Laboratorio di Programmazione ad oggetti

Ph.D. Juri Di Rocco
juri.dirocco@univaq.it
http://jdirocco.github.io

Sommario

- Inversion of Control (IoC)
- Introduzione
- Moduli
- IoC Container
- Scope dei beans
- Annotazioni e classpath Scanning
- Configurazione Java

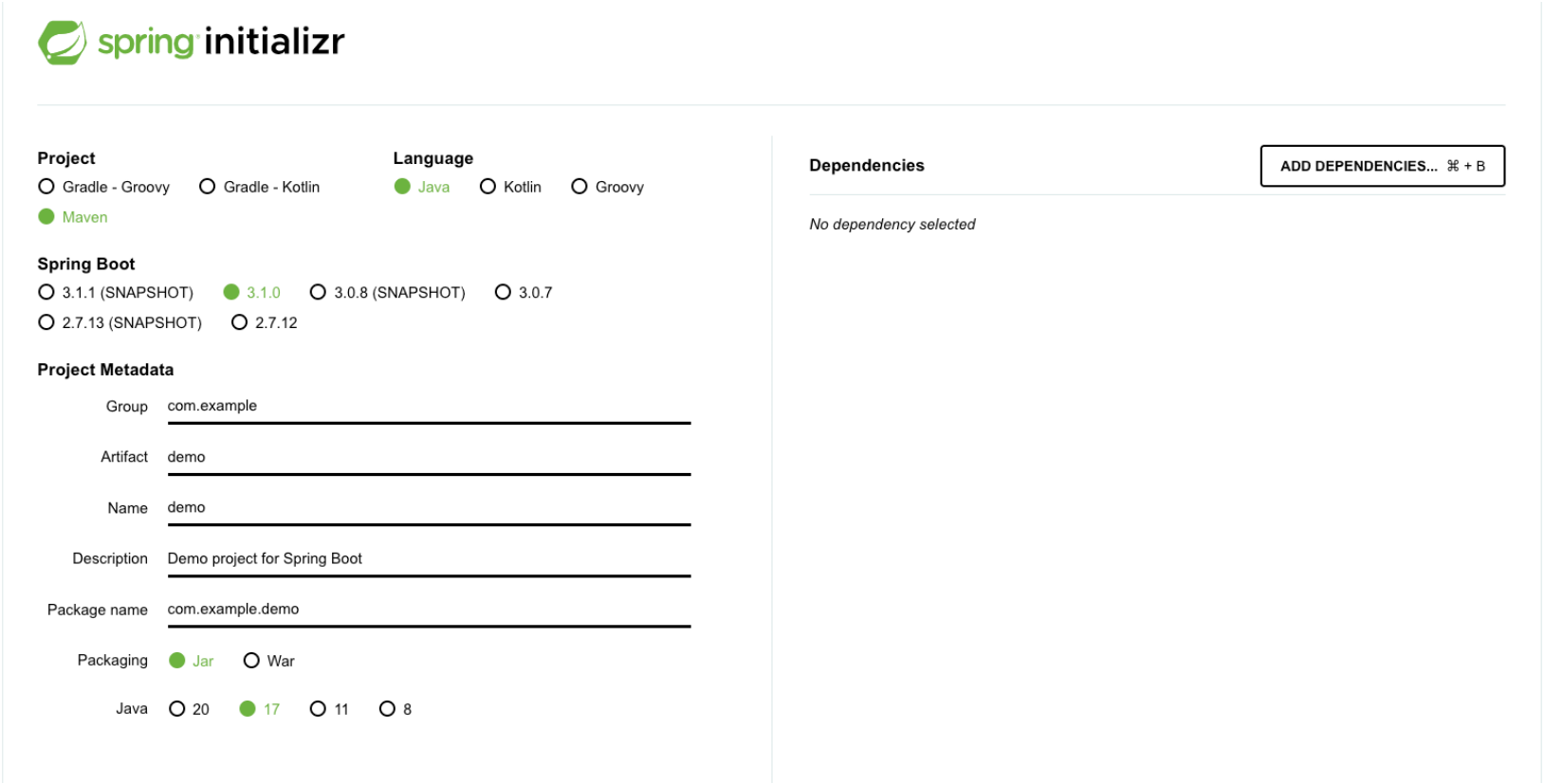
Risorse

- Inversion of Control
  http://martinfowler.com/articles/injection.html
- Libro
  - Titolo: Spring in Action (**quarta** edizione)
  - Autore: Craig Walls
  - ISBN: 9781617294945
- Sito

http://spring.io

Download boilerblate

› https://start.spring.io



Inversion of Control (1)

```
public interface MovieFinder {
    List<Movie> findAll();
}

public class ColonDelimitedMovieFinder implements MovieFinder {
     public ColonDelimitedMovieFinder(String fileName) {
          …..
     }
     public List<Movie> findAll() {
          ……

     }
}
```

