Spring Framework
Module 2 – Components Model (IoC, DI)

Evgeniy Krivosheev
Andrey Stukalenko
Vyacheslav Yakovenko
Last update: Feb, 2012

Содержание

LUXOFT TRAINING

- Inversion of Control (IoC) / Dependency Injection (DI)
- Семейство IoC контейнеров в Spring Framework
- Работа с IoC контейнером
- Beans как компоненты
- Внешние зависимости
- Dependency Injection (DI)
- Autowiring
- Использование аннотаций
- Области видимости бинов
- Жизненный цикл бина
- Дополнительные возможности ApplicationContext

Spring Framework :: IoC / DI

- В основе Spring лежит паттерн Inversion of Control (IoC)
  - “Hollywood Principle” – Don't call me, I'll call you (Какой дизайн-шаблон из каталога GoF ассоциируется у вас с таким же слоганом?);
  - Основная идея – устранение зависимости компонентов приложения от конкретных реализаций и делегировании полномочий по управлению созданием нужных экземпляров классов IoC контейнеру.
- Мартин Фаулер (Martin Fowler) предложил название Dependency Injection (DI) – оно лучше отражает суть паттерна (http://martinfowler.com/articles/injection.htm)

Spring Framework :: IoC / DI

- **Задача:**
  - Два компонента – A и B
  - Компонент A использует функционал компонента B

- **Классическое «прямое» решение:**

```
public class A {
    private B m_b = new B();
    public doSomething() {
        b.do();
    }
}
```

- **Проблемы:**
  - Класс A напрямую зависит от класса B;
  - Невозможно тестировать A в отрыве от B (если для B нужна база – для тестирования A она также понадобится);
  - Временем жизни объекта B управляет A – нельзя использовать тот же объект в других местах;
  - Нельзя «подменить» B на другую реализацию;

Spring Framework :: IoC / DI

- **Выделение интерфейса:**

```
public class A {
   private IB m_ib = new B();
   public void doSomething() {
      m_ib.do();
   }
}
public class B implements IB { … }
public interface IB { … }
```

- **Проблемы:**
  - Класс A напрямую зависит от класса B;
  - Невозможно тестировать A в отрыве от B (если для B нужна база – для тестирования A она также понадобится);
  - Временем жизни объекта B управляет A – нельзя использовать тотже объект в других местах;
  - «Подменить» B на другую реализацию можно, но только через изменение кода;
  - Итог: ни одну проблему не решили.

Spring Framework :: IoC / DI

- **Service Locator** – имеется фабрика, которая по идентификатору возвращает нужную реализацию интерфейса:

```
public class A {
    private IB m_ib = serviceLocator.getService(IB.class);
    public void doSomething() {
        m_ib.do();
    }
}
```

- **Плюсы:**
  - Временем жизни B управляет локатор;
  - Класс A не зависит от реализации IB, соответственно, можно подсунуть любую нужную реализацию;
- **Проблемы:**
  - Класс A зависит от Service Locator;
  - Тестировать A в отрыве от B возможно, но потребуется настройка локатора, вполне возможно нетривиальная;

Spring Framework :: IoC / DI

- **Dependency Injection** – компонент A не ищет зависимости сам, а получает их извне (например, через конструктор или метод-сеттер):

```
public interface IB { … }
public class A {
   private IB m_ib;
   public A(IB value) { m_ib = value; }
   public void doSomething() {
      m_ib.do();
   }
}
```

- **Плюсы:**
  - A не зависит ни от чего;
  - Значительно упрощает тестирование A;
- **Проблемы:**
  - Не решает проблемы в большом масштабе;
  - … = new A(new B(new C(new D(new E(…);
  - Возникают серьезные проблемы с управлением временем жизни объектов;

