customer** che usa un **PurchasedOrder** per acquistarne uno
- Per creare un libro, usiamo una classe di servizio **BookService** che istanzia un oggetto **Book**
  - generando un ID univoco usando una classe **NumberGenerator** e rendendolo persistente in un database
- Il servizio di ID univoco può essere fatto con un ISBN (13 cifre) o con un ISSN con 8 cifre
- Importante: il risultato di **BookService** "dipenderà" da quale tra un **IsbnGenerator** e un **IssnGenerator** verrà usato
  - Cosa significa tutto ciò?

51

## Il diagramma delle classi



- L'interfaccia **NumberGenerator** ha un solo metodo `String generateNumber()`
- Il metodo `generateNumber()` è implementato da **IsbnGenerator** e **IssnGenerator**
- Il **BookService** **dipende** dall'interfaccia per la generazione di un book number

52

## Un esempio di riferimento

- Come facciamo a connettere un **BookService** alla implementazione "ISBN" di **NumberGenerator**?
- La prima soluzione:
  - Usare il buon vecchio metodo con la keyword **new**
    - POJO senza CDI

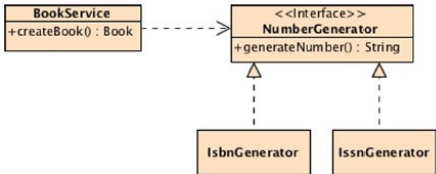
53

Un POJO senza CDI  
(creare dipendenze usando la keyword new)

Listing 2-2

```
public class BookService {  
    private NumberGenerator numberGenerator;  
  
    public BookService() {  
        this.numberGenerator = new ISBNGenerator();  
    }  
  
    public Book createBook(String title, Float price,  
        String description) {  
        Book book = new Book(title, price, description);  
        book.setIsbn(numberGenerator.generateNumber());  
        return book;  
    }  
}
```

Classe BookService



54

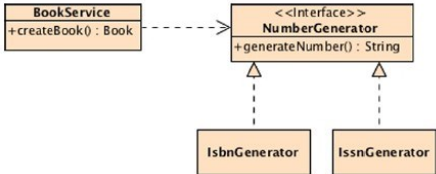
Un POJO senza CDI  
(creare dipendenze usando la keyword new)

Listing 2-2

```
public class BookService {  
    private NumberGenerator numberGenerator;  
  
    public BookService() {  
        this.numberGenerator = new ISBNGenerator();  
    }  
  
    public Book createBook(String title, Float price,  
        String description) {  
        Book book = new Book(title, price, description);  
        book.setIsbn(numberGenerator.generateNumber());  
        return book;  
    }  
}
```

Classe BookService

Una istanza per generare ID univoche



55

Un POJO senza CDI  
(creare dipendenze usando la keyword new)

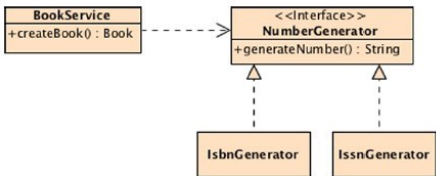
Listing 2-2

```
public class BookService {
    private NumberGenerator numberGenerator;

    public BookService() {
        this.numberGenerator = new ISBNGenerator();
    }

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

Casse BookService  
Una istanza per generare ID univoche  
Il costruttore istanzia il generatore di ID



56

Un POJO senza CDI  
(creare dipendenze usando la keyword new)

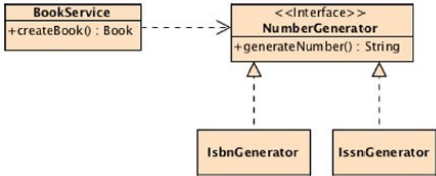
Listing 2-2

```
public class BookService {
    private NumberGenerator numberGenerator;

    public BookService() {
        this.numberGenerator = new ISBNGenerator();
    }

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

Classe BookService  
Una istanza per generare ID univoche  
Il costruttore istanzia il generatore di ID  
Il metodo per creare un libro, setta l'ID con il numero generato



57

## Un esempio di riferimento

- Cosa succede se vogliamo scegliere fra diverse implementazioni e non essere legati solo all'IsbnGenerator?
  - Una soluzione:
    - passare l'implementazione al costruttore e lasciare ad una classe esterna il compito di decidere quale implementazione usare
      - un POJO scegliendo le dipendenze usando il costruttore

58

## Come disaccoppiare: un POJO scegliendo le dipendenze usando il costruttore

Listing 2-3

```
public class BookService {
    private NumberGenerator numberGenerator;

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

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

BookService bs1 = new BookService(new IsbnGenerator())

//...oppure...

BookService bs2 = new BookService(new IssnGenerator())
```

Al costruttore viene passato il  
generatore da usare

59

Come disaccoppiare: un POJO scegliendo le dipendenze usando il costruttore

Listing 2-3

```
public class BookService {
    private NumberGenerator numberGenerator;

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

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

BookService bs1 = new BookService(new IsbnGenerator())
//...oppure...
BookService bs2 = new BookService(new IssnGenerator())
```

Al costruttore viene passato il generatore da usare

e assegnato alla variabile

60

Come disaccoppiare: un POJO scegliendo le dipendenze usando il costruttore

Listing 2-3

```
public class BookService {
    private NumberGenerator numberGenerator;

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

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

BookService bs1 = new BookService(new IsbnGenerator())
//...oppure...
BookService bs2 = new BookService(new IssnGenerator())
```

Al costruttore viene passato il generatore da usare

e assegnato alla variabile

... e poi utilizzato

61

## Come disaccoppiare: un POJO scegliendo le dipendenze usando il costruttore

Listing 2-3

```
public class BookService {
    private NumberGenerator numberGenerator;

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

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

BookService bs1 = new BookService(new IsbnGenerator())
//...oppure...
BookService bs2 = new BookService(new IssnGenerator())
```

Al costruttore viene passato il generatore da usare

e assegnato alla variabile

... e poi utilizzato

Come chiamarlo

Una classe esterna potrà usare il BookService con l'implementazione di cui ha bisogno

62

## Come disaccoppiare: un POJO scegliendo le dipendenze usando il costruttore

Listing 2-3

```
public class BookService {
    private NumberGenerator numberGenerator;

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

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

BookService bs1 = new BookService(new IsbnGenerator())
//...oppure...
BookService bs2 = new BookService(new IssnGenerator())
```

Al costruttore viene passato il generatore da usare

e assegnato alla variabile

... e poi utilizzato

Come chiamarlo

Classico esempio di *Inversion of Control* costruito manualmente

Il controllo di creare la dipendenza tra BookService e NumberGenerator è invertito dal momento che è dato ad una classe esterna e non alla classe stessa

63

## Come disaccoppiare: un POJO scegliendo le dipendenze usando il costruttore

Listing 2-3

```
public class BookService {
    private NumberGenerator numberGenerator;

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

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

```
BookService bs1 = new BookService(new IsbnGenerator())
//...oppure...
BookService bs2 = new BookService(new IssnGenerator())
```

Un container può fare tutto questo  
automaticamente con CDI

Al costruttore viene passato il  
generatore da usare

e assegnato alla variabile

... e poi utilizzato

Come chiamarlo

Classico esempio di *Inversion of Control*  
costruito manualmente

Il controllo di creare la dipendenza tra  
*BookService* e *NumberGenerator* è invertito  
dal momento che è dato ad una classe esterna e  
non alla classe stessa



64

## @Inject: no dependencies built by hand

- In realtà non c'è necessità di fare injection manualmente, si può lasciare questa libertà al container che lo farà per noi

65



## @Inject: no dependencies built by hand

Listing 2-4

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

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

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

Si informa il container che  
su questa proprietà dovrà  
iniettare un riferimento

Injection  
point

66

## @Inject: no dependencies built by hand

Listing 2-4

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

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

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

Si informa il container che su  
questa proprietà dovrà  
iniettare un riferimento  
... ad una implementazione  
di NumberGenerator

67

## @Inject: no dependencies built by hand

Listing 2-4

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

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

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

Si informa il container che su questa proprietà dovrà iniettare un riferimento ... ad una implementazione di NumberGenerator

Nel metodo viene usata

68

## @Inject: no dependencies built by hand

Listing 2-4

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

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

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

Si informa il container che su questa proprietà dovrà iniettare un riferimento ... ad una implementazione di NumberGenerator

Nel metodo viene usata

La definizione della classe IsbnGenerator che risponde ai requisiti: **senza alcuna specifica particolare**

69

## @Inject: no dependencies built by hand

Java EE

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

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

POJO

```
public class BookService {
    private NumberGenerator numberGenerator;

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

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

Differenza minima tra  
l'implementazione a  
POJOs e quella JEE!

70

## Punti di Injection

- L'injection può avvenire attraverso tre possibili meccanismi:
  - Su una proprietà
  - Sul costruttore
  - Sul setter
- Quando devo usare una proprietà rispetto ad un costruttore o rispetto ad un setter?  
Nessuna differenza sostanziale: solo stile di programmazione

71

## Injection di DEFAULT

- Assumiamo che `NumberGenerator` abbia una sola implementazione
  - `IsbnGenerator`
- CDI sarà in grado di fare injection semplicemente usando `@Inject`

```
@Inject
private NumberGenerator numberGenerator;
```

- Questa viene definita **default injection**

73

## Injection di DEFAULT

- Assumiamo che `NumberGenerator` abbia una sola implementazione
  - `IsbnGenerator`
- CDI sarà in grado di fare injection semplicemente usando `@Inject`

```
@Inject
private NumberGenerator numberGenerator;
```

Se si definisce un bean senza  
qualificatore questo avrà  
automaticamente il qualificatore  
`@Default`

- Questa è definita injection di default
  - Se uno specifico qualificatore non viene definito, si assume il qualificatore  
`@javax.enterprise.inject.Default`

```
@Inject @Default
private NumberGenerator numberGenerator;
```