Inversion of Control (2)

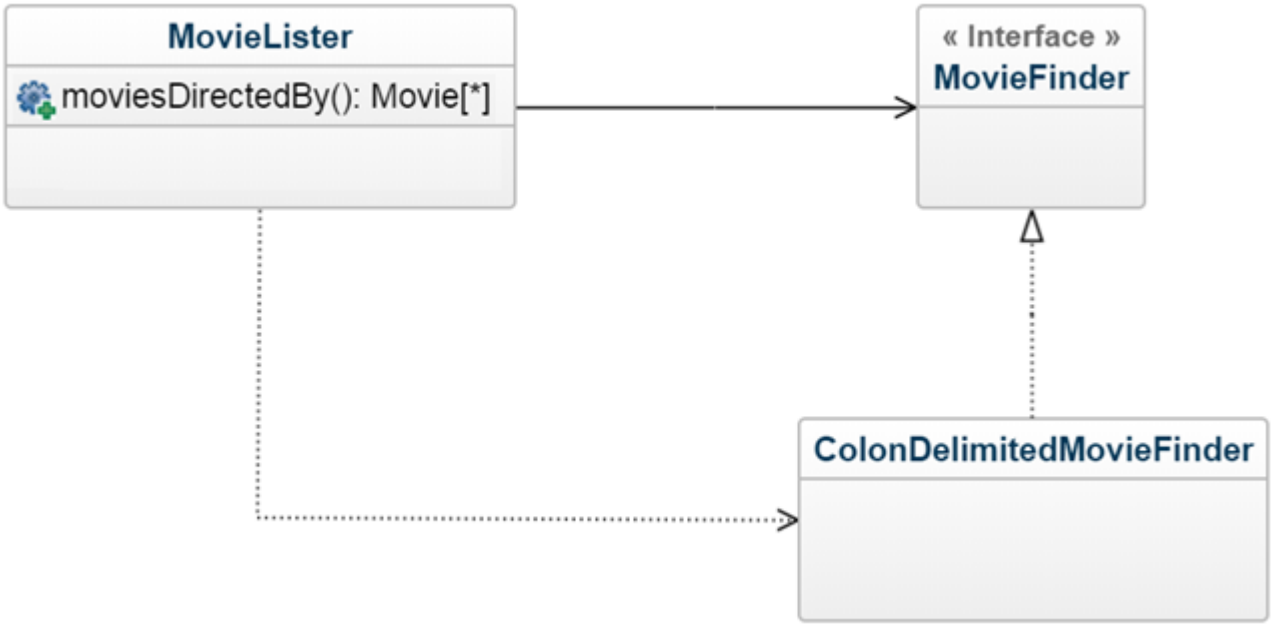
```
class MovieLister {
  private MovieFinder finder;
  public MovieLister() {
    finder = new ColonDelimitedMovieFinder("movies1.txt");
  }
  public Movie[] moviesDirectedBy(String arg) {
    List<Movie> allMovies = finder.findAll();
    List<Movie> result = new ArrayList<>();
    for (Movie movie: allMovies) {
                  if (movie.getDirector().equals(arg))
                        result.add(movie);
    }
    return (Movie[]) result.toArray(new Movie[result.size()]);
......
```

Inversion of Control (3)

