

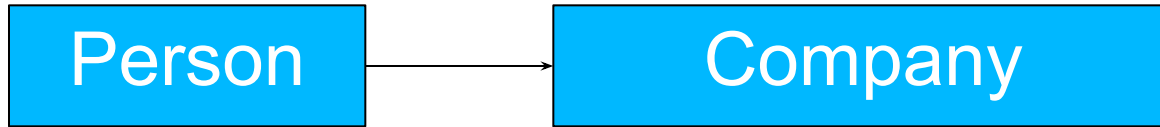


Spring Framework

Module 2 – Components Model (IoC, DI)

Evgeniy Krivosheev
Andrey Stukalenko
Vyacheslav Yakovenko
Vladimir Sonkin
Last update: Feb, 2013

Spring Framework :: Связи между объектами



Традиционный подход

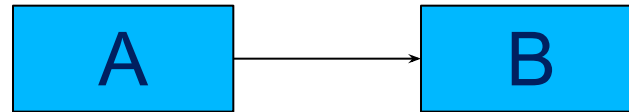
```
class Person {
    public String name;
    public Company company;

    public Person() {
        name = "Иван Иванов";
        Company company = new Company();
        company.name = "Luxoft";
    }
}

class Company {
    public String name;
}
```

Spring Framework :: Связи между объектами

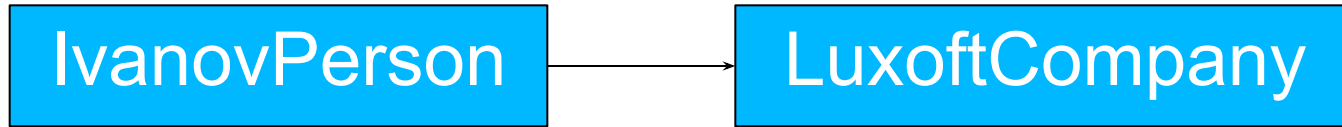
Традиционный подход



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 :: Связи между объектами



Подход с использованием паттерна Singleton

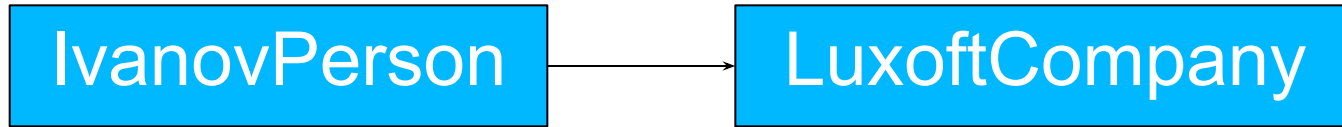
```
class Person {  
    public String name;  
    public Company company;  
}
```

```
class Company {  
    public String name;  
}
```

```
class IvanovPerson extends Person {  
    public Person ivanovPerson = new Person();  
    public static Person create() {  
        ivanovPerson.name = "Иван Иванов";  
        ivanovPerson.company =  
        LuxoftCompany.create();  
        return ivanovPerson;  
    }  
}
```

```
class LuxoftCompany extends Company {  
    public Company luxoftCompany = new  
    Company();  
    public LuxoftCompany() {  
        luxoftCompany = "Luxoft";  
    }  
    public static Company create() {  
        return luxoftCompany;  
    }  
}
```

Spring Framework :: Связи между объектами



Подход с использованием паттерна Singleton

- Отдельный класс специально под нашу задачу
- В коде **IvanovPerson.create()** стоит прямая ссылка на этот класс
- В случае перевода Иванова в другую компанию, надо менять этот код
- Для тестирования невозможно «на время» подменить компанию