`@Default` è un built-in qualifier che  
informa CDI di iniettare la  
implementazione di default

identici

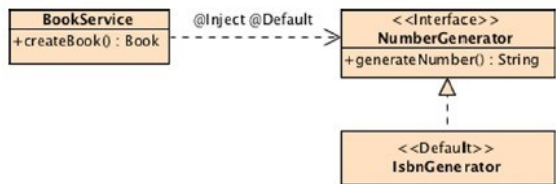
74

# Injection di DEFAULT

```
@Default
public class IsbnGenerator implements NumberGenerator {

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

Si indica il valore di default



Class diagram with @Default Injection

75

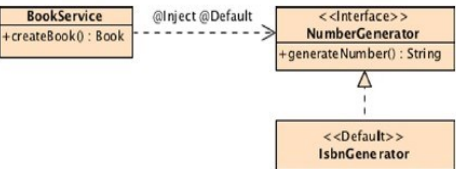
# Injection di DEFAULT

```
@Default
public class IsbnGenerator implements NumberGenerator {

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

Si indica il valore di default

Senza di esso (ed in presenza di più implementazioni), il container farebbe la injection solo se IsbnGenerator fosse l'unica implementazione esistente



Class diagram with @Default Injection

Spesso però bisogna scegliere fra diverse implementazioni

Abbiamo necessità dei Qualificatori

76

## Organizzazione della lezione

- Introduzione
  - Dependency Injection
  - Life-cycle Management
  - Interception
  - Loose Coupling and Strong Typing
  - Deployment Descriptor
- Come scrivere un CDI Bean
  - Injection
  - **Qualifiers**
  - Producers/Disposers
  - Scope
- Interceptors
  - Classi Interceptor e Ciclo di vita
  - Interceptor multipli
- Decorators e Eventi
- Conclusioni

77

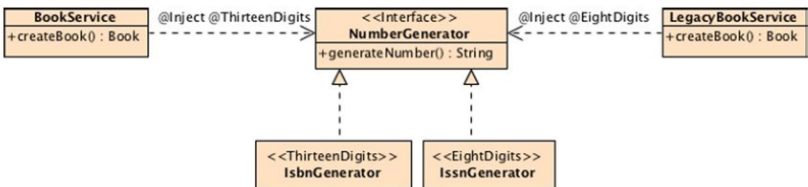
## Qualifiers

- Nella fase di inizializzazione, il container deve validare che **esattamente un bean** che **soddisfi ogni injection point** esista
- **Se non esiste nessuna implementazione** di NumberGenerator disponibile
  - Il Container impedisce il deploy e genera un **errore di dipendenza non soddisfatta**
- **Se una sola implementazione esiste**, l'injection funzionerà usando il qualificatore **@Default**
- **Se esiste più di una default implementation**
  - Il container informa che c'è una **dipendenza ambigua e non farà il deploy**
  - Questo perché la type-safe fallisce nel momento in cui il container è incapace di identificare il bean da iniettare

78

# Qualifiers

- Come si fa una componente a capire quale implementazione (IsbnGenerator oppure IssnGenerator) deve essere iniettata? Dobbiamo aiutare il container a scegliere quale usare
- Non ci si affida a file esterni XML di configurazione: type-safety! Si usano i Qualificatori!!!
- Supponiamo di avere due servizi: uno aggiornato (ISBN con 13 cifre) e uno "legacy" con ISSN a 8 cifre

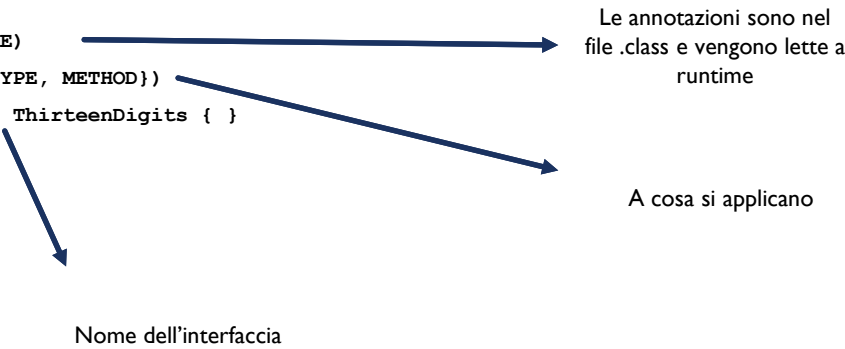


79

# Qualifiers

Listing 2-7

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



80

# Qualifiers

Listing 2-7

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

Annotazione User-defined

Listing 2-8

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

# Qualifiers

Listing 2-7

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

Annotazione User-defined

Con il nome di  
@ThirteenDigits

Listing 2-8

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



# Qualifiers

Listing 2-7

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

Annotazione User-defined  
Con il nome di  
@ThirteenDigits

Listing 2-8

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

Altra annotazione il cui nome è  
@EightDigits

# Qualifiers

Listing 2-9

```
@ThirteenDigits
public class IsbnGenerator implements NumberGenerator {
    public String generateNumber() {
        return "13-84356-" + Math.abs(new
            Random().nextInt());
    }
}
```

Il generatore ISBN annotato a  
13 cifre

Listing 2-10

```
@EightDigits
public class IssnGenerator implements NumberGenerator {
    public String generateNumber() {
        return "8-" + Math.abs(new Random().nextInt());
    }
}
```

Il generatore ISSN annotato a  
8 cifre

# BookService “Moderno” e quello “Legacy”

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

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

public class LegacyBookService {
    @Inject @EightDigits

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

Listing 2-11

L'injection point viene annotato con il qualificatore definito dall'utente per il tipo ISBN

85

# BookService “Moderno” e quello “Legacy”

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

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

public class LegacyBookService {
    @Inject @EightDigits

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

Listing 2-11

L'injection point viene annotato con il qualificatore definito dall'utente per il tipo ISBN

Listing 2-11

Per la applicazione legacy, invece si annota con il qualificatore per il tipo ISSN (a 8 cifre)

Listing 2-12

86

## Qualificatori con parametri

- Ogni volta che bisogna scegliere fra diverse implementazioni, bisogna creare un qualificatore (annotazioni)
- Supponiamo che si abbia necessità di altri due number generators (per numeri a 2 e 10 cifre)
- Avremo pertanto le seguenti annotazioni:
  - @TwoDigits
  - @EightDigits
  - @TenDigits
  - @ThirteenDigits

87

## Qualificatori con parametri

- Supponiamo inoltre che i numeri generati possano essere pari o dispari
- Avremo pertanto le seguenti annotazioni:
  - @TwoOddDigits
  - @TwoEvenDigits
  - @EightOddDigits
  - @EightEvenDigits
  - @TenOddDigits
  - @TenEvenDigits
  - @ThirteenOddDigits
  - @ThirteenEvenDigits
- Un modo per evitare la moltiplicazione delle annotazioni: usare i [members](#)

88

## Qualificatori con parametri

**@NumberOfDigits with a  
Digits Enum and a Parity  
Boolean**

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface NumberOfDigits{
    Digits value();
    boolean odd();
}
public enum Digits{
    TWO,
    EIGHT,
    TEN,
    THIRTEEN
}
```

Un qualificatore con parametri

89

## Qualificatori con parametri

**@NumberOfDigits with a  
Digits Enum and a Parity  
Boolean**

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface NumberOfDigits{
    Digits value();
    boolean odd();
}
public enum Digits{
    TWO,
    EIGHT,
    TEN,
    THIRTEEN
}
```

Un qualificatore con parametri

A cosa si può applicare

90

# Qualificatori con parametri

@NumberOfDigits with a  
Digits Enum and a Parity  
Boolean

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface NumberOfDigits {
    Digits value();
    boolean odd();
}
public enum Digits {
    TWO,
    EIGHT,
    TEN,
    THIRTEEN
}
```

Un qualificatore con parametri  
A cosa si può applicare  
Il nome del qualificatore

91

# Qualificatori con parametri

@NumberOfDigits with a  
Digits Enum and a Parity  
Boolean

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface NumberOfDigits{
    Digits value();
    boolean odd();
}
public enum Digits{
    TWO,
    EIGHT,
    TEN,
    THIRTEEN
}
```

Un qualificatore con parametri  
A cosa si può applicare  
Il nome del qualificatore  
Il primo parametro (un enum)

92

# Qualificatori con parametri

@NumberOfDigits with a  
Digits Enum and a Parity  
Boolean

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface NumberOfDigits {
    Digits value();
    boolean odd();
}
public enum Digits {
    TWO,
    EIGHT,
    TEN,
    THIRTEEN
}
```

- Un qualificatore con parametri
- A cosa si può applicare
- Il nome del qualificatore
- Il primo parametro (un enum)
- Il secondo parametro (numero pari/dispari)

# Qualificatori con parametri

@NumberOfDigits with a  
Digits Enum and a Parity  
Boolean

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface NumberOfDigits {
    Digits value();
    boolean odd();
}
public enum Digits {
    TWO,
    EIGHT,
    TEN,
    THIRTEEN
}
```

- Un qualificatore con parametri
- A cosa si può applicare
- Il nome del qualificatore
- Il primo parametro (un enum)
- Il secondo parametro (numero pari/dispari)
- La definizione delle cifre per enumerazione

# Qualificatori con parametri

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface NumberOfDigits {
    Digits value();
    boolean odd();
}

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

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

- Un qualificatore con parametri
- A cosa si può applicare
- Il nome del qualificatore
- Il primo parametro (un enum)
- Il secondo parametro (numero pari/dispari)
- La definizione delle cifre per enumerazione
- Come si usa

95

# Qualificatori con parametri

@NumberOfDigits with a  
Digits Enum and a Parity  
Boolean

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface NumberOfDigits {
    Digits value();
    boolean odd();
}

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

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

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

- Un qualificatore con parametri
- A cosa si può applicare
- Il nome del qualificatore
- Il primo parametro (un enum)
- Il secondo parametro (numero pari/dispari)
- La definizione delle cifre per enumerazione
- Come si usa
- L'implementazione

96

## Alternative

- Invece di scegliere tra diverse implementazioni della stessa interfaccia...
- Può essere necessario dover scegliere una implementazione completamente alternativa, dipendente dallo scenario di deployment
- Esempio tipico: una classe di mock-up per le attività di unit-testing

```
@Alternative
public class MockGenerator implements NumberGenerator {
    public String generateNumber() {
        return "MOCK";
    }
}
```