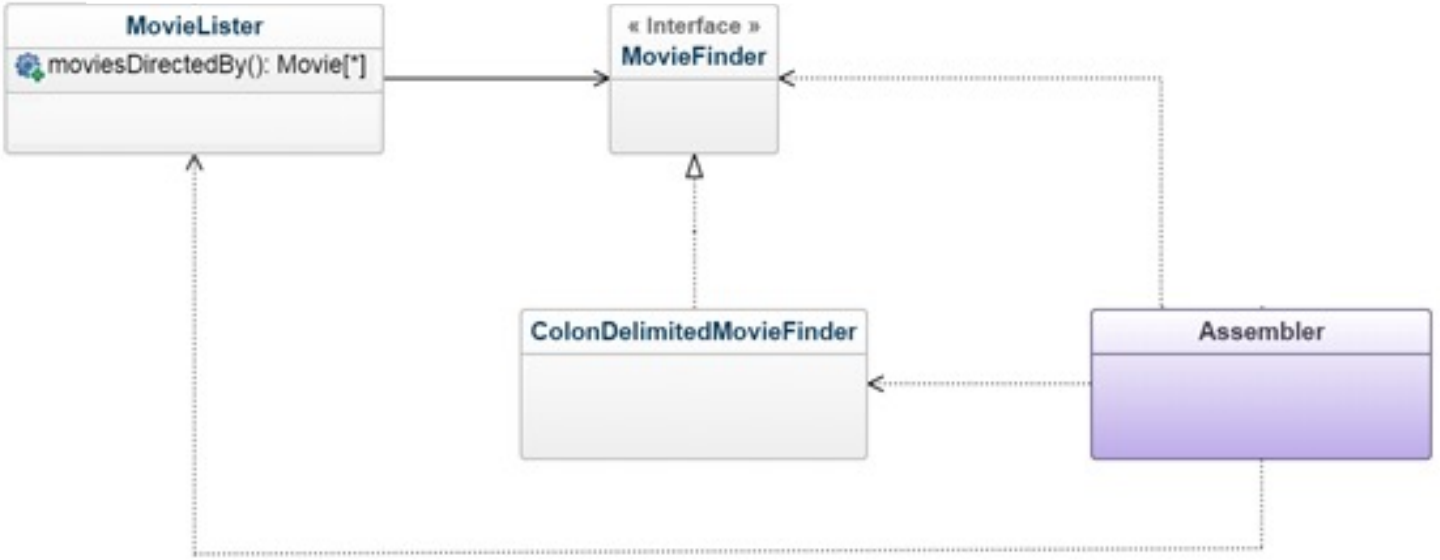
```
public class MovieApplication {

        public static void main(String[] args) {
           MovieLister movieLister = new MovieLister();
           Movie[] movies = movieLister.moviesDirectedBy("Sergio Leone");
           for (Movie movie: movies) {
                      //Print all
           }
                 ……
        }
}
```

Inversion of Control (4)



Inversion of Control (5)



› Due scopi principali
  – Creare componenti
  – Assemblare (Iniettare) componenti

› Stili principali di dependency injection
  – Constructor Injection
  – Setter Injection

Inversion of Control (6)

```
1)
class MovieLister {
    private MovieFinder finder;
    public MovieLister(MovieFinder finder) {
        this.finder = finder;
    }
}
2)
class MovieLister {
    private MovieFinder finder;
    public void setFinder(MovieFinder finder) {
        this.finder = finder;
    }
}
```

Inversion of Control (7)

```
public class MovieApplication {

 public static void main(String[] args) {
  MovieFinder movieFinder = new ColonDelimitedMovieFinder(…);
  MovieLister movieLister = new MovieLister(movieFinder);
  Movie[] movies = movieLister.moviesDirectedBy("Sergio Leone");
  for (Movie movie: movies) {
            //Print all
  }
    ……
 }

}
```

Inversion of Control (8)

```
public class MovieApplication {

 public static void main(String[] args) {
  MovieFinder movieFinder=new ColonDelimitedMovieFinder(…);
  MovieLister movieLister=new MovieLister();
  movieLister.setMovieFinder(movieFinder);
  Movie[] movies = movieLister.moviesDirectedBy("Sergio Leone");
  for (Movie movie: movies) {
            //Print all
  }
    ……
 }

}
```