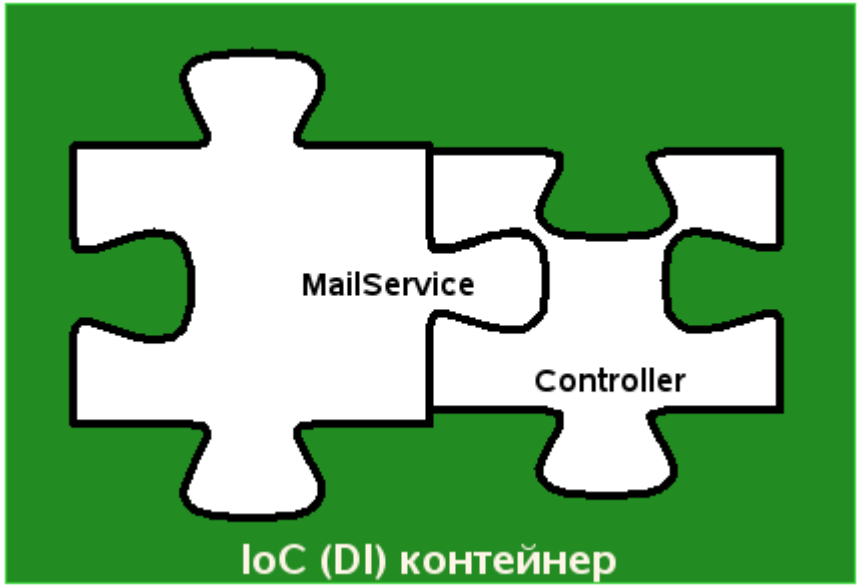
Spring Framework :: IoC / DI

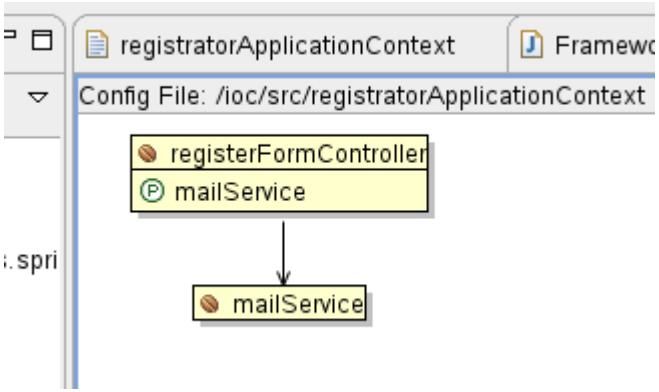
- **Решение – Inversion of Control контейнер**

```
// resolve all dependencies
A aobj = container.resolve(A.class);
```

- Похож на Service Locator;
- При запросе объекта какого-либо типа контейнер решает – кого вернуть;
- Для каждого типа, зарегистрированного в контейнере существует карта зависимостей (т.е. описание – что передавать в конструктор, каким свойствам что присваивать и т.д.);
- Содержит ассоциации – для какого запрошенного идентификатора какой объект вернуть;
- Для каждой зависимости запрошенного объекта при его создании контейнер создает (или подставляет уже созданный) соответствующий объект, у которого также рекурсивно разрешаются зависимости;
- Управляет временем жизни создаваемых объектов;
- Потенциально, IoC контейнер можно использовать как Service Locator, но этого надо по возможности избегать.

Spring Framework :: IoC / DI





Spring Framework :: IoC / DI

Преимущества IoC контейнеров:

- Управление зависимостями и применение изменений без перекомпиляции;
- Упрощение повторного использования классов или компонентов;
- Упрощение unit-тестирования;
- Более "чистый" код (классы не инициализируют вспомогательные объекты);
- В IoC контейнер лучше всего выносить те интерфейсы, реализация которых может быть изменена в текущем проекте или в будущих проектах.

Spring Framework :: Семейство IoC контейнеров

- BeanFactory – базовый интерфейс, представляющий IoC контейнер в Spring Framework (используемая реализация: **XmlBeanFactory**):
  - BeanFactory предоставляет только базовую низкоуровневую функциональность.
- ApplicationContext – интерфейс, расширяющий BeanFactory и добавляющий различную функциональность к базовым возможностям контейнера:
  - простота интеграции со Spring AOP;
  - работа с ресурсами и сообщениями;
  - обработка событий;
  - Специфические контексты приложений (как, например, WebApplicationContext);
- В реальной жизни используются в основном ApplicationContext-ы
- BeanFactory может быть использован в исключительных случаях, например, при интеграции Spring со своим фреймворком или в случаях, когда ресурсы критичны, а ничего кроме IoC контейнера не требуется.

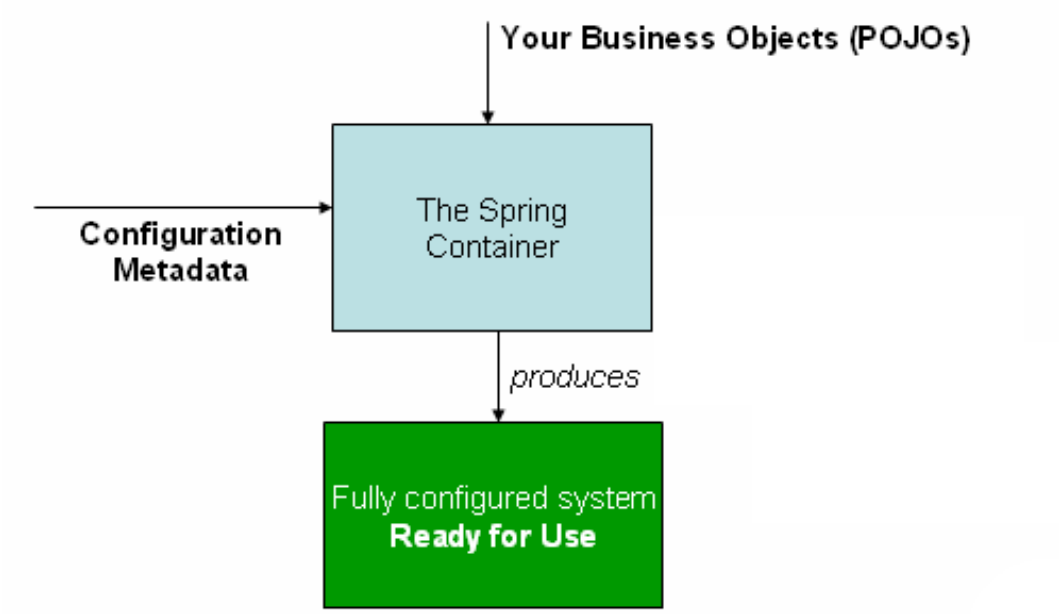
Spring Framework :: Семейство IoC контейнеров