Spring Framework :: Связи между объектами



Person

Company

Подход с использованием IoC

POJO – plain old Java Object

```
class Person {  
    public String name;  
    public Company company;  
}
```

```
class Company {  
    public String name;  
}
```

```
class BankApplication {  
    @Autowired  
    @Required  
    private CR companyReport;  
    public void setCompanyReport();  
}
```

application-context.xml

```
<beans autowire=byName>  
    <bean id="ivanov" class="Person">  
        <property name="name" value="Иван Иванов"/>  
        <property name="company" ref="luxoftCompany"/>  
    </bean>  
    <bean id="luxoftCompany" class="Company">  
        <property name="name" value="Luxoft"/>  
    </bean>  
    <bean id="companyReport" class="CompanyReport">  
        <property name="company" ref="luxoftCompany"/>  
    </bean>  
    <bean id="bankApplication" class="BankApplication">  
        <property name="companyReport"  
            ref="companyReport"/>  
    </bean>
```

Spring Framework :: Связи между объектами



Person

Company

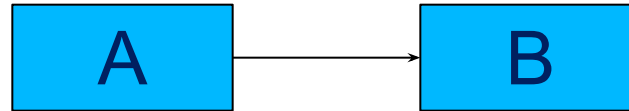
Подход с использованием IoC

Преимущества:

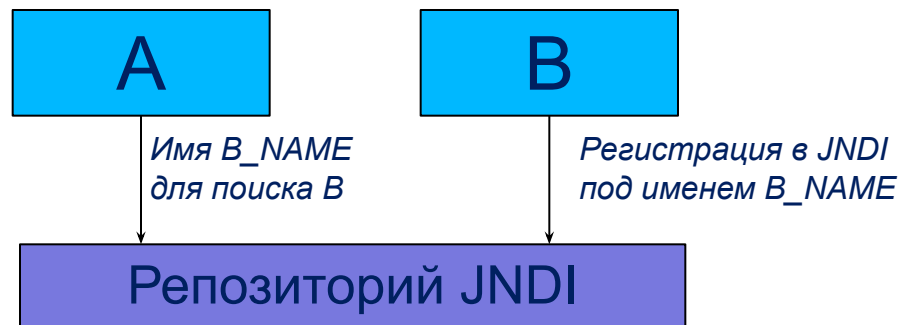
- контейнер создает необходимые объекты и управляет их временем жизни
- Person и Company не связаны друг с другом и независимы от внешних библиотек
- application-context документирует систему и связи между объектами
- легкость внесения изменений в связи системы

Spring Framework :: Связи между объектами

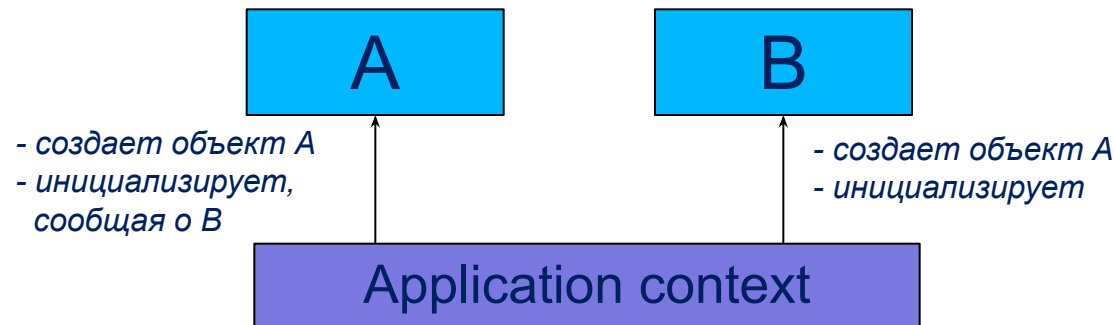
Традиционный подход: связи между объектами внутри кода



Паттерн Service Locator (JNDI в JEE): объекты в репозитории



IoC: объекты ничего не знают друг о друге



```
class A {  
    private B b;  
}
```

```
class B {  
}
```


Spring Framework :: IoC

Инверсия управления (Inversion of Control, IoC) — принцип объектно-ориентированного программирования, используемый для уменьшения связанности объектов.

- Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба должны зависеть от абстракции.
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Техники реализации:

- Фабричный метод (англ. Factory pattern)
- Service locator (англ. Service locator pattern)
- Внедрение зависимости (англ. Dependency injection)
 - Через метод класса (англ. Setter injection)
 - Через конструктор (англ. Constructor injection)
 - Через интерфейс внедрения (англ. Interface injection)
- IoC контейнер (англ. IoC-container)

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;
 - поддержка интернационализации;
 - Специфические контексты приложений (как, например, **WebApplicationContext**);

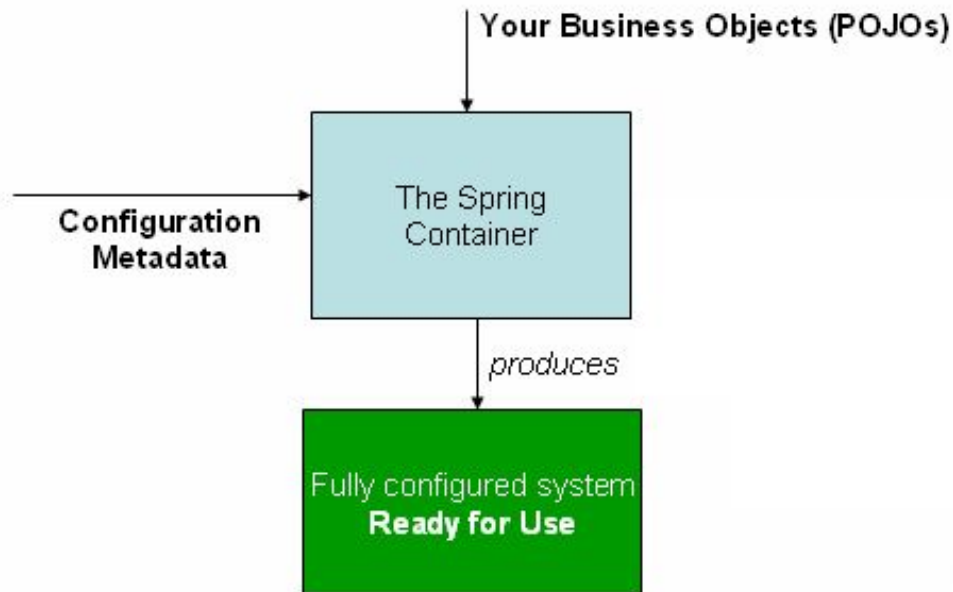
Spring Framework :: Семейство IoC контейнеров



- 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 уровне 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:

```
public void main() {  
    ApplicationContext context =  
        new ClassPathXmlApplicationContext("application-context.xml"  
BankApplication bankApplication =  
        context.getBean("bankApplication");  
}
```

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

Spring Framework :: Работа с IoC контейнером



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 :: Создание Bean

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="serviceLocator"  
factory-method="createClientServiceInstance" />
```


Spring Framework :: Отложенная инициализация



- Для конкретного бина:

```
<bean id="lazy" class="ru.luxoft.training.ClientService"  
      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.

Упражнения

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

Spring Framework :: Импорт контекста

Часто удобно разбивать контекст на несколько файлов:

```
<beans>
```

```
  <import resource="services.xml"/>
```

```
  <import resource="resources/messageSource.xml"/>
```

```
  <import resource="/resources/themeSource.xml"/>
```

```
  <bean id="bean1" class="..."/>
```

```
  <bean id="bean2" class="..."/>
```

```
</beans>
```

Spring Framework :: Подключение property-файлов к xml-контексту



```
<bean
  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations" value="classpath:com/foo/jdbc.properties" />
</bean>
```

```
<bean id="dataSource" destroy-method="close"
  class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="${jdbc.driverClassName}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>
```

jdbc.properties:

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsq1://production:9002
jdbc.username=sa
jdbc.password=root
```

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

Внедрение зависимости через конструктор