Annotazione per denotare una implementazione alternativa

97

## Alternative

- Invece di scegliere tra diverse implementazioni della stessa interfaccia...
- Può essere necessario dover scegliere una implementazione completamente alternativa, dipendente dallo scenario di deployment
- Esempio tipico: una classe di mock-up per le attività di unit-testing

```
@Alternative
public class MockGenerator implements NumberGenerator {
    public String generateNumber() {
        return "MOCK";
    }
}
```

Annotazione per denotare una implementazione alternativa

ID fittizia restituita

98



## Il file beans.xml

- File necessario per far capire che il container deve applicare CDI
  - un CDI bean è definito come un POJO che è in un archivio che contiene un file beans.xml
- In questo caso lo usiamo per segnalare che abbiamo delle alternative

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
  "http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/beans 1 1.xsd"
  version="1.1" bean-discovery-mode="all">

  <alternatives>
    <class>
      org.agoncal.book.javaee7.chapter02.MockGenerator
    </class>
  </alternatives>

</beans>
```

Namespace

99

## Il file beans.xml

- File necessario per far capire che il container deve applicare CDI
  - un CDI bean è definito come un POJO che è in un archivio che contiene un file beans.xml
- In questo caso lo usiamo per segnalare che abbiamo delle alternative

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
  "http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/beans 1 1.xsd"
  version="1.1" bean-discovery-mode="all">

  <alternatives>
    <class>
      org.agoncal.book.javaee7.chapter02.MockGenerator
    </class>
  </alternatives>

</beans>
```

Namespace

Indicazione dell' XML Schema usato per la validazione

100

# Il file beans.xml

- File necessario per far capire che il container deve applicare CDI
  - un CDI bean è definito come un POJO che è in un archivio che contiene un file beans.xml
- In questo caso lo usiamo per segnalare che abbiamo delle alternative

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
"http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans 1 1.xsd"
version="1.1" bean-discovery-mode="all">

<alternatives>
<class>
org.agoncal.book.javaee7.chapter02.MockGenerator
</class>
</alternatives>

</beans>
```

Namespace

Indicazione dell' XML Schema  
usato per la validazione

Attenzione! "all"! Altrimenti prende  
solamente quelli "named" (stay  
tuned!) (non considera tutti i bean ma  
solo quelli con annotated scope)

101

# Il file beans.xml

- File necessario per far capire che il container deve applicare CDI
  - un CDI bean è definito come un POJO che è in un archivio che contiene un file beans.xml
- In questo caso lo usiamo per segnalare che abbiamo delle alternative

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
"http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans 1 1.xsd"
version="1.1" bean-discovery-mode="all">

<alternatives>
<class>
org.agoncal.book.javaee7.chapter02.MockGenerator
</class>
</alternatives>

</beans>
```

Namespace

Indicazione dell' XML Schema  
usato per la validazione

Attenzione! "all"! Altrimenti prende  
solamente quelli "named" (stay  
tuned!) (non considera tutti i bean ma  
solo quelli con annotated scope)

La alternativa

102

## Organizzazione della lezione

- Introduzione
  - Dependency Injection
  - Life-cycle Management
  - Interception
  - Loose Coupling and Strong Typing
  - Deployment Descriptor
- Come scrivere un CDI Bean
  - Injection
  - Qualifiers
  - Producers/Disposers
  - Scope
- Interceptors
  - Classi Interceptor e Ciclo di vita
  - Interceptor multipli
- Decorators e Eventi
- Conclusioni

103

## Cosa si può iniettare in un CDI Bean?

- Per il container, un CDI Bean è un qualsiasi POJO messo in un package con il file `beans.xml`
- Finora abbiamo visto che si può iniettare un CDI Bean in un altro CDI Bean
  - facile: entrambi gestiti dal container!
- Come si fa a iniettare una classe tipo `java.util.Date` o `java.lang.String`?
  - Il problema: queste classi stanno nel file `rt.jar` (java runtime environment classes) che **non** contiene il file `beans.xml`
- Non possiamo modificare `rt.jar`: **Si devono usare i producer!**

104

## I Producer

- Aggiungiamo una classe che produce valori, annotati con `javax.enterprise.inject-Produces`
- Significa che ora questi valori possono essere iniettati

```
public class NumberProducer {
    @Produces @ThirteenDigits
    private String prefix13digits = "13-";

    @Produces @ThirteenDigits
    private int editorNumber = 4356;

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

Nome della classe

105

## I Producer

- Aggiungiamo una classe che produce valori, annotati con `javax.enterprise.inject-Produces`
- Significa che ora questi valori possono essere iniettati

```
public class NumberProducer {
    @Produces @ThirteenDigits
    private String prefix13digits = "13-";

    @Produces @ThirteenDigits
    private int editorNumber = 4356;

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

Nome della classe

Definizione di una stringa

106

# I Producer

- Aggiungiamo una classe che produce valori, annotati con `javax.enterprise.inject-Produces`
- Significa che ora questi valori possono essere iniettati

```
public class NumberProducer {
    @Produces @ThirteenDigits
    private String prefix13digits = "13-";

    @Produces @ThirteenDigits
    private int editorNumber = 4356;

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

Nome della classe

Definizione di una stringa

Definizione di un intero

107

# I Producer

- Aggiungiamo una classe che produce valori, annotati con `javax.enterprise.inject-Produces`
- Significa che ora questi valori possono essere iniettati

```
public class NumberProducer {
    @Produces @ThirteenDigits
    private String prefix13digits = "13-";

    @Produces @ThirteenDigits
    private int editorNumber = 4356;

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

Nome della classe

Definizione di una stringa

Definizione di un intero

Definizione di un metodo

108

# I Producer

- Aggiungiamo una classe che produce valori, annotati con `javax.enterprise.inject.Produces`
- Significa che ora questi valori possono essere iniettati

```
public class NumberProducer {
    @Produces @ThirteenDigits
    private String prefix13digits = "13-";

    @Produces @ThirteenDigits
    private int editorNumber = 4356;

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

Nome della classe

Definizione di una stringa

Definizione di un intero

Definizione di un metodo

Il valore restituito dal metodo viene iniettato

109

# Come usare i Producer

```
public class IsbnGenerator implements NumberGenerator {
    @Inject @ThirteenDigits
    private String prefix;

    @Inject @ThirteenDigits
    private int editorNumber;

    @Inject @Random
    private double postfix;

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

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

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

Nella classe ISBN (già un CDI Bean che viene iniettato per BookService)

110

# Come usare i Producer

```
public class IsbnGenerator implements NumberGenerator {
    @Inject @ThirteenDigits
    private String prefix;

    @Inject @ThirteenDigits
    private int editorNumber;

    @Inject @Random
    private double postfix;

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

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

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

Nella classe ISBN (già un CDI Bean che viene iniettato per BookService)

Viene iniettata una stringa di prefisso

111

# Come usare i Producer

```
public class IsbnGenerator implements NumberGenerator {
    @Inject @ThirteenDigits
    private String prefix;

    @Inject @ThirteenDigits
    private int editorNumber;

    @Inject @Random
    private double postfix;

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

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

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

Nella classe ISBN (già un CDI Bean che viene iniettato per BookService)

Viene iniettata una stringa di prefisso

... e un numero di editore

112

## Come usare i Producer

```
public class IsbnGenerator implements NumberGenerator {
    @Inject @ThirteenDigits
    private String prefix;

    @Inject @ThirteenDigits
    private int editorNumber;

    @Inject @Random
    private double postfix;

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

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

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

Nella classe ISBN (già un CDI Bean che viene iniettato per BookService)

Viene iniettata una stringa di prefisso

... e un numero di editore

... e un valore random

113

## Come usare i Producer

```
public class IsbnGenerator implements NumberGenerator {
    @Inject @ThirteenDigits
    private String prefix;

    @Inject @ThirteenDigits
    private int editorNumber;

    @Inject @Random
    private double postfix;

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

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

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

Nella classe ISBN (già un CDI Bean che viene iniettato per BookService)

Viene iniettata una stringa di prefisso

... e un numero di editore

... e un valore random

... in modo che il numero generato è completamente dipendente da essi, ma debolmente accoppiato (a deploy-time)

114



## Come usare i Producer

```
public class IsbnGenerator implements NumberGenerator {
    @Inject @ThirteenDigits
    private String prefix;

    @Inject @ThirteenDigits
    private int editorNumber;

    @Inject @Random
    private double postfix;

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

### Strong typing in action:

Usando la sintassi `@Inject @ThirteenDigits`,  
CDI sa che bisogna iniettare una stringa, un  
intero o un'implementazione di  
`NumberGenerator`

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

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

Nella classe ISBN (già un CDI Bean che  
viene iniettato per `BookService`)

Viene iniettata una stringa di prefisso

... e un numero di editore

... e un valore random

... in modo che il numero generato è  
completamente dipendente da essi, ma  
debolmente accoppiato (a deploy-time)

**Importante la type safety!**

115

## I Disposers

- Nell'esempio precedente, i producers sono stati usati per creare tipi di dato o POJO da far iniettare: non avevamo bisogno di eliminarli
- In alcuni casi (JDBC connection, JMS session, ...) è necessario prevedere esplicitamente la distruzione degli oggetti
- Un disposer permette di realizzare una pulizia "**personalizzata**" di un oggetto che era stato restituito da un metodo di producer

116

# Un esempio di Disposer

```
public class JDBCConnectionProducer {  
  
    @Produces  
    private Connection createConnection() {  
        Connection conn = null;  
        try{  
            Class.forName(  
                "org.apache.derby.jdbc.EmbeddedDriver").newInstance();  
            conn = DriverManager.getConnection(  
                "jdbc:derby:memory:chapter02DB","APP",  
                "APP");  
        }catch(InstantiationException |  
            IllegalAccessException |  
            ClassNotFoundException ) {  
            e.printStackTrace();  
        }  
        return conn;  
    }  
  
    private void closeConnection(@Disposes Connection conn)  
        throws SQLException {  
        conn.close();  
    }  
}
```

La classe produce una  
connessione da far utilizzare

Creata e chiusa una connessione JDBC

117

# Un esempio di Disposer