LUXOFT TRAINING

- Существует несколько реализаций ApplicationContext, доступных для использования. Основными являются:
  - `GenericXmlApplicationContext` (since v.3.0);
  - `ClassPathXmlApplicationContext`;
  - `FileSystemXmlApplicationContext`;
  - `WebApplicationContext`;
- XML является традиционным способом задания конфигурации контейнера, хотя существуют и другие способы задания метаданных (аннотации, Java код и т.д.);
- Во многих случаях проще и быстрее конфигурировать контейнер с помощью аннотаций. Но надо помнить, что аннотированные конфигурации содержат некоторые ограничения и вносят дополнительные зависимости на уровене кода;
- В большинстве случаев пользователю (разработчику) не придется самому инициализировать Spring IoC контейнер;

Spring Framework :: Работа с IoC контейнером

В общем виде, работа IoC контейнера Spring может быть представлена в виде следующей диаграммы:



- В процессе создания и инициализации контейнера классы вашего приложения объединяются с метаданными (конфигурацией контейнера) и на выходе вы получаете полностью сконфигурированное и готовое к работе приложение.

Spring Framework :: Работа с IoC контейнером

- Создание контейнера

```
ApplicationContext ctx =
  new ClassPathXmlApplicationContext("services.xml");
```

Spring Framework :: Работа с IoC контейнером

LUXOFT TRAINING

- Пример конфигурации:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <bean id="myService" class="foo.bar.ServiceImpl">
   <property name="param1" value="some value" />
   <property name="otherBean" ref="otherBeanService" />
  </bean>

  <bean id="otherBeanService" class="..."/>

  <!-- more bean definitions go here -->
</beans>
```

Spring Framework :: Beans как компоненты

LUXOFT TRAINING

- В терминологии Spring, любой Java объект, который управляется контейнером называется “Bean”;
- Bean – это обычный Java объект (POJO);
- Необязательно наличие default (без аргументов) конструктора;

- При конфигурации можно задать:
  - Уникальный идентификатор (ID) бина;
  - Полное имя класса;
  - Поведение бина внутри контейнера (scope, lifecycle callbacks, и т.д.);
  - Зависимости от других бинов;
  - Прочие параметры (например, размер пула соединений).

Spring Framework :: Beans как компоненты

- При помощи конструктора:

```
<bean id="example1"
    class="ru.luxoft.training.samples.Example" />
```

- При помощи статического фабричного метода:

```
<bean id="clientService"
    class="ru.luxoft.training.samples.ClientService"
    factory-method="createInstance" />
```

- При помощи не статического фабричного метода

```
<bean id="serviceFactory"
    class="examples.DefaultServiceFactory" />

<bean id="clientService"
    factory-bean="serviceFactory"
    factory-method="createClientServiceInstance" />
```

Spring Framework :: отложенная инициализация

- Для конкретного бина:

```
<bean id="lazy" class="…" lazy-init="true" />
```

- Для всех бинов в контейнере:

```
<beans default-lazy-init="true">
…
</beans>
```

- Если singleton - бин зависит от lazy - бина, то lazy - бин создастся сразу, при создании singleton - бина.

Spring Framework :: внешние зависимости

- **Единственная жесткая внешняя зависимость фреймворка – Jakarta Commons Logging API (JCL)**

- Зависимость разрешается в runtime

- Альтернативы использованию JCL – SLF4J:
  - Убрать commons.logging из зависимостей и из classpath
  - Добавить SLF4J-JCL в classpath (jcl-over-slf4j)
  - Добавить SLF4J-API в classpath

- Для использования Log4J достаточно:
  - Добавить log4j библиотеку в classpath
  - Добавить log4j конфигурационный файл (log4j.properties или log4j.xml) в корень classpath

Упражнения

- №3: “Hello, World” пример для Spring Framework:
  - 20 мин – самостоятельная работа;
  - 10 мин – обсуждение;

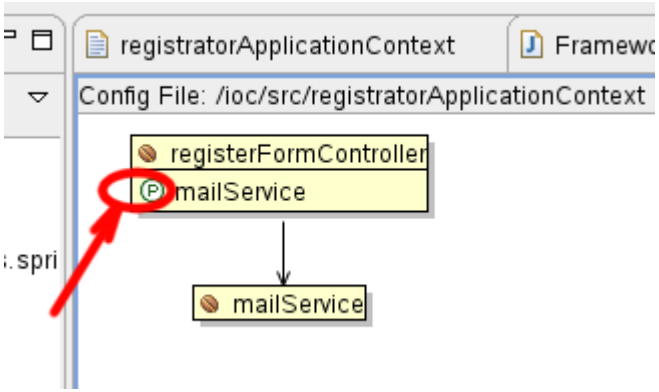
Spring Framework :: Создание псевдонимов

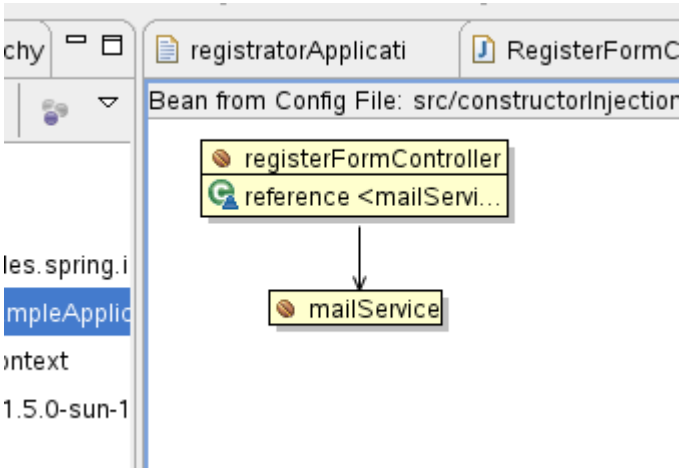
```
<alias fromName="originalName" toName="aliasName" />
```

- После такой инструкции бин с именем originalName будет также доступен под именем aliasName;
- Такая необходимость часто возникает, когда архитектура приложения изначально создана с учетом возможности расширения, но при этом пока в конкретных разделах такой необходимости не возникает (и, соответственно, нет смысла плодить дополнительные объекты).

Spring Framework :: DI

- Существует два основных типа внедрения зависимостей (DI):
  - Внедрение зависимости через конструктор
  - Внедрение зависимости через set-метод





Spring Framework :: Constructor DI

```
public class ConstructorInjection {
   private Dependency dep;
   private String descr;

   public ConstructorInjection(Dependency dep, String descr) {
      this.dep = dep;
      this.descr = descr;
   }
}

<bean id="dependency" class="Dependency" />
<bean id="constrInj" class="ConstructorInjection">
   <constructor-arg ref="dependency" />
   <constructor-arg value="Constructor DI" />
</bean>
```

Spring Framework :: Constructor DI

- Циклическая зависимость:

```
class A {
    private B b;
    A(B b) {
        this.b = b;
    }
}
```

```
class B {
    private A a;
    B(A a) {
        this.a = a;
    }
}
```

- При Constructor DI для этих классов – *BeanCurrentlyInCreationException*
- Решение – в одном или обоих классах заменить Constructor DI на Setter DI

Spring Framework :: Setter DI

```
public class SetterInjection {
   private Dependency dep;
   private String descr;

   public void setDep(Dependency dep) {
      this.dep = dep;
   }
   public void setDescr(String descr) {
      this.descr = descr;
   }
}

<bean id="dependency" class="Dependency" />
<bean id="setterInj" class="SetterInjection">
   <property name="dep" ref="dependency" />
   <property name="descr" value="Setter DI" />
</bean>
```

Spring Framework :: Autowiring

LUXOFT TRAINING

- Spring может автоматически связывать (добавлять зависимости) между бинами вместо <ref>;

- В некоторых случаях это может существенно сократить объем затрат на конфигурирование контейнера;
- Позволяет автоматически обрабатывать изменения в связи с расширением объектной модели (например, при добавлении новых зависимостей они подключатся автоматически);
- Связывание по типу может работать, когда доступен только один бин определенного типа;
- Менее понятно для чтения и прослеживания зависимостей, чем явное задание зависимостей;

- Задается с помощью атрибута autowire в определении бина