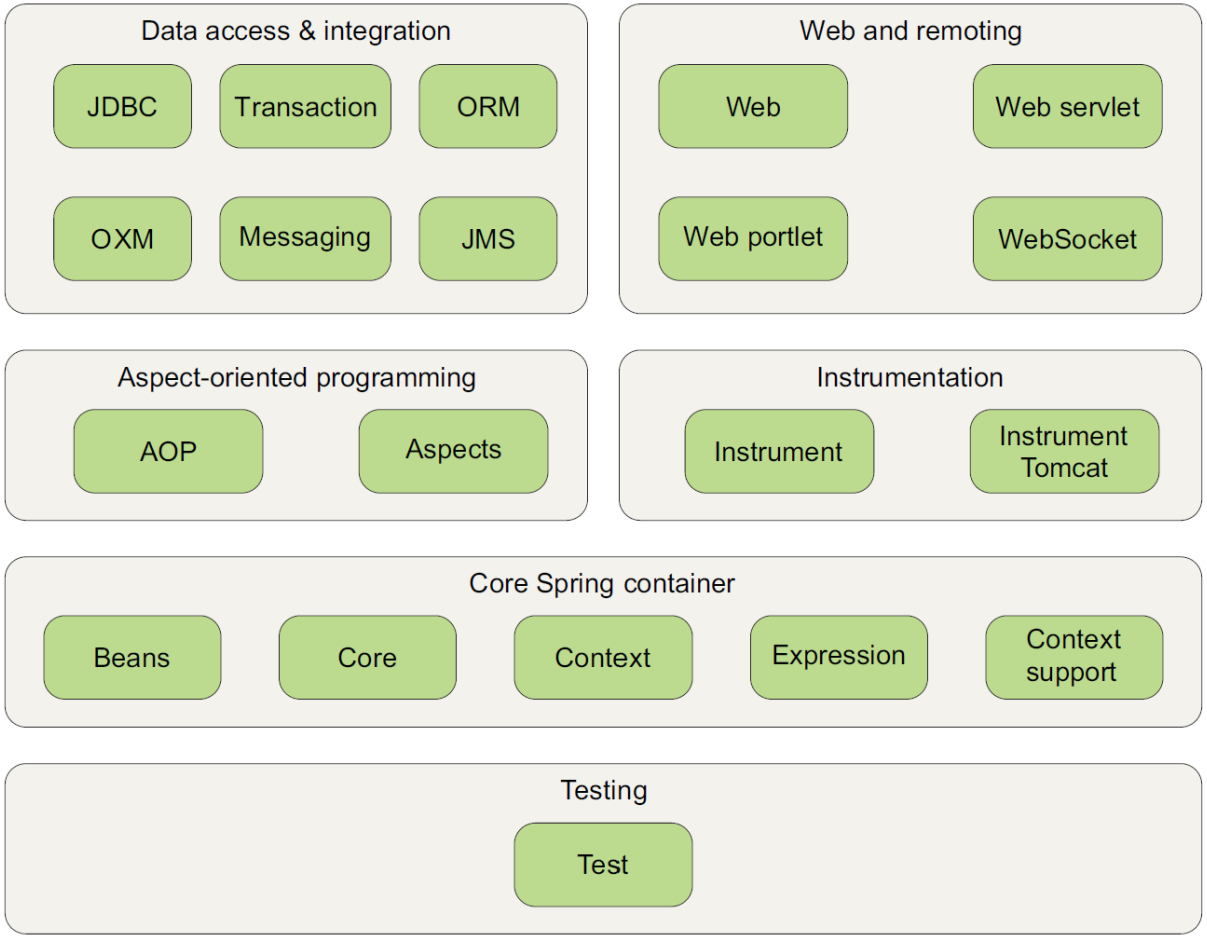
Introduzione (1)

- Spring è una piattaforma che fornisce una infrastruttura per lo sviluppo di applicazioni in Java
- Gestisce l'infrastruttura cosicché ci si può focalizzare sull'applicazione
- Permette di costruire applicazioni con "plain old Java objects" (POJOs) e di applicare servizi di livello enterprise in modo non invasivo ai POJO
- Particolarmente adatta allo sviluppo di applicazioni Java EE

Introduzione (2)

- Esempi di vantaggi nell'uso di Spring
  - Esecuzione di un metodo Java in una transazione di un database senza interagire con le API delle transazioni
  - Rendere un metodo locale Java come una procedura remota senza conoscere le API remote
  - Rendere un metodo locale Java gestore di messaggi senza conoscere le API JMS
  - …..
- Spring (tramite il componente IoC) **formalizza** il modo di **comporre** componenti per costruire una applicazione
- Spring formalizza design pattern come oggetti di "first-class" che possono essere integrati all'interno dell'applicazione

Moduli


Data access & integration
JDBC
Transaction
ORM
OXM
Messaging
JMS
Web and remoting
Web
Web servlet
Web portlet
WebSocket
Aspect-oriented programming
AOP
Aspects
Instrumentation
Instrument
Instrument Tomcat
Core Spring container
Beans
Core
Context
Expression
Context support
Testing
Test