```
public class JDBCConnectionProducer {  
  
    @Produces  
    private Connection createConnection() {  
        Connection conn = null;  
        try{  
            Class.forName(  
                "org.apache.derby.jdbc.EmbeddedDriver").newInstance();  
            conn = DriverManager.getConnection(  
                "jdbc:derby:memory:chapter02DB","APP",  
                "APP");  
        }catch(InstantiationException |  
            IllegalAccessException |  
            ClassNotFoundException ) {  
            e.printStackTrace();  
        }  
        return conn;  
    }  
  
    private void closeConnection(@Disposes Connection conn)  
        throws SQLException {  
        conn.close();  
    }  
}
```

La classe produce una  
connessione da far utilizzare

Un metodo, il cui parametro è annotato  
con @Disposes viene chiamato quando il  
contesto ("scope") del client termina

Creata e chiusa una connessione JDBC

118

## Un esempio di Disposer

```
public class JDBCConnectionProducer {

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

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

La classe produce una connessione da far utilizzare

Un metodo, il cui parametro è annotato con `@Disposes` viene chiamato quando il contesto ("scope") del client termina

In il metodo `closeConnection()` termina la connessione JDBC

Creata e chiusa una connessione JDBC

119

## Organizzazione della lezione

- Introduzione
  - Dependency Injection
  - Life-cycle Management
  - Interception
  - Loose Coupling and Strong Typing
  - Deployment Descriptor
- Come scrivere un CDI Bean
  - Injection
  - Qualifiers
  - Producers/Disposers
  - Scope
- Interceptors
  - Classi Interceptor e Ciclo di vita
  - Interceptor multipli
- Decorators e Eventi
- Conclusioni

120

## CDI: la C di Context

- Ogni oggetto gestito da CDI ha un ambito (context) ben definito
  - ... ed un ciclo di vita che è legato all'ambito
- In Java lo scope di un POJO è semplice: si crea una istanza di una classe con la keyword **new** e ci si appoggia al Garbage Collection per liberare memoria
- In CDI un bean è legato ad un contesto e rimane attivo in questo contesto fino a quando il container non lo distrugge
- Tipi di scope:
  - Application scope
  - Session scope
  - Request scope
  - Conversation scope
  - Dependent pseudo-scope

121

## Scope: Application

- Application scope (**@ApplicationScoped**):
  - Si estende per l'intera durata di una applicazione
  - Il bean viene creato solo una volta per la durata dell'applicazione e viene distrutto quando l'applicazione termina
  - Vantaggi: utile per utility o classi helper, oppure per oggetti che memorizzano dati condivisi dall'intera applicazione
  - Svantaggi: problemi di efficienza, concorrenza (dati acceduti da diversi threads)

122

## Scope: Session

- Session scope (**@SessionScoped**):
  - Si estende attraverso diverse richieste HTTP o diverse invocazioni identificate come una singola sessione utente
  - Il bean viene creato per la durata di una sessione HTTP e viene distrutto quando la sessione termina
  - Vantaggi: utile per oggetti che sono necessari per la durata di una sessione, ad esempio, user preferences o login credentials

123

## Scope: Request

- Request scope (**@RequestScoped**):
  - Singola richiesta HTTP o invocazione di metodo
  - Il bean viene creato per la durata di una invocazione di un metodo e viene distrutto quando il metodo termina

124

## Scope: Conversation

- Conversation scope (**@ConversationScoped**)
  - Si estende attraverso invocazioni multiple all'interno di sessioni con punti di partenza e fine identificate dall'applicazione

125

## Scope: Dependent

- Dependent pseudo-scope (**@Dependent**):
  - E' semplicemente il default scope per CDI

126

## Scope: esempio di sessione

- Una istanza di ShoppingCart è legata ad una sessione utente ed è condivisa da tutte le richieste nel contesto di questa sessione

```
@SessionScoped  
public class ShoppingCart implements Serializable {...}
```

127

## Scope: esempio di conversazione

- Rispetto agli altri è diverso, in quanto programmabile
- Qui è l'applicazione stessa che definisce quella che è una "conversazione"
- La conversazione può essere composta di diverse richieste, ed ha un inizio/fine ben definito (ad esempio un wizard)
- In un certo senso, si trova tra la @RequestScoped e la @SessionScoped

**Request**  
Singola richiesta HTTP/  
invocazione

**Conversation  
Scope**

**Session**  
Diverse richieste HTTP/  
invocazioni

128

## Scope: esempio di conversazione

### Customer creation wizard web application

- Il wizard si compone di tre passi:
  - Il cliente inserisce login information
    - Username e password
  - Il cliente aggiunge dettagli nell'account
    - Nome, cognome, indirizzo email
  - Conferma delle informazioni raccolte e creazione dell'account

129

## Un wizard @ConversationScoped

```
@ConversationScoped
public class CustomerCreatorWizard implements Serializable
{
    private Login login;
    private Account account;
    @Inject
    private CustomerService customerService;
    @Inject
    private Conversation conversation;
    public void saveLogin() {
        conversation.begin();
        login = new Login();
        //Sets login properties
    }
    public void saveAccount() {
        account = new Account();
        //Sets account properties
    }
    public void createCustomer() {
        Customer customer = new Customer();
        customer.setLogin(login);
        customer.setAccount(account);
        customerService.createCustomer(customer);
        conversation.end();
    }
}
```

Scope di Conversazione

130



## Un wizard @ConversationScoped

```
@ConversationScoped
public class CustomerCreatorWizard implements Serializable
{
    private Login login;
    private Account account;
    @Inject
    private CustomerService customerService;
    @Inject
    private Conversation conversation;
    public void saveLogin() {
        conversation.begin();
        login = new Login();
        //Sets login properties
    }
    public void saveAccount() {
        account = new Account();
        //Sets account properties
    }
    public void createCustomer() {
        Customer customer = new Customer();
        customer.setLogin(login);
        customer.setAccount(account);
        customerService.createCustomer(customer);
        conversation.end();
    }
}
```

Scope di Conversazione

Iniezione di CustomerService per creare un Customer

131

## Un wizard @ConversationScoped

```
@ConversationScoped
public class CustomerCreatorWizard implements Serializable
{
    private Login login;
    private Account account;
    @Inject
    private CustomerService customerService;
    @Inject
    private Conversation conversation;
    public void saveLogin() {
        conversation.begin();
        login = new Login();
        //Sets login properties
    }
    public void saveAccount() {
        account = new Account();
        //Sets account properties
    }
    public void createCustomer() {
        Customer customer = new Customer();
        customer.setLogin(login);
        customer.setAccount(account);
        customerService.createCustomer(customer);
        conversation.end();
    }
}
```

Scope di Conversazione

Iniezione di CustomerService per creare un Customer

...e di una conversazione

132

## Un wizard @ConversationScoped

```
@ConversationScoped
public class CustomerCreatorWizard implements Serializable
{
    private Login login;
    private Account account;
    @Inject
    private CustomerService customerService;
    @Inject
    private Conversation conversation;
    public void saveLogin() {
        conversation.begin();
        login = new Login();
        //Sets login properties
    }
    public void saveAccount() {
        account = new Account();
        //Sets account properties
    }
    public void createCustomer() {
        Customer customer = new Customer();
        customer.setLogin(login);
        customer.setAccount(account);
        customerService.createCustomer(customer);
        conversation.end();
    }
}
```

Scope di Conversazione

Iniezione di CustomerService per creare un Customer

... e di una conversazione

Segnala l'inizio della conversazione  
(all'invocazione del metodo saveLogin())

133

## Un wizard @ConversationScoped

```
@ConversationScoped
public class CustomerCreatorWizard implements Serializable
{
    private Login login;
    private Account account;
    @Inject
    private CustomerService customerService;
    @Inject
    private Conversation conversation;
    public void saveLogin() {
        conversation.begin();
        login = new Login();
        //Sets login properties
    }
    public void saveAccount() {
        account = new Account();
        //Sets account properties
    }
    public void createCustomer() {
        Customer customer = new Customer();
        customer.setLogin(login);
        customer.setAccount(account);
        customerService.createCustomer(customer);
        conversation.end();
    }
}
```

Scope di Conversazione

Iniezione di CustomerService per creare un Customer

... e di una conversazione

Segnala l'inizio della conversazione  
(all'invocazione del metodo saveLogin())

Dopo la creazione del Customer...

La conversazione termina

(conversation.end())

134

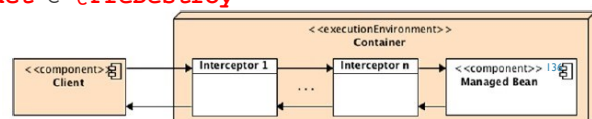
## Organizzazione della lezione

- Introduzione
  - Dependency Injection
  - Life-cycle Management
  - Interception
  - Loose Coupling and Strong Typing
  - Deployment Descriptor
- Come scrivere un CDI Bean
  - Injection
  - Qualifiers
  - Producers/Disposers
  - Scope
- Interceptors
  - Classi Interceptor e Ciclo di vita
  - Interceptor multipli
- Decorators e Eventi
- Conclusioni

135

## Gli Interceptors

- Permettono di aggiungere funzionalità cross-cutting ai bean CDI
- Quando un metodo viene invocato da un client su un Managed Bean (CDI Bean, EJB, RESTful Web Service, ...) allora la chiamata viene intercettata dal container e vengono inseriti gli interceptor
- Quattro tipi di interceptor:
  - Associati ad un costruttore della classe target: **@AroundConstruct**
  - Associati ad uno specifico business method: **@AroundInvoke**
  - Di timeout: **@AroundTimeout**
  - Call-back del ciclo di vita: **@PostConstruct** e **@PreDestroy**



136

## Gli Interceptors

- Come si definiscono gli interceptor?
- Il modo più semplice:
  - aggiungendoli al bean stesso

137

## Gli Interceptors

- Vediamo il seguente esempio:
  - La classe **CustomerService** annota **logMethod()** con **@AroundInvoke**
  - Il metodo **logMethod()** fa logging di un messaggio in entrata ed in uscita da un metodo
  - Una volta fatto il deploy del Managed Bean, ogni invocazione client per **createCustomer()** o per **findCustomerById()** sarà intercettata ed il metodo **logMethod()** verrà applicato
  - Lo scope di questo interceptor è limitato al bean stesso (la target class)

138

## Come si definiscono gli Interceptors

La maniera più semplice annotando metodi al bean stesso: target class interceptor

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;
    @Inject
    private Logger logger;
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }
    public Customer findCustomerById(Long id) {
        return em.find(Customer.class, id);
    }
}