```
<bean id="" class="" autowire="value" />
```

Spring Framework :: Autowiring

LUXOFT TRAINING

- Типы автоматического связывания:
  - **no** – запрет на автосвязывание – значение по умолчанию;
  - **byName** – автосвязывание по имени свойства. Контейнер будет искать бин с ID, совпадающим с именем свойства. Если такой бин не найден – объект остается несвязанным;
  - **byType** – автосвязывание по типу параметра. Работает только в случае наличия единственного экземпляра бина соответствующего класса в контейнере. Если более одного бина – **UnsatisfiedDependencyException**;
  - **constructor** – контейнер ищет бин (или бины) совпадающие по типу с параметрами конструктора. Если более одного бина одного типа или более одного конструктора – **UnsatisfiedDependencyException**;

Spring Framework :: Использование аннотаций

LUXOFT TRAINING

- Контейнер Spring также может быть сконфигурирован с использованием аннотаций;

- Основные типы поддерживаемых аннотаций:
  - @Required
  - @Autowired
  - @Component

- Для поддержки конфигурации через аннотации, в конфигурации Spring контейнера должно быть указано следующее свойство:

```
<context:annotation-config/>
```

Spring Framework :: Использование аннотаций

LUXOFT TRAINING

@Required

- Применяется только к SET методам бинов;
- Определяет что соответствующее свойство бина должно быть вычислено на этапе конфигурации (через конфигурацию или автоматическое связывание);
- Если соответствующее свойство не может быть задано – контейнер сгенерирует соответствующее исключение, что позволит избежать «неожиданных» NullPointerException в процессе работы системы;

```
public class SimpleMovieLister {
   private MovieFinder movieFinder;

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

Spring Framework :: Использование аннотаций

LUXOFT TRAINING

@Autowired

- Применяется к:
  - SET методам бинов;
  - Конструкторам;
  - Методам с несколькими параметрами;
  - Свойствам (в том числе, приватным);
  - К массивам и типизированным коллекциям (будут привязаны ВСЕ бины соответствующего класса)

- Возможно использование с @Qualifier(“name”) – в таком случае будет автоматически привязан бин с соответствующим ID;

- По-умолчанию генерируется исключение если не найден ни один подходящий бин. Это поведение может быть изменено с помощью @Autowired(required=false);

Spring Framework :: Использование аннотаций

LUXOFT TRAINING

@Component

- Используется для задания Spring компонент без использования XML конфигурации
- Применяемся к классам

- Является базовым стереотипом для любого Spring-managed компонента
- Рекомендуется использовать более точные стереотипы*:
  - **@Service**
  - **@Repository**
  - **@Controller**
- В большинстве случаев, если вы не уверены, какой именно стереотип использовать – используйте @Service
- Для автоматической регистрации бинов через аннотации необходимо указать следующую инструкцию в конфигурации контейнера:

```
<context:component-scan base-package="org.example"/>
```

* - история возникновения стереотипов их назначение берут свое начало в книге Эрика Эванса «Domain-Driven Design: Tackling Complexity in the Heart of Software»