Moduli: Core e Beans

- Forniscono le parti fondamentali del framework ovvero IoC e Dependency Injection
- Interfaccia `BeanFactory` è un'implementazione sofisticata del pattern factory
- Rimuovono la necessità di singleton programmatici e permette di disaccoppiare la configurazione e la specifica delle dipendenze dalla logica dell'applicazione

Moduli: Context

- Costruito sui moduli Core e Beans
- Permette di accedere agli oggetti con uno stile simile ad un registro JNDI
- Aggiunge alle caratteristiche del modulo Beans l'internazionalizzazione, propagazione degli eventi, loading delle risorse, ...
- Supporta anche le caratteristiche di Java EE come, EJB, JMX, ...
- Interfaccia `ApplicationContext` è il punto focale del modulo

IoC Container (1)

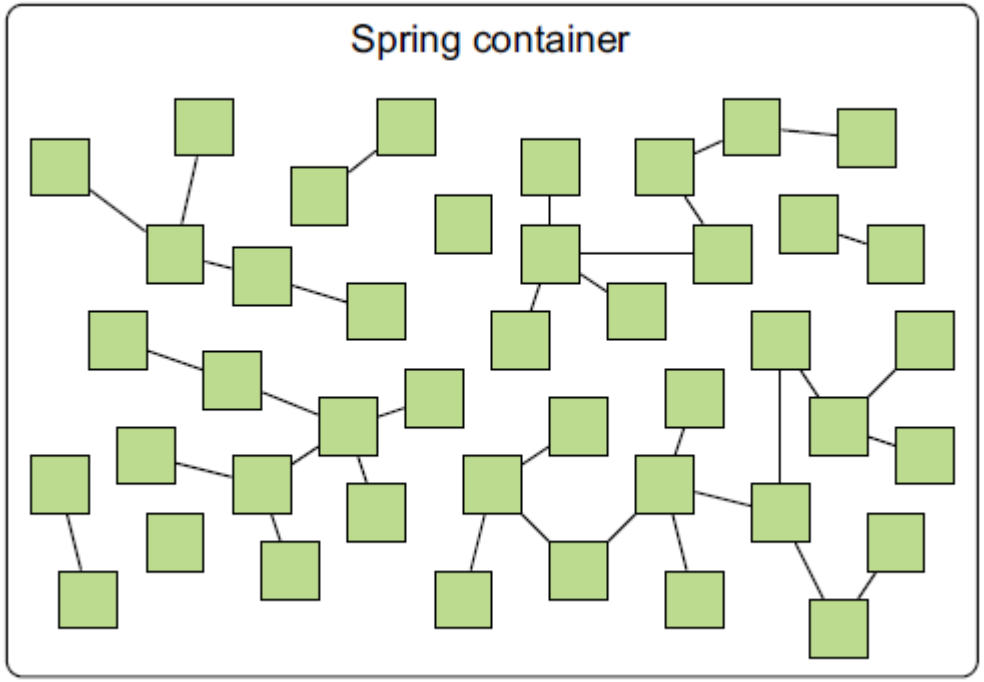
- Package `org.springframework.beans` e `org.springframework.context` basi per IoC container
- Interfaccia `BeanFactory` fornisce un meccanismo avanzato di configurazione capace di gestire qualsiasi tipo di oggetto
- `ApplicationContext` sotto-interfaccia di `BeanFactory`
  - Permette maggiore integrazione con caratteristiche di Spring come gestione delle risorse, pubblicazione degli eventi, ecc

IoC Container (2)

- Tramite metadati di configurazione IoC instanzia, configura e assembla oggetti
- Metadati di configurazione sono rappresentati tramite XML, annotazioni Java, codice Java
- Vengono fornite diverse implementazioni di `ApplicationContext`
- `ClassPathXmlApplicationContext` o `FileSystemXmlApplicationContext`

```
ApplicationContext context = new
ClassPathXmlApplicationContext(
                new String[] {"services.xml", "daos.xml"});
```

IoC Container (3)



Dependency Injection (DI)

- Processo dove gli oggetti definiscono le loro dipendenze rispetto agli altri oggetti
- Tali dipendenze possono essere specificate in diversi modi ovvero tramite argomenti del costruttore, argomenti ad un metodo factor, proprietà
- Container **inietta** tali dipendenze mentre **crea** il bean
- Codice più chiaro poiché disaccoppiamento tra oggetto e sue dipendenze