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

Tutti i metodi sono transazioni

139

## Come si definiscono gli Interceptors

La maniera più semplice annotando metodi al bean stesso: target class interceptor

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;
    @Inject
    private Logger logger;
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }
    public Customer findCustomerById(Long id) {
        return em.find(Customer.class, id);
    }
}

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

Tutti i metodi sono transazioni  
Valore iniettato

140

# Come si definiscono gli Interceptors

La maniera più semplice annotando metodi al bean stesso: target class interceptor

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;
    @Inject
    private Logger logger;
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }
    public Customer findCustomerById(Long id) {
        return em.find(Customer.class, id);
    }
}

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

Tutti i metodi sono transazioni  
Valore iniettato  
Valore iniettato

141

# Come si definiscono gli Interceptors

La maniera più semplice annotando metodi al bean stesso: target class interceptor

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;
    @Inject
    private Logger logger;
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }
    public Customer findCustomerById(Long id) {
        return em.find(Customer.class, id);
    }
}

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

Tutti i metodi sono transazioni  
Valore iniettato  
Valore iniettato  
Metodo di business I

142

# Come si definiscono gli Interceptors

La maniera più semplice annotando metodi al bean stesso: target class interceptor

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;
    @Inject
    private Logger logger;
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }
    public Customer findCustomerId(Long id) {
        return em.find(Customer.class, id);
    }

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

Tutti i metodi sono transazioni  
Valore iniettato  
Valore iniettato  
Metodo di business 1  
Metodo di business 2

143

# Come si definiscono gli Interceptors

La maniera più semplice annotando metodi al bean stesso: target class interceptor

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;
    @Inject
    private Logger logger;
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }
    public Customer findCustomerId(Long id) {
        return em.find(Customer.class, id);
    }

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

Tutti i metodi sono transazioni  
Valore iniettato  
Valore iniettato  
Metodo di business 1  
Metodo di business 2  
Metodo interceptor

144

# Come si definiscono gli Interceptors

La maniera più semplice annotando metodi al bean stesso: target class interceptor

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;
    @Inject
    private Logger logger;
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }
    public Customer findCustomerById(Long id) {
        return em.find(Customer.class, id);
    }
}

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

- Tutti i metodi sono transazioni
- Valore iniettato
- Valore iniettato
- Metodo di business 1
- Metodo di business 2
- Metodo interceptor
- Contesto, iniettato dal container

145

# Come si definiscono gli Interceptors

La maniera più semplice annotando metodi al bean stesso: target class interceptor

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;
    @Inject
    private Logger logger;
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }
    public Customer findCustomerById(Long id) {
        return em.find(Customer.class, id);
    }
}

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

- Tutti i metodi sono transazioni
- Valore iniettato
- Valore iniettato
- Metodo di business 1
- Metodo di business 2
- Metodo interceptor
- Contesto, iniettato dal container
- Uso del contesto per fare il log in entrata usando l'istanza e il nome del metodo
- Il container invoca prima il logMethod()
- Logging del nome dell'invoked bean e dell'invoked method

146



## Come si definiscono gli Interceptors

La maniera più semplice annotando metodi al bean stesso: target class interceptor

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;
    @Inject
    private Logger logger;
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }
    public Customer findCustomerById(Long id) {
        return em.find(Customer.class, id);
    }
}

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

Tutti i metodi sono transazioni  
Valore iniettato  
Valore iniettato  
Metodo di business 1  
Metodo di business 2  
Metodo interceptor  
Contesto, iniettato dal container  
Uso del contesto per fare il log in  
entrata usando l'istanza e il metodo

Il metodo `proceed()` dice al  
container di procedere con  
l'interceptor successivo oppure con  
il business method

147

## Come si definiscono gli Interceptors

La maniera più semplice annotando metodi al bean stesso: target class interceptor

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;
    @Inject
    private Logger logger;
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }
    public Customer findCustomerById(Long id) {
        return em.find(Customer.class, id);
    }
}

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

Tutti i metodi sono transazioni  
Valore iniettato  
Valore iniettato  
Metodo di business 1  
Metodo di business 2  
Metodo interceptor  
Contesto, iniettato dal container  
Uso del contesto per fare il  
log in entrata usando  
l'istanza e il metodo  
Il metodo `proceed()` dice al container di  
procedere con l'interceptor successivo oppure  
con il business method  
Invocato alla fine del metodo `createCustomer()`  
Quando termina, l'interceptor finisce la sua  
esecuzione loggando un exit message  
(`logger.exiting()`)

Stesso discorso per entrambi i business methods

148

# Per poter essere un Interceptor

- Il metodo deve avere questa firma:

```
@AroundInvoke
private Object <METHOD>(InvocationContext ic) throws Exception {
```
- Il metodo può avere tutti i modificatori di accesso (public, private, protected) tranne static e final
- Deve avere il contesto come parametro e restituire un Object
- Può lanciare eccezioni
- Il contesto passato permette di concatenare diversi interceptor in modo che il contesto sia passato da uno all'altro, aggiungendo se necessario delle informazioni

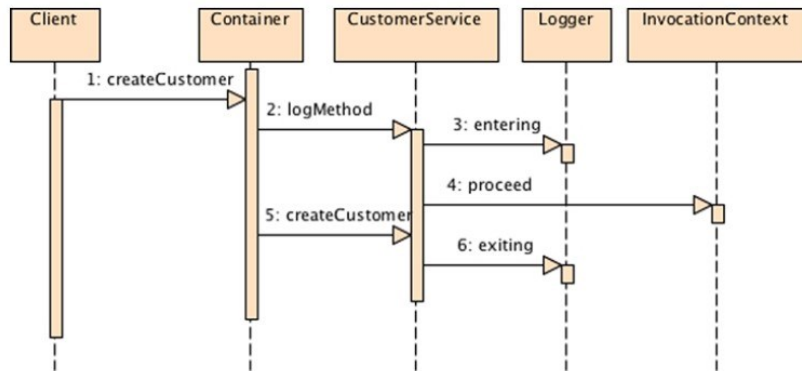
149

# I metodi di InvocationContext

Method	Description
getContextData	Allows values to be passed between interceptor methods in the same InvocationContext instance using a Map.
getConstructor	Returns the constructor of the target class for which the interceptor was invoked.
getMethod	Returns the method of the bean class for which the interceptor was invoked.
getParameters	Returns the parameters that will be used to invoke the business method.
getTarget	Returns the bean instance that the intercepted method belongs to.
getTimer	Returns the timer associated with a @Timeout method.
proceed	Causes the invocation of the next interceptor method in the chain. It returns the result of the next method invoked. If a method is of type void, proceed returns null.
setParameters	Modifies the value of the parameters used for the target class method invocation. The types and the number of parameters must match the bean's method signature, or IllegalArgumentException is thrown.

150

## Il risultato dell'Interceptor



151

## Organizzazione della lezione

- Introduzione
  - Dependency Injection
  - Life-cycle Management
  - Interception
  - Loose Coupling and Strong Typing
  - Deployment Descriptor
- Come scrivere un CDI Bean
  - Injection
  - Qualifiers
  - Producers/Disposers
  - Scope
- Interceptors
  - Classi Interceptor e Ciclo di vita
  - Interceptor multipli
- Decorators e Eventi
- Conclusioni

152

## Come si definiscono gli Interceptor

- L'esempio che abbiamo visto (Listing 2-23) definisce un interceptor disponibile soltanto per **CustomerService**
- **La maggior parte delle volte si vuole isolare i cross-cutting concerns in una classe separata e usarla in maniera trasversale per diversi bean**
- Il Logging è un tipico esempio di servizio da fornire in maniera trasversale
- Per definire una classe interceptor, bisogna sviluppare una classe separata ed istruire il container ad applicarla su uno specific bean o su uno specifico metodo
- Esempio: prendiamo il metodo **logMethod()** e lo isoliamo in una classe separata (**LoggingInterceptor**)
  - Aggiungendo anche una seconda funzionalità di logging

153

## Class Interceptors

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

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

In questa classe ...

154

## Class Interceptors

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

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

In questa classe ...

... fattorizziamo il metodo di log

155

## Class Interceptors

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

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

In questa classe ...

... fattorizziamo il metodo di log

Si chiede di iniettare il logger

156

## Class Interceptors

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

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

In questa classe ...

...fattorizziamo il metodo di log

Si chiede di iniettare il logger

Il metodo `init ()` annotato con `@AroundConstruct` viene invocato solo quando il costruttore sul bean è chiamato

157

## Class Interceptors

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

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

In questa classe ...

...fattorizziamo il metodo di log

Si chiede di iniettare il logger

Il metodo `init ()` annotato con `@AroundConstruct` viene invocato solo quando il costruttore sul bean è chiamato

...si fa il log della creazione ... dopodiché `proceed()`

158

## Class Interceptors

- Ora questa classe può essere utilizzata da ogni bean interessato a questo interceptor
- I bean interessati devono informare il container e lo fanno dichiarando l'annotazione `@Interceptor`

159

## Come si usa una classe Interceptor (su metodo)

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;
    @Interceptors(LoggingInterceptor.class)
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }
    public Customer findCustomerById(Long id) {
        return em.find(Customer.class, id);
    }
}
```

Valore iniettato

160

# Come si usa una classe Interceptor (su metodo)

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;
    @Interceptors(LoggingInterceptor.class)
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }
    public Customer findCustomerById(Long id) {
        return em.find(Customer.class, id);
    }
}
```

Valore iniettato  
Dichiarazione dell'interceptor da applicare

161

# Come si usa una classe Interceptor (su metodo)

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;
    @Interceptors(LoggingInterceptor.class)
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }
    public Customer findCustomerById(Long id) {
        return em.find(Customer.class, id);
    }
}
```

Valore iniettato  
Dichiarazione dell'interceptor da applicare  
... a questo metodo

162



## Come si usa una classe Interceptor (su metodo)

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;
    @Interceptors(LoggingInterceptor.class)
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }
    public Customer findCustomerId(Long id) {
        return em.find(Customer.class, id);
    }
}
```

Valore iniettato

Dichiarazione dell'interceptor da applicare

... a **questo** metodo

Questo non viene intercettato

E se vogliamo intercettare entrambi i metodi?