Spring Framework :: области видимости бинов

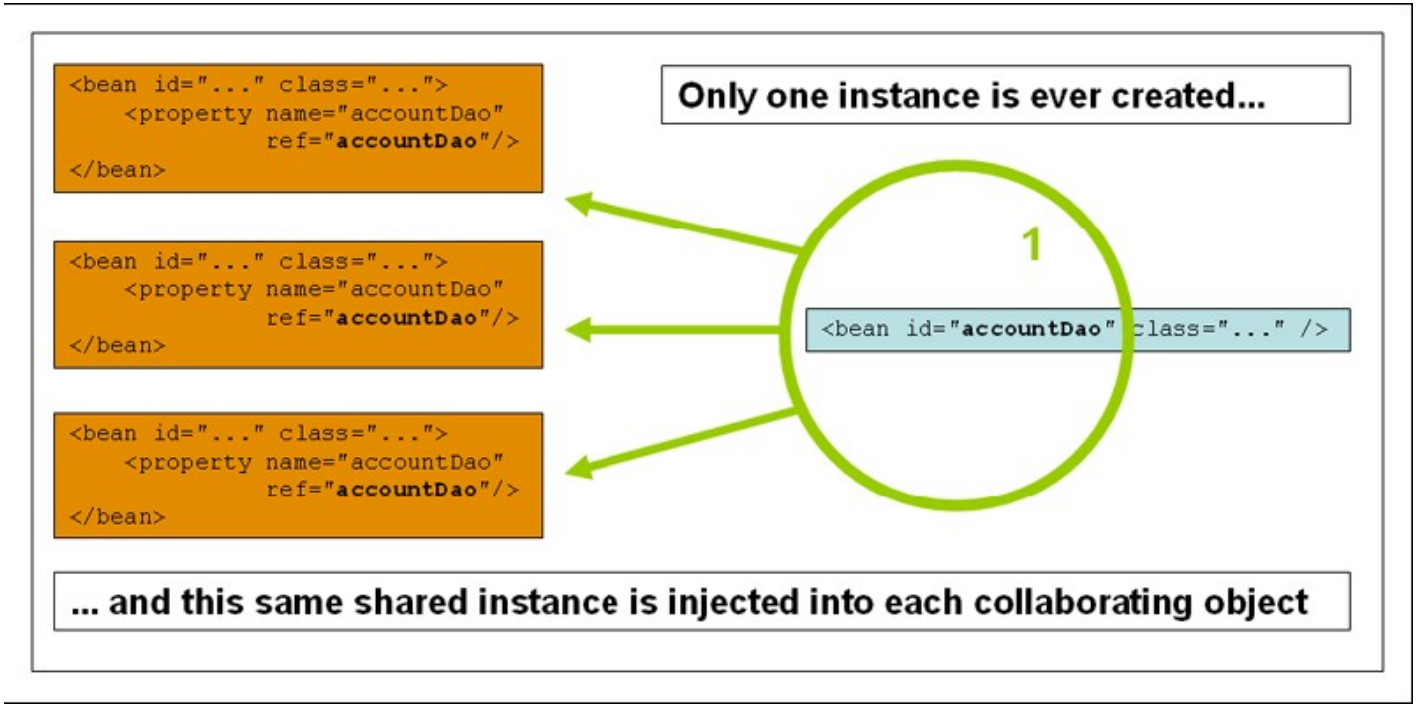
LUXOFT TRAINING

Bean Scope

- Общие: Область видимости бинов
  - Singleton
  - Prototype
- Специфичные для Web
  - Request
  - Session
  - Global session

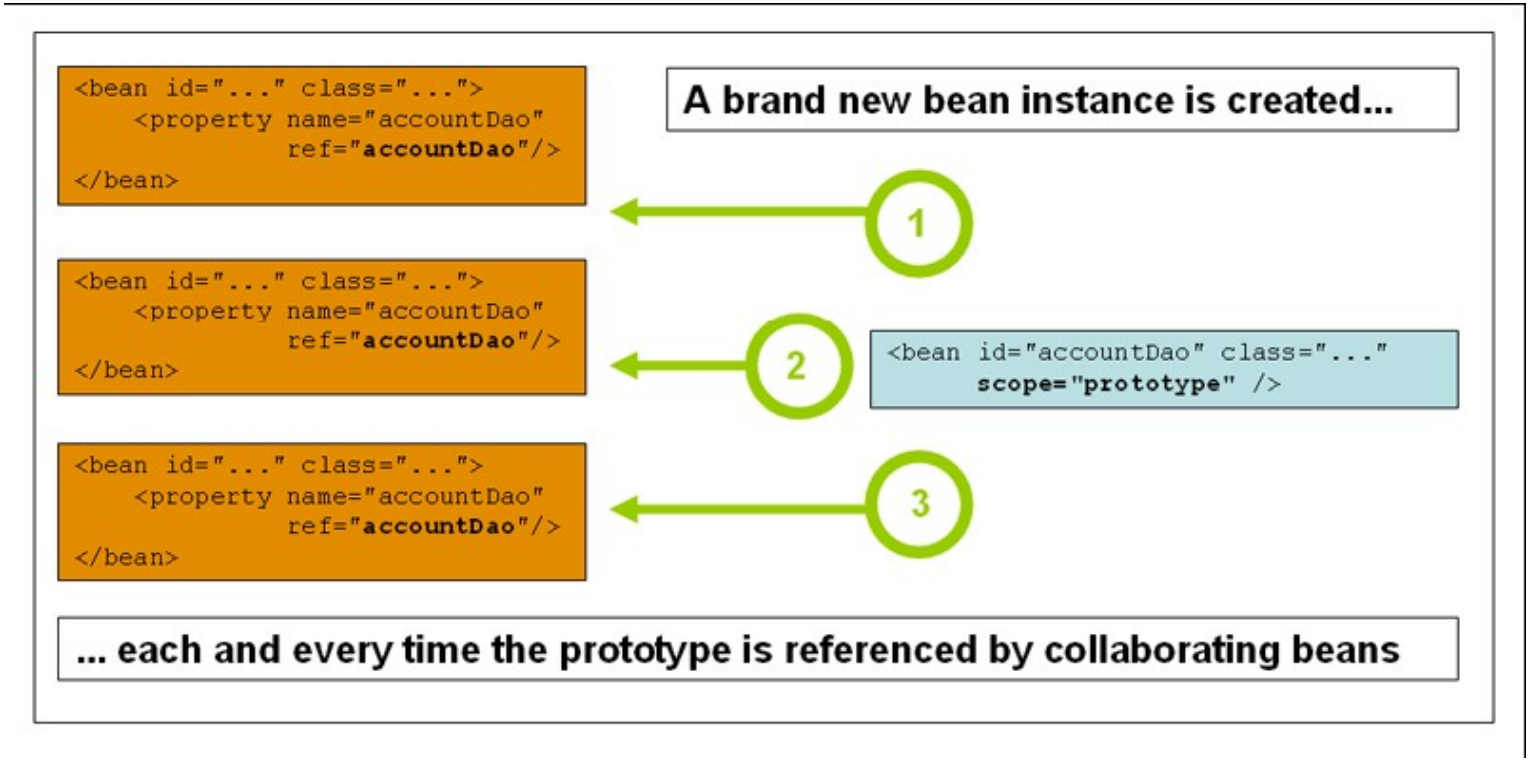
Spring Framework :: области видимости бинов