Processo Risoluzione dipendenze

- `ApplicationContext` viene creato ed inizializzato con i metadata di configurazione che descrivono i beans
- Metadata possono essere specificati tramite XML, codice Java o **annotazioni**
- Per ogni bean, le sue dipendenze sono fornite quando il bean è creato
- Ogni dipendenza è un riferimento a un altro bean oppure ad una proprietà definita all'interno del container (Spring effettua conversioni implicite)

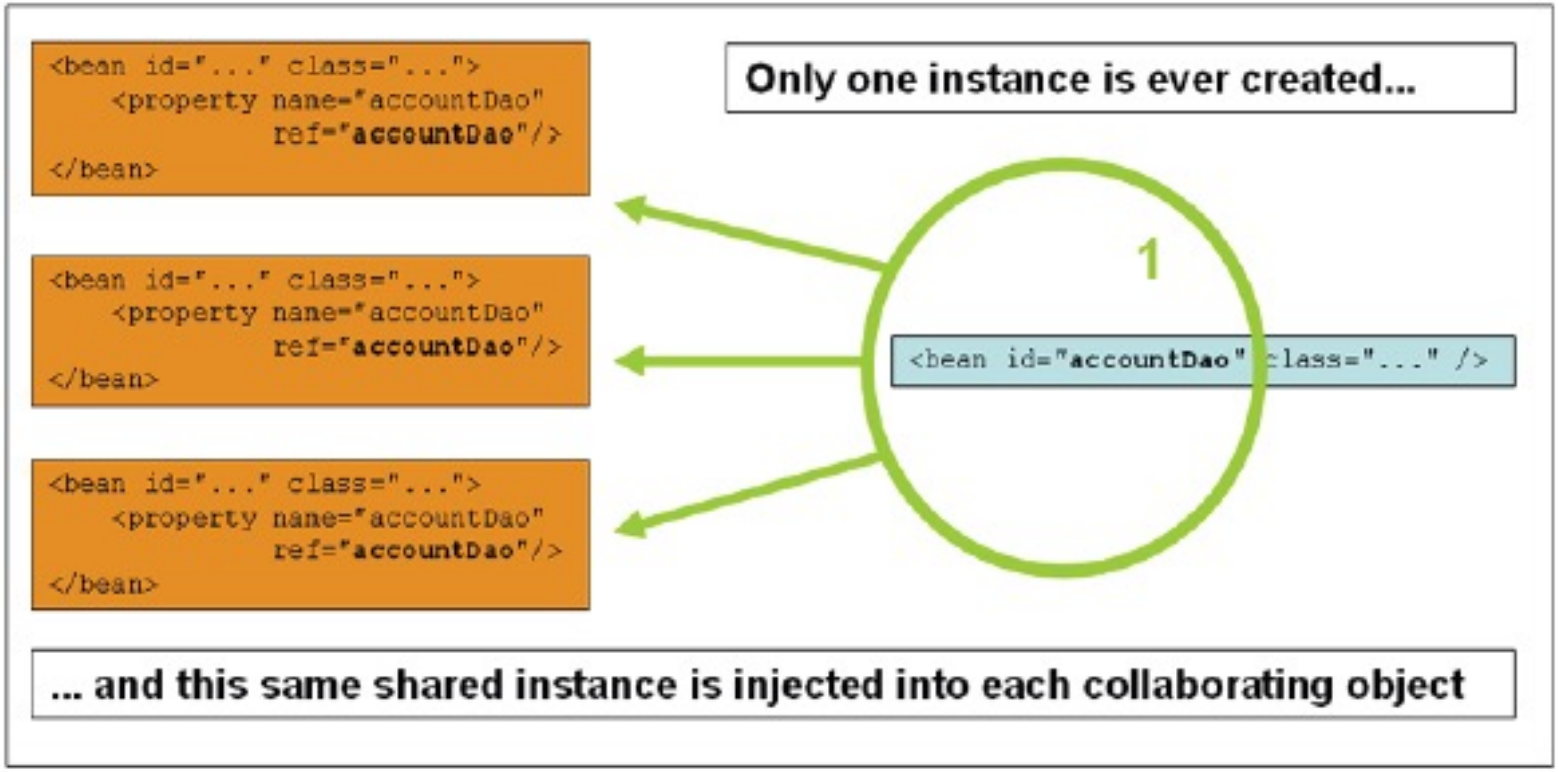
Scope dei beans (1)