163

## Come si usa una classe Interceptor

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;
    @Interceptors(LoggingInterceptor.class)
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }
    public Customer findCustomerId(Long id) {
        return em.find(Customer.class, id);
    }
}
```

Valore iniettato

Dichiarazione dell'interceptor da applicare

...

... a **questo** metodo

Questo non viene intercettato

Se si vuole l'interceptor su tutti i metodi si può aggiungere **@Interceptors** su tutto il bean

```
@Transactional
@Interceptors(LoggingInterceptor.class)
public class CustomerService {
    @Inject
    private EntityManager em;
    public void createCustomer(Customer customer) {}
    public Customer findCustomerId(Long id) {}
}
```

164

## Come si usa una classe Interceptor

```
@Transactional
@Interceptors(LoggingInterceptor.class)
public class CustomerService {

    public void createCustomer(Customer customer) {}
    public Customer findCustomerId(Long id) {}

    @ExcludeClassInterceptors
    public Customer updateCustomer(Customer customer) {}
}
```

Se si vuole applicare l'interceptor ad un intero bean ad esclusione di un metodo si può usare l'annotazione:

`javax.interceptor.ExcludeClassInterceptors`

165

## Come si definiscono gli Interceptor Life-Cycle Interceptors

- Con una annotazione callback si può informare il container di invocare un metodo in una particolare fase (**life-cycle phase**)
  - @PostConstruct e @PreDestroy
  - Ad esempio, se si vuol fare logging ogni volta che una istanza di un bean viene creata, è sufficiente aggiungere l'annotazione @PostConstruct su un metodo del bean e ad aggiungere ad esso un qualche meccanismo di logging

166

## Come si definiscono gli Interceptor Life-Cycle Interceptors

- **Ma se si vogliono catturare life-cycle events tra diversi tipi di beans?**
- Soluzione:
  - I Life-cycle Interceptors permettono di isolare una porzione di codice in una classe ed invocarla quando un evento viene intercettato
  - Esempio tipico: un profiler che monitora creazione e risorse usate
- Vediamo il seguente esempio:
  - Classe ProfileInterceptor (Listing 2-26 ) con 2 metodi:
    - logMethod(): usato per postconstruction (@PostConstruct)
    - profile(): usato per method interception (@AroundInvoke)

167

## Come si definiscono gli Interceptor Life-Cycle Interceptors

Listing 2-26

```
public class ProfileInterceptor {
    @Inject
    private Logger logger;
    @PostConstruct
    public void logMethod(InvocationContext ic) throws
        Exception {
        logger.fine(ic.getTarget().toString());
        try {
            ic.proceed();
        } finally {
            logger.fine(ic.getTarget().toString());
        }
    }
    @AroundInvoke
    public Object profile(InvocationContext ic) throws
        Exception {
        long initTime = System.currentTimeMillis();
        try {
            return ic.proceed();
        } finally {
            long diffTime = System.currentTimeMillis() - initTime;
            logger.fine(ic.getMethod()+"."+diffTime+" millis");
        }
    }
}
```

Il logger viene iniettato

168

# Come si definiscono gli Interceptor

## Life-Cycle Interceptors

Listing 2-26

```
public class ProfileInterceptor {
    @Inject
    private Logger logger;
    @PostConstruct
    public void logMethod(InvocationContext ic) throws
        Exception {
        logger.fine(ic.getTarget().toString());
        try {
            ic.proceed();
        } finally {
            logger.fine(ic.getTarget().toString());
        }
    }
    @AroundInvoke
    public Object profile(InvocationContext ic) throws
        Exception {
        long initTime = System.currentTimeMillis();
        try {
            return ic.proceed();
        } finally {
            long diffTime = System.currentTimeMillis() - initTime;
            logger.fine(ic.getMethod()+"."+diffTime+" millis");
        }
    }
}
```

Il logger viene iniettato

logMethod() usato per postconstruction  
@PostConstruct.  
Prende in input un InvocationContext restituisce  
void

169

# Come si definiscono gli Interceptor

## Life-Cycle Interceptors

Listing 2-26

```
public class ProfileInterceptor {
    @Inject
    private Logger logger;
    @PostConstruct
    public void logMethod(InvocationContext ic) throws
        Exception {
        logger.fine(ic.getTarget().toString());
        try {
            ic.proceed();
        } finally {
            logger.fine(ic.getTarget().toString());
        }
    }
    @AroundInvoke
    public Object profile(InvocationContext ic) throws
        Exception {
        long initTime = System.currentTimeMillis();
        try {
            return ic.proceed();
        } finally {
            long diffTime = System.currentTimeMillis() - initTime;
            logger.fine(ic.getMethod()+"."+diffTime+" millis");
        }
    }
}
```

Il logger viene iniettato

logMethod() usato per postconstruction  
@PostConstruct.  
Prende in input un InvocationContext restituisce  
void

profile() usato come metodo interceptor  
@AroundInvoke

170

# Come si definiscono gli Interceptor

## Life-Cycle Interceptors

Listing 2-26

```
public class ProfileInterceptor {
    @Inject
    private Logger logger;
    @PostConstruct
    public void logMethod(InvocationContext ic) throws
        Exception {
        logger.fine(ic.getTarget().toString());
        try {
            ic.proceed();
        } finally {
            logger.fine(ic.getTarget().toString());
        }
    }
    @AroundInvoke
    public Object profile(InvocationContext ic) throws
        Exception {
        long initTime = System.currentTimeMillis();
        try {
            return ic.proceed();
        } finally {
            long diffTime = System.currentTimeMillis() - initTime;
            logger.fine(ic.getMethod()+"."+diffTime+" millis");
        }
    }
}
```

Il logger viene iniettato

logMethod() usato per postconstruction  
@PostConstruct.  
Prende in input un InvocationContext restituisce  
void

profile() usato come metodo interceptor  
@AroundInvoke

Per ogni metodo si calcola il tempo... si ferma  
l'orologio

171

# Come si definiscono gli Interceptor

## Life-Cycle Interceptors

Listing 2-26

```
public class ProfileInterceptor {
    @Inject
    private Logger logger;
    @PostConstruct
    public void logMethod(InvocationContext ic) throws
        Exception {
        logger.fine(ic.getTarget().toString());
        try {
            ic.proceed();
        } finally {
            logger.fine(ic.getTarget().toString());
        }
    }
    @AroundInvoke
    public Object profile(InvocationContext ic) throws
        Exception {
        long initTime = System.currentTimeMillis();
        try {
            return ic.proceed();
        } finally {
            long diffTime = System.currentTimeMillis() - initTime;
            logger.fine(ic.getMethod()+"."+diffTime+" millis");
        }
    }
}
```

Il logger viene iniettato

logMethod() usato per postconstruction  
@PostConstruct.  
Prende in input un InvocationContext restituisce  
void

profile() usato come metodo interceptor  
@AroundInvoke

Per ogni metodo si calcola il tempo... si ferma  
l'orologio

... si esegue il metodo invocato

172

## Come si definiscono gli Interceptor Life-Cycle Interceptors

Listing 2-26

```
public class ProfileInterceptor {
    @Inject
    private Logger logger;
    @PostConstruct
    public void logMethod(InvocationContext ic) throws
        Exception {
        logger.fine(ic.getTarget().toString());
        try {
            ic.proceed();
        } finally {
            logger.fine(ic.getTarget().toString());
        }
    }
    @AroundInvoke
    public Object profile(InvocationContext ic) throws
        Exception {
        long initTime = System.currentTimeMillis();
        try {
            return ic.proceed();
        } finally {
            long diffTime = System.currentTimeMillis() - initTime;
            logger.fine(ic.getMethod()+"."+diffTime+" millis");
        }
    }
}
```

Il logger viene iniettato

logMethod() usato per postconstruction  
@PostConstruct.

Prende in input un InvocationContext restituisce  
void

profile() usato come metodo interceptor  
@AroundInvoke

Per ogni metodo si calcola il tempo... si ferma  
l'orologio

> ... si esegue il metodo invocato

> ... e si fa il log di quanto tempo ha preso

173

## Come si definiscono gli Interceptor Life-Cycle Interceptors

- Life-cycle interceptors prendono un InvocationContext come parametro e restituiscono void anziché un Object
- Per applicare l'interceptor appena visto (Listing 2-26), il bean **CustomerService** (Listing 2-27) ha necessità di:
  - Usare l'annotazione @Interceptors
  - Definire l'interesse per ProfileInterceptor

174

## Come si definiscono gli Interceptor

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

    @Inject
    private EntityManager em;

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

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

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

Quando il bean è **istanziato** dal container, **il metodo `logMethod()` verrà invocato prima del metodo `init()`**

Se un client chiama `createCustomer()` oppure `findCustomerById()` il metodo `profile()` verrà invocato

175

## Organizzazione della lezione

- Introduzione
  - Dependency Injection
  - Life-cycle Management
  - Interception
  - Loose Coupling and Strong Typing
  - Deployment Descriptor
- Come scrivere un CDI Bean
  - Injection
  - Qualifiers
  - Producers/Disposers
  - Scope
- Interceptors
  - Classi Interceptor e Ciclo di vita
  - Interceptor multipli
- Decorators e Eventi
- Conclusioni

176

## Concatenazione di Interceptor

- E' possibile chiedere l'invocazione di più interceptors, passando una lista di interceptors
- L'ordine con cui sono invocati è determinato dall'ordine con cui sono specificati nell'annotazione @Interceptors

177

## Concatenazione di Interceptor

@Stateless

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

» Per tutti metodi (tranne quelli esclusi): si definiscono: I1, I2

178



## Concatenazione di Interceptor

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

Per tutti metodi (tranne quelli esclusi) si definiscono: I1, I2

Qui vengono chiamati prima I1 poi I2

179

## Concatenazione di Interceptor

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

Per tutti metodi (tranne quelli esclusi) si definiscono: I1, I2

Qui vengono chiamati prima I1 poi I2

Qui vengono chiamati I1, I2 poi I3, I4

180

## Concatenazione di Interceptor

@Stateless

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

Per tutti metodi (tranne quelli esclusi) si definiscono: I1, I2

Qui vengono chiamati prima I1 poi I2

Qui vengono chiamati I1, I2 poi I3, I4

Qui non viene chiamato nessun interceptor

181

## Binding fra Interceptor

- Gli interceptor sono specificati in maniera diretta nel codice, finora @Interceptors(LoggingInterceptor.class)
- Questo è typesafe ma non è loosely coupled: stretta dipendenza tra il codice e l'interceptor
- Si può effettivamente disaccoppiare l'interceptor dal codice che lo usa permettendo il binding
- Il binding è una annotazione @InterceptorBinding che permette all'utente di fare il binding non direttamente verso una classe
- Un esempio della definizione di un Binding @Loggable