- **Singleton**
  - По-умолчанию
  - Один экземпляр бина в контейнере

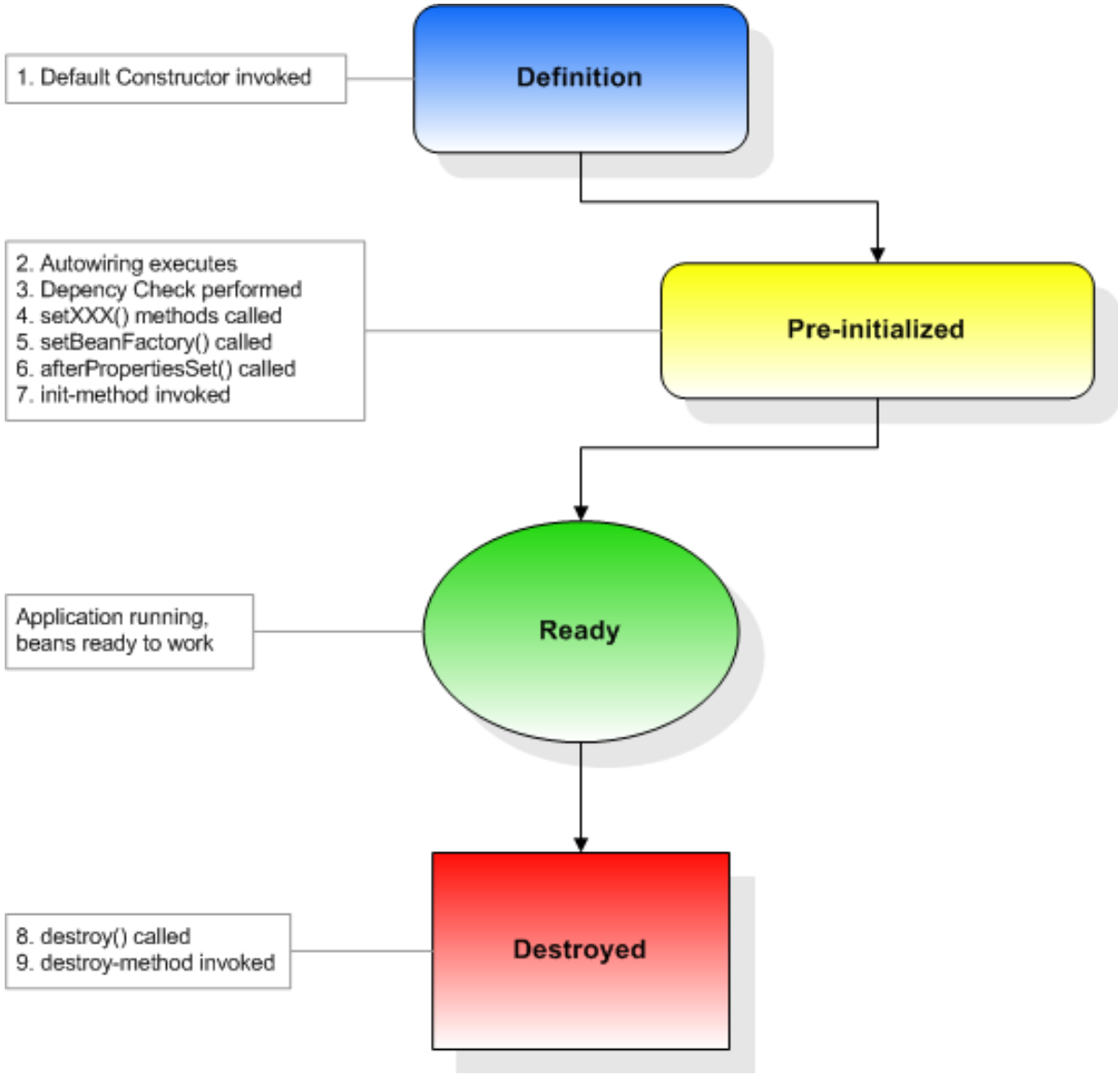


Spring Framework :: области видимости бинов

- **Prototype**
  - Каждый раз при внедрении в другой бин или при вызове метода getBean() создается новый экземпляр бина



Spring Framework :: жизненный цикл бина



Spring Framework :: жизненный цикл бина

- Создается BeanFactory или ApplicationContext;
- У каждого бина есть зависимости в виде свойств или аргументов конструктора;
- Каждое свойство или аргумент конструктора задано или точным значением, или ссылкой на другой бин;
- Должна быть возможность конвертации значений свойств или аргументов конструктора из формата в котором они заданы в нужный формат (см. PropertyEditor);
- Контейнер валидирует конфигурацию каждого бина;
- Свойства инициализируются в тот момент, когда бин реально создан;
- Singleton бины создаются в момент создания конструктора (если не lazy);
- Остальные – только при обращении к ним.