- Quando si definisce un bean, si crea una “ricetta” per creare le istanze della classe che il bean definisce
- Spring permette di controllare lo scope degli oggetti creati per una particolare definizione di un bean
- Spring supporta **6** scope, tre dei quali sono disponibili soltanto all’interno dell’ `ApplicationContext` in ambito web

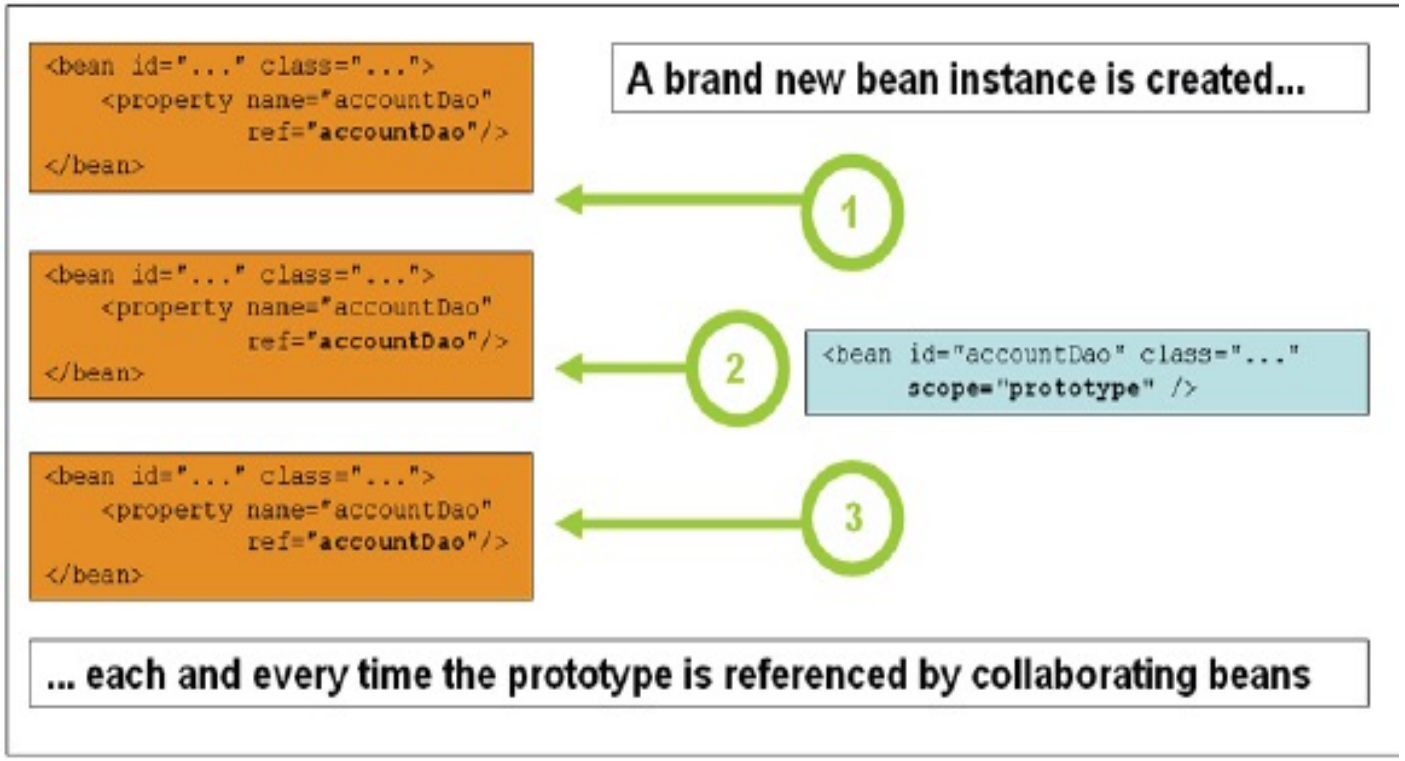
Scope dei beans (2)

- **singleton** (Default): definisce una singola istanza per la definizione del bean all'interno container Spring
- **prototype**: una singola definizione per un qualsiasi numero di istanze degli oggetti
- **request**: per ogni richiesta HTTP viene creata una istanza della definizione del bean
- **session**: per ogni sessione HTTP viene creata una istanza della definizione del bean
- **global session**: poco usato
- **websocket**: lo scope è relavito al ciclo di vita di una WebSocket. Valido soltanto nel contesto web di Spring