```
@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Loggable {}
```

Codice simile a quello  
usato per i  
Qualificatori

182

## Binding fra Interceptor

- Una volta definito l'interceptor binding bisogna attaccarlo all'interceptor stesso
- Tutto ciò viene fatto annotando l'interceptor con sia con `@Interceptor` che con l'interceptor binding `@Loggable`

183

## Come si usa il Binding

```
@Interceptor←
@Loggable
public class LoggingInterceptor
@Inject
private Logger logger;
@AroundInvoke
public Object logMethod(InvocationContext ic)
throws Exception {
    logger.entering(ic.getTarget().toString(),
        ic.getMethod().getName());
    try{
        return ic.proceed();
    }finally{
        logger.exiting(ic.getTarget().toString(),
            ic.getMethod().getName());
    }
}
```

E' un interceptor, ma ...

184

## Come si usa il Binding

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

E' un interceptor, ma ...

... ha binding per @Loggable

185

## Come si usa il Binding

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

E' un interceptor, ma ...

... ha binding per @Loggable

Viene iniettato un logger

186

## Come si usa il Binding

```
@Interceptor
@Loggable
public class LoggingInterceptor {
    @Inject
    private Logger logger;
    @AroundInvoke
    public Object logMethod(InvocationContext ic)
        throws Exception {
        logger.entering(ic.getTarget().toString(),
            ic.getMethod().getName());

        try{
            return ic.proceed();
        }finally{
            logger.exiting(ic.getTarget().toString(),
                ic.getMethod().getName());
        }
    }
}
```

E' un interceptor, ma ...

... ha binding per @Loggable

Viene iniettato un logger

Per ogni metodo si fa il log

187

## Come si usa il Binding

- Ora si può applicare l'interceptor al bean annotando la classe bean con l'interceptor binding
- **Loose coupling (l'implementazione della classe Interceptor è altrove)**
- L'interception binding è sul bean questo significa che ogni metodo verrà intercettato e loggato

```
@Transactional
@Loggable
public class CustomerService {
    @Inject
    private EntityManager em;

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

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

188

## Come si usa il Binding

```
@Transactional
@Interceptors(LoggingInterceptor.class)
public class CustomerService {
    @Inject
    private EntityManager em;
    public void createCustomer(Customer customer) {}
    public Customer findCustomerById(Long id) {}
}
```

```
@Transactional
@Loggable
public class CustomerService {
    @Inject
    private EntityManager em;

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

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

189

## Come si usa il Binding

- Come per gli interceptor, l'interceptor binding può essere applicato anche ad un solo metodo anziché su tutto il bean

```
@Transactional
public class CustomerService {
    @Loggable
    public void createCustomer(Customer customer) {...}
    public Customer findCustomerById(Long id) {...}
}
```

190

## Il file beans.xml per Interceptor

- Come per le alternative, va abilitato esplicitamente l'uso degli interceptor

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
"http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans 1 1.xsd"
version="1.1"bean-discovery-mode="all">

<interceptors>
<class>
org.agoncal.book.javaee7.chapter02.LoggingInterceptor
</class>
</interceptors>
</beans>
```

Namespace

191

## Il file beans.xml per Interceptor

- Come per le alternative, va abilitato esplicitamente l'uso degli interceptor

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
"http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans 1 1.xsd"
version="1.1"bean-discovery-mode="all">

<interceptors>
<class>
org.agoncal.book.javaee7.chapter02.LoggingInterceptor
</class>
</interceptors>
</beans>
```

Namespace

Indicazione dell' XML  
Schema usato per la  
validazione

192

## Il file beans.xml per Interceptor

- Come per le alternative, va abilitato esplicitamente l'uso degli interceptor

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
"http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans 1 1.xsd"
version="1.1" bean-discovery-mode="all">
<interceptors>
<class>
org.agoncal.book.javaee7.chapter02.LoggingInterceptor
</class>
</interceptors>
</beans>
```

- > Namespace
- > Indicazione dell' XML Schema usato per la validazione

Attenzione! "all"! Altrimenti prende solamente quelli "annotated" (stay tuned!)

193

## Il file beans.xml per Interceptor

- Come per le alternative, va abilitato esplicitamente l'uso degli interceptor

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
"http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans 1 1.xsd"
version="1.1" bean-discovery-mode="all">
<interceptors>
<class>
org.agoncal.book.javaee7.chapter02.LoggingInterceptor
</class>
</interceptors>
</beans>
```

Namespace

Indicazione dell' XML Schema usato per la validazione

Attenzione! "all"! Altrimenti prende solamente quelli "annotated" (stay tuned!)

Indicazione delle classi degli interceptor

194



## Priorità fra Interceptors

- L'Interceptor Binding introduce un livello di indirectione ma fa perdere la possibilità di ordinare gli interceptors
  - `@Interceptor({I1.class, I2.class})`
- Soluzione:
  - Possibile definire una priorità
- La regola: interceptors con valori più piccoli verranno invocati per primi

195

## Priorità fra Interceptors

Interceptor



```
@Interceptor
@Loggable
@Priority(200)
public class LoggingInterceptor {
    @Inject
    private Logger logger;
    @AroundInvoke
    public Object logMethod(InvocationContext ic) throws
        Exception {
        logger.entering(ic.getTarget().toString(),
            ic.getMethod().getName());
        try{
            return ic.proceed();
        }finally{
            logger.exiting(ic.getTarget().toString(),
                ic.getMethod().getName());
        }
    }
}
```

196

# Priorità fra Interceptors

Interceptor

Binding

```
@Interceptor
@Loggable
@Priority(200)
public class LoggingInterceptor {
    @Inject
    private Logger logger;
    @AroundInvoke
    public Object logMethod(InvocationContext ic) throws
        Exception {
        logger.entering(ic.getTarget().toString(),
            ic.getMethod().getName());

        try{
            return ic.proceed();
        }finally{
            logger.exiting(ic.getTarget().toString(),
                ic.getMethod().getName());
        }
    }
}
```

197

# Priorità fra Interceptors

Interceptor

Binding

Qui si indica la priorità: intero, più basso maggiore priorità

```
@Interceptor
@Loggable
@Priority(200)
public class LoggingInterceptor {
    @Inject
    private Logger logger;
    @AroundInvoke
    public Object logMethod(InvocationContext ic) throws
        Exception {
        logger.entering(ic.getTarget().toString(),
            ic.getMethod().getName());

        try{
            return ic.proceed();
        }finally{
            logger.exiting(ic.getTarget().toString(),
                ic.getMethod().getName());
        }
    }
}
```

198

## Priorità fra Interceptors

```
@Interceptor
@Loggable
@Priority(200)
public class LoggingInterceptor {
    @Inject
    private Logger logger;
    @AroundInvoke
    public Object logMethod(InvocationContext ic) throws
        Exception {
        logger.entering(ic.getTarget().toString(),
            ic.getMethod().getName());

        try{
            return ic.proceed();
        }finally{
            logger.exiting(ic.getTarget().toString(),
                ic.getMethod().getName());
        }
    }
}
```

Interceptor

Binding

Qui si indica la priorità: intero, più basso, maggiore priorità

Si fa iniettare il logger

199

## Priorità fra Interceptors

```
@Interceptor
@Loggable
@Priority(200)
public class LoggingInterceptor {
    @Inject
    private Logger logger;
    @AroundInvoke
    public Object logMethod(InvocationContext ic) throws
        Exception {
        logger.entering(ic.getTarget().toString(),
            ic.getMethod().getName());

        try{
            return ic.proceed();
        }finally{
            logger.exiting(ic.getTarget().toString(),
                ic.getMethod().getName());
        }
    }
}
```

Interceptor

Binding

Qui si indica la priorità: intero, più basso maggiore priorità

Si fa iniettare il logger

Metodo interceptor

200

## Organizzazione della lezione

- Introduzione
  - Dependency Injection
  - Life-cycle Management
  - Interception
  - Loose Coupling and Strong Typing
  - Deployment Descriptor
- Come scrivere un CDI Bean
  - Injection
  - Qualifiers
  - Producers/Disposers
  - Scope
- Interceptors
  - Classi Interceptor e Ciclo di vita
  - Interceptor multipli
- Decorators e Eventi
- Conclusioni

201

## Decorators

- Gli interceptor servono a inserire task che siano trasversali alla logica di business (transazioni, sicurezza, logging)
- Per loro natura, **devono** essere totalmente all'oscuro della semantica delle azioni che intercettano
- I decoratori hanno un compito complementare: servono ad aggiungere logica ad un metodo di business
- Pattern storico della Gang of Four: si prende una classe e la si "decora" con una classe

202

# Decorators

- Un esempio:
  - l'ISSN generator genera numeri a 8 cifre
  - Vogliamo decorarlo con una classe che permette di trasformare ISSN a 8 cifre, in un ISBN a 13 cifre
  - Anziché usare entrambe le classi come fatto in precedenza (Listing 2-9 e 2-10)

Listing 2-9. The IsbnGenerator Bean with the @ThirteenDigits Qualifier

```
@ThirteenDigits
public class IsbnGenerator implements NumberGenerator {

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

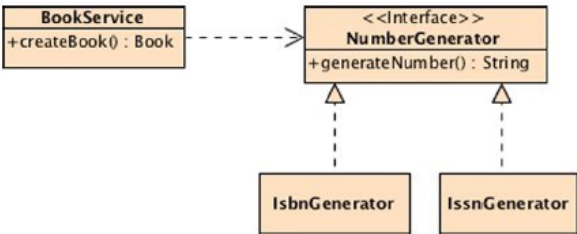
Listing 2-10. The IssnGenerator Bean with the @EightDigits Qualifier

```
@EightDigits
public class IssnGenerator implements NumberGenerator {

    public String generateNumber() {
        return "8-" + Math.abs(new Random().nextInt());
    }
}
```