Spring Framework :: жизненный цикл бина

Управление бином, реализуя интерфейсы из Spring

- Создание
  - Реализовать интерфейс InitializingBean
  - Переопределить метод afterPropertiesSet()
- Удаление
  - Реализовать интерфейс DisposableBean
  - Переопределить метод destroy()

Spring Framework :: жизненный цикл бина

Управление бином без зависимости от Spring в коде

- В нужный бин добавить методы для инициализации и/или удаления и указать их в объявлении бина:

```
<bean id="example" class="Example"
  init-method="init"
  destroy-method="cleanup" />
```

- Можно задать методы для создания и/или удаления для всех бинов внутри контейнера:

```
<beans default-init-method="init"
  default-destroy-method="cleanup">
```

Spring Framework :: Доп.возможности ApplicationContext

- Чтобы получить доступ к контексту (например, для публикации своих событий) достаточно у бина имплементировать интерфейс ApplicationContextAware

```
public class CommandManager implements ApplicationContextAware {

   private ApplicationContext applicationContext;

   public void setApplicationContext(
        ApplicationContext applicationContext)
        throws BeansException
   {
      this.applicationContext = applicationContext;
   }
}
```

Spring Framework :: Доп.возможности ApplicationContext

LUXOFT TRAINING

- Обработка событий внутри ApplicationContext обеспечивается при помощи
  - Класса ApplicationEvent
  - Интерфейса ApplicationListener

- При наступлении события нотифицируются все бины, зарегистрированные в контейнере и реализующие интерфейс ApplicationListener

- ApplicationEvent – основные реализации:
  - **ContextRefreshedEvents** – создание или обновление ApplicationContext
    - Синглетоны созданы
    - ApplicationContext готов к использованию
  - **ContextClosedEvent**
    - после использования close() метода
  - **RequestHandledEvent**
    - только для веб приложения

Spring Framework :: Доп.возможности ApplicationContext

LUXOFT TRAINING

- Получение стандартных событий:

```
public class MyBean implements ApplicationListener {
   Public void onApplicationEvent(ApplicationEvent event) {
      …
   }
}
```

- Публикация собственных событий*:

```
public class CustomEvent extends ApplicationEvent {
   public CustomEvent (Object obj) {
      super(obj);
   }
}
context.publishEvent(new CustomEvent(new Object()));
```

* - As of Spring 3.0, an ApplicationListener can generically declare the event type that it is interested in. When registered with a Spring ApplicationContext, events will be filtered accordingly, with the listener getting invoked for matching event objects only. Interface ApplicationListener<E extends ApplicationEvent>

Spring Framework :: Доп.возможности ApplicationContext

LUXOFT TRAINING

- Интерфейс ApplicationContext наследует интерфейс MessageSource и, соответственно, предоставляет функциональность интернационализации (i18n)

- При загрузке автоматически ищет MessageSource бин в конфигурации (бин должен наследоваться от MessageSource и иметь id=“messageSource”)

- Если такой бин не может быть найден нигде в контексте – ApplicationContext создает экземпляр «заглушки» - DelegatingMessageSource для корректной обработки соответствующих методов

Spring Framework :: Доп.возможности ApplicationContext

LUXOFT TRAINING

Пример конфигурации Message Source:

```
<bean id="messageSource"
  class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <value>format</value>
      <value>exceptions</value>
      <value>windows</value>
    </list>
  </property>
</bean>
```

- Содержание format_**en_GB**.properties

```
message=Alligators rock!
```

- Содержание format_**ru_RU**.properties

```
message=Крокодилы – нереально круты!
```

Spring Framework :: Инициализция коллекций

LUXOFT TRAINING

```
<bean id="..." class="...">
<property name="someList">
  <list>
      <value>a list element followed by a reference</value>
      <ref bean="myDataSource" />
  </list>
</property>
<!-- results in a setSomeMap(java.util.Map) call -->
<property name="someMap">
  <map>
      <entry key="an entry" value="just some string"/>
      <entry key ="a ref" value-ref="myDataSource"/>
  </map>
</property>
<!-- results in a setSomeSet(java.util.Set) call -->
<property name="someSet">
  <set>
      <value>just some string</value>
      <ref bean="myDataSource" />
  </set>
</property>
</bean>
```

Упражнения

- №4: Разработка простейшего приложения:
  - 50 мин – самостоятельная работа;
  - 10 мин – обсуждение;

Вопросы!?