Scope dei beans: Singleton



Scope dei beans: Prototype



Wiring bean

- Spring offre tre meccanismi per creare bean e comporre (DI) i bean
  - File configurazione XML
  - Discovery implicito nella creazione dei bean (classpath scanning) e composizione automatica (autowiring)
  - File configurazione Java

Classpath scanning

- Annotazioni utilizzate
  - `@Component` è un generico stereotipo per qualsiasi componente gestito da Spring
  - `@Repository`, **`@Service`**, e `@Controller` sono specializzazioni di `@Component` per utilizzi specifici come per i layer di persistenza, servizio e presentation rispettivamente

Annotazioni

› Un metodo alternativo all'uso di file XML per definire le **dipendenze** è tramite l'annotazione **`@Autowired`**

Annotazioni: @Autowired

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Autowired
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

Annotazioni: @Autowired

```
public class MovieRecommender {

    private final CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public MovieRecommender(CustomerPreferenceDao customerDao) {
        this.customerPreferenceDao = customerDao;
    }

    // ...
}
```

A partire da Spring 4.3 l'annotazione **non** è necessaria se è presente un solo costruttore
Se ce ne sono diversi in almeno uno deve essere presente l'annotazione per indicare al container quale costruttore utilizzare

Annotazioni: @Autowired

```
public class MovieRecommender {

    private MovieCatalog movieCatalog;

    private CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public void prepare(MovieCatalog movieCatalog,
                        CustomerPreferenceDao customerPreferenceDao) {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }

    // ...
}
```

Annotazioni: @Autowired

```
public class MovieRecommender {

    private final CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    private MovieCatalog movieCatalog;

    @Autowired
    public MovieRecommender(CustomerPreferenceDao customerDao) {
        this.customerPreferenceDao = customerDao;
    }

    // ...
}
```

Annotazioni: @Autowired

```
public class MovieRecommender {
    @Autowired
    private MovieCatalog[] movieCatalogs;

    // ...
}
public class MovieRecommender {
    private Set<MovieCatalog> movieCatalogs;

    @Autowired
    public void setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {
        this.movieCatalogs = movieCatalogs;
    }
    // ...
}
```

Configurazione Java (1)

```
@Configuration
@ComponentScan
public class CDPlayerConfig {
    @Bean
    public CompactDisc sgtPeppers() {
        return new SgtPeppers();
    }
}
```

- `@Configuration` identifica una classe di configurazione e contiene dettagli sui beans che verranno create
- Classe annotate con `@Configuration` sostituisce file XML
- `@ComponentScan` abilita la scansione a tutti i package (e sotto-package) di dove è contenuta la classe `CDPlayerConfig`
- `@Bean` è l'equivalente del tag `bean` dentro il file XML

Configurazione Java (2)

```
@Bean
public CompactDisc sgtPeppers() {
   return new SgtPeppers();
}
@Bean
public CDPlayer cdPlayer() {
   return new CDPlayer(sgtPeppers());
}
```

- `sgtPeppers()` è annotato con `@Bean` pertanto Spring intercetta tale chiamata e ritorna una istanza del bean registrato
- Id del bean è come il nome del metodo ovvero `cdPlayer()`

Configurazione Java (3)

```
@Bean
public CDPlayer cdPlayer() {
   return new CDPlayer(sgtPeppers());
}
@Bean
public CDPlayer anotherCDPlayer() {
   return new CDPlayer(sgtPeppers());
}
```

- `sgtPeppers()` non viene trattato come una qualsiasi invocazione di un metodo Java: ovvero ogni istanza di `CDPlayer` ha la propria istanza
- Spring intercetta la chiamata, e poiché tutti i beans per default sono singleton le singole istanze di `CDPlayer` avranno la stessa istanza del bean di `sgtPeppers`

Configurazione Java (4)

```
@Bean
public CDPlayer cdPlayer(CompactDisc compactDisc) {
    return new CDPlayer(compactDisc);
}
```

- Istanza di `CompactDisc` viene passata come parametro da Spring utilizzando il bean creato in precedenza con `sgtPeppers()`
- Codice più leggibile e la creazione del bean relativo a `CompactDisc` può essere specificata in modi diversi