# Il punto di partenza dell'esempio

- Due classi che implementano la stessa interfaccia
- Usati da un Bean CDI BookService che ha come punto di iniezione un riferimento ad un NumberGenerator



## Decorators

- Il nostro obiettivo:
  - Decorare un ISSN generator number per aggiungere un algoritmo extra che trasformi un numero 8-digit in un numero 13-digit

205

## Esempio di Decorator

Listing 2-34

```
@Decorator <
public class From8To13DigitsDecorator
    implements NumberGenerator {
    @Inject @Delegate
    private NumberGenerator numberGenerator;

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

E' un decoratore

206

# Esempio di Decorator

Listing 2-34

```
@Decorator
public class From8To13DigitsDecorator
    implements NumberGenerator {
    @Inject @Delegate
    private NumberGenerator numberGenerator;

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

E' un decoratore  
Nome del bean

207

# Esempio di Decorator

Listing 2-34

```
@Decorator
public class From8To13DigitsDecorator
    implements NumberGenerator {
    @Inject @Delegate
    private NumberGenerator numberGenerator;

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

E' un decoratore  
Nome del bean  
Quali funzionalità (astratte) implementa

208

# Esempio di Decorator

Listing 2-34

```
@Decorator
public class From8To13DigitsDecorator
    implements NumberGenerator {
    @Inject @Delegate
    private NumberGenerator numberGenerator;

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

E' un decoratore

Nome

Quali funzionalità (astratte) implementa

Il punto di iniezione serve ad inserire l'oggetto delegato. Si permette al decorator di invocare l'oggetto delegato (il target bean IssnNumberGenerator) e pertanto di invocare ogni metodo di business su di esso (numberGenerator.generateNumber())

209

# Esempio di Decorator

Listing 2-34

```
@Decorator
public class From8To13DigitsDecorator
    implements NumberGenerator {
    @Inject @Delegate
    private NumberGenerator numberGenerator;

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

E' un decoratore

Nome

Quali funzionalità (astratte) implementa

Il punto di iniezione serve a inserire l'oggetto delegato. Si permette al decorator di invocare l'oggetto delegato (il target bean IssnNumberGenerator) e pertanto di invocare ogni metodo di business su di esso (numberGenerator.generateNumber())

Metodo generateNumber() che sovrascrive il metodo originale

Metodo che sovrascrive il metodo originale

210



## Il file beans.xml per Decorator

- Di default i decorator sono disabilitati come le alternative ed interceptor

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
"http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
version="1.1" bean-discovery-mode="all">

<decorators>
<class>
org.agoncal.book.javaee7.chapter02.From8To13DigitsDecorator
</class>
</decorators>
</interceptors>
</beans>
```

211

## Gli eventi

- Dependency Injection, Alternative, Interceptor e Decorators servono a offrire funzionalità e comportamento aggiuntivo, debolmente accoppiato, stabilito a tempo di deploy o a tempo di esecuzione
- Gli eventi fanno un passo avanti...
  - permettono le interazioni tra i beans senza nessuna dipendenza a tempo di compilazione

212

## Gli eventi

- Un bean può definire un evento, un altro lo lancia, e un altro bean lo gestisce, ma...
  - ...i bean possono essere in package separati, o in layer separati
- Questo schema segue il design pattern Observer/Observable
- Un produttore (Event producer) lancia eventi:
  - usando l'interfaccia `javax.enterprise.event.Event`
  - chiamando il metodo `fire()` e passando l'oggetto evento

213

## Gli eventi: un esempio

- Un esempio: ogni volta che si crea un libro viene lanciato un evento per un servizio di inventario
  - `BookService` lancia un evento (`BookAddEvent`) ogni volta che un libro viene creato
  - Il codice `bookAddEvent.fire(book)` lancia l'evento e notifica ogni metodo observer interessato a questo evento
  - Il contenuto dell'evento è l'oggetto `Book`, mandato dal producer al consumer

214

# Esempio di eventi

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

    @Inject
    private Event<Book> bookAddedEvent;

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

Listing 2-36

```
public class InventoryService {
    @Inject
    private Logger logger;
    List<Book> inventory = new ArrayList<>();
    public void addBook(@Observes Book book) {
        logger.info("Adding book"+ book.getTitle() +
                    "to inventory");
        inventory.add(book);
    }
}
```

Listing 2-37

Iniezione di dipendenza

# Esempio di eventi

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

    @Inject
    private Event<Book> bookAddedEvent;

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

Listing 2-36

```
public class InventoryService {
    @Inject
    private Logger logger;
    List<Book> inventory = new ArrayList<>();
    public void addBook(@Observes Book book) {
        logger.info("Adding book"+ book.getTitle() +
                    "to inventory");
        inventory.add(book);
    }
}
```

Listing 2-37

Iniezione di dipendenza

Viene iniettato un evento che  
si può produrre (di tipo Book)

# Esempio di eventi

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

    @Inject
    private Event<Book> bookAddedEvent;

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

Listing 2-36

```
public class InventoryService {
    @Inject
    private Logger logger;
    List<Book> inventory = new ArrayList<>();
    public void addBook(@Observes Book book){
        logger.info("Adding book"+ book.getTitle() +
                    "to inventory");
        inventory.add(book);
    }
}
```

Listing 2-37

Iniezione di dipendenza  
Viene iniettato un evento che  
si può produrre (di tipo Book)  
Viene lanciato l'evento

# Esempio di eventi

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

    @Inject
    private Event<Book> bookAddedEvent;

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

Listing 2-36

```
public class InventoryService {
    @Inject
    private Logger logger;
    List<Book> inventory = new ArrayList<>();
    public void addBook(@Observes Book book){
        logger.info("Adding book"+ book.getTitle() +
                    "to inventory");
        inventory.add(book);
    }
}
```

Listing 2-37

Iniezione di dipendenza  
Viene iniettato un evento che  
si può produrre (di tipo Book)  
Viene lanciato l'evento  
In un'altra classe ...

# Esempio di eventi

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

    @Inject
    private Event<Book> bookAddedEvent;

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

Listing 2-36

```
public class InventoryService {
    @Inject
    private Logger logger;
    List<Book> inventory = new ArrayList<>();
    public void addBook(@Observes Book book){
        logger.info("Adding book"+ book.getTitle() +
                    "to inventory");
        inventory.add(book);
    }
}
```

Listing 2-37

Iniezione di dipendenza

Viene iniettato un evento che  
si può produrre (di tipo Book)

Viene lanciato l'evento

In un'altra classe ...

Il parametro di questo metodo è  
annotato con @Observes: riceve  
tutti gli eventi di quel tipo  
(Book) che vengono generati

# Esempio di eventi

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

    @Inject
    private Event<Book> bookAddedEvent;

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

Listing 2-36

```
public class InventoryService {
    @Inject
    private Logger logger;
    List<Book> inventory = new ArrayList<>();
    public void addBook(@Observes Book book){
        logger.info("Adding book"+ book.getTitle() +
                    "to inventory");
        inventory.add(book);
    }
}
```

Listing 2-37

ATTENZIONE! Gestione eventi  
sincrona: createBook() attende  
che addBook() sia terminato!

## Gestione di eventi diversi

- Rivisitiamo il bean BookService per aggiungere un altro evento
  - Quando un book viene creato
    - Viene generato un evento bookAddedEvent
  - Quando un book viene rimosso
    - Viene generato un evento bookRemovedEvent
  - Entrambi di tipo Book
- Per distinguere i due eventi questi vengono qualificati con @Added o @Removed
  - Si usano pertanto i qualificatori

221

## Il BookService con diversi Eventi

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

    @Inject @Added
    private Event<Book> bookAddedEvent;

    @Inject @Removed
    private Event<Book> bookRemovedEvent;

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

    public void deleteBook(Book book) {
        bookRemovedEvent.fire(book);
    }
}
```

Iniettato l'evento @Added da produrre

Possibile differenziare tra eventi, con tipi (typesafe)

222

# Il BookService con diversi Eventi

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

    @Inject @Added
    private Event<Book> bookAddedEvent;

    @Inject @Removed
    private Event<Book> bookRemovedEvent;

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

    public void deleteBook(Book book) {
        bookRemovedEvent.fire(book);
    }
}
```

- > Iniettato l'evento @Added da produrre
- > Iniettato l'evento @Removed da produrre

223

# Il BookService con diversi Eventi

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

    @Inject @Added
    private Event<Book> bookAddedEvent;

    @Inject @Removed
    private Event<Book> bookRemovedEvent;

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

    public void deleteBook(Book book) {
        bookRemovedEvent.fire(book);
    }
}
```

- Iniettato l'evento @Added da produrre
- Iniettato l'evento @Removed da produrre
- Viene generato l'evento di add

224

## Il BookService con diversi Eventi

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

    @Inject @Added
    private Event<Book> bookAddedEvent;

    @Inject @Removed
    private Event<Book> bookRemovedEvent;

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

    public void deleteBook(Book book) {
        bookRemovedEvent.fire(book);
    }
}
```

Iniettato l'evento @Added da produrre

Iniettato l'evento @Removed da produrre

Viene generato l'evento di add

Viene generato l'evento di remove

225

## Un Observer del BookService

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

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

    public void removeBook(@Observes @Removed Book book) {
        logger.warning("Removing book" + book.getTitle() +
            "from inventory");
        inventory.remove(book);
    }
}
```

Osserva un evento Book ma qualificato da @Added

Osserva entrambi gli eventi, in due metodi diversi

226



## Un Observer del BookService

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

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

    public void removeBook(@Observes @Removed Book book) {
        logger.warning("Removing book"+ book.getTitle() +
            "from inventory");
        inventory.remove(book);
    }
}
```

Osserva un evento Book ma qualificato  
da @Added

Osserva un evento Book ma qualificato  
da @Removed

227

## Un Observer del BookService

- Uso dei membri
- Il seguente codice "intercetta" tutti i libri aggiunti che hanno un prezzo maggiore di 100

```
void addBook(@Observes @Added @Price(greaterThan=100) Book book)
```

228



Corso di Laurea in Informatica  
III Anno Triennale  
Programmazione Distribuita



# Context and Dependency Injection

Delfina Malandrino

[dmalandrino@unisa.it](mailto:dmalandrino@unisa.it)

<http://www.unisa.it/docenti/delfinamalandrino>