```
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

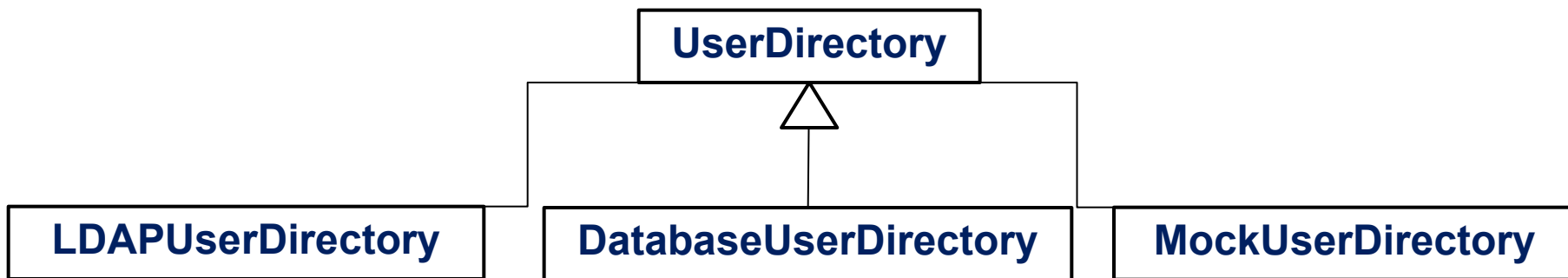
```
public class SetterInjection {  
    private Dependency dep;  
    private String descr;  
  
    public void setDep(Dependency dep) {  
        this.dep = dep;  
    }  
    public void setDescr(String descr) {  
        this.descr = descr;  
    }  
}
```

```
<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

Пример: сервисный класс для получения информации о пользователях



Пусть есть классы, которым нужна информация о пользователях:

```

class LoginManager {
  UserDirectory userDirectory;
}
class UserDirectorySearch {
  UserDirectory userDirectory;
}
class UserInfo {
  UserDirectory userDirectory;
}

<bean id="userDirectory" class="LDAPUserDirectory" />
<bean id="loginManager" class="LoginManager">
  <property name="userDirectory" ref="userDirectory"/>
</bean>
<bean id="userDirectorySearch"
      class="UserDirectorySearch">
  <property name="userDirectory" ref="userDirectory"/>
</bean>
<bean id="userInfo" class="UserInfo">
  <property name="userDirectory" ref="userDirectory"/>
</bean>
  
```

Spring Framework :: Autowiring

Теперь включим автоматическое связывание (autowire)

```
class LoginManager {  
    UserDirectory userDirectory;  
}
```

```
class UserInfo {  
    LDAPUserDirectory ldapUserDirectory;  
}
```

```
class UserDirectorySearch {  
    UserDirectory userDirectory;  
}
```

Свойство **userDirectory** автоматически инициализируется:

```
<bean id="userDirectory" class="LDAPUserDirectory" />
```

```
<bean id="loginManager" class="LoginManager" autowire="byName">  
</bean>
```

```
<bean id="userDirectorySearch" class="UserDirectorySearch"  
autowire="byName">  
</bean>
```

```
<bean id="userInfo" class="UserInfo" autowire="byType">  
</bean>
```

Spring Framework :: Autowiring

- 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 (магия!);
- **Задается с помощью атрибута `autowire` в определении бина**

```
<bean id="..." class="..." autowire="no|byName|byType|constructor" />
```

Spring Framework :: Autowiring

- 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



- 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/>
```

@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;  
}  
}
```

@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



@Resource

```
public class SimpleMovieLister {  
    private MovieFinder movieFinder;  
  
    @Resource(name="myMovieFinder")  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
}
```


@Component

- Используется для задания Spring компонент без использования XML конфигурации
- Применяется 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"/>
```

Spring Framework :: Использование аннотаций



Пример использования компонентов:

```
package com.luxoft.calculator;  
@Service("adder")  
public class Adder {  
    public int add(int a, int b) {  
        return a + b;  
    }  
}
```

```
package com.luxoft.calculator;  
@Component("calculator")  
public class Calculator {  
    @Autowired  
    private Adder adder;  
  
    public void makeAnOperation() {  
        int r1 = adder.add(1,2);  
        System.out.println("r1 = " + r1);  
    }  
}
```

application_context.xml:

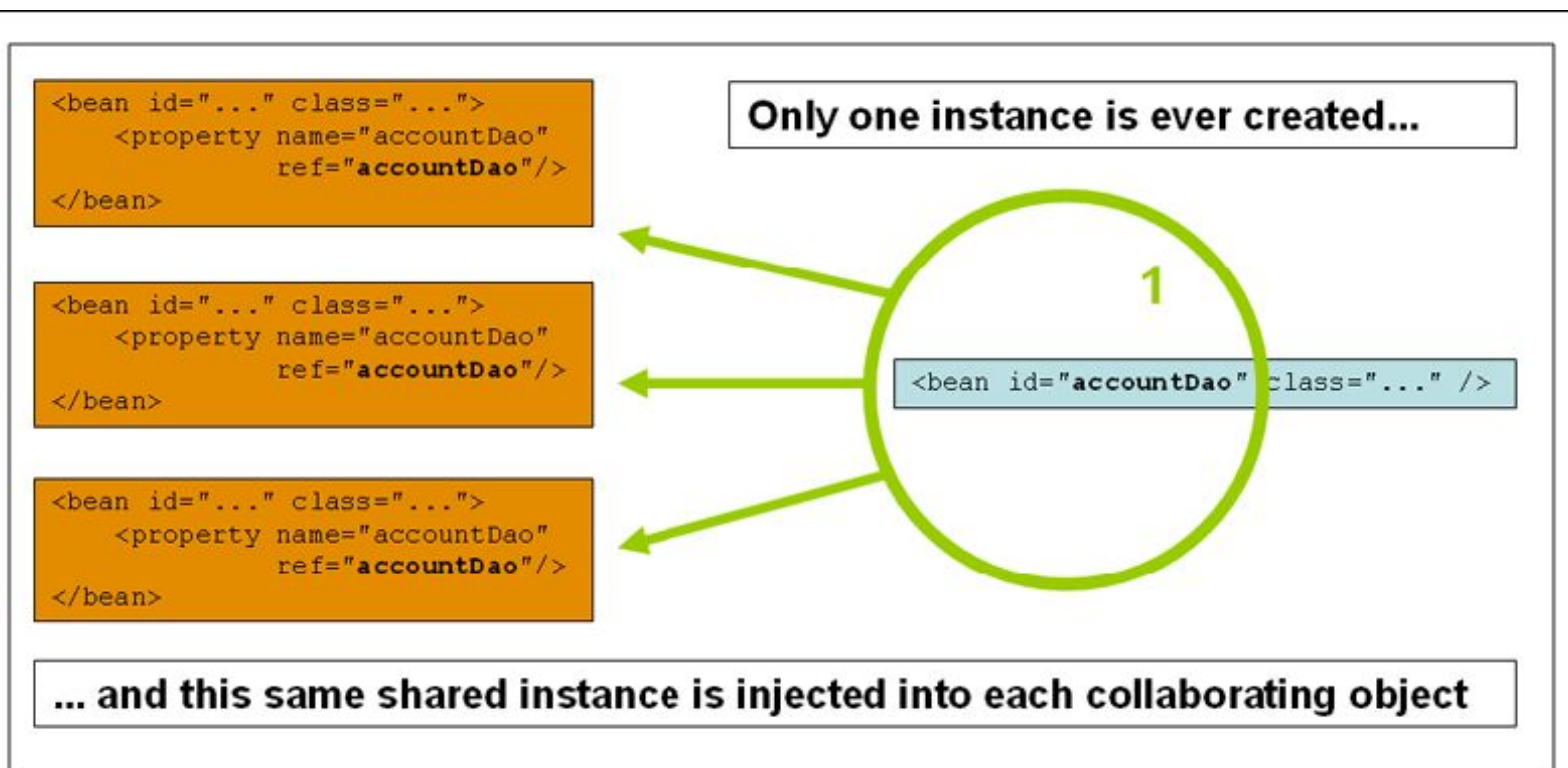
```
<context:component-scan base-package="com.luxoft.calculator"/>  
<bean id="adder" class="Adder"/>
```

Spring Framework :: scope бинов

- Singleton

- По-умолчанию
- Один экземпляр бина в контейнере

```
<bean name="single" class="Single" scope="singleton" />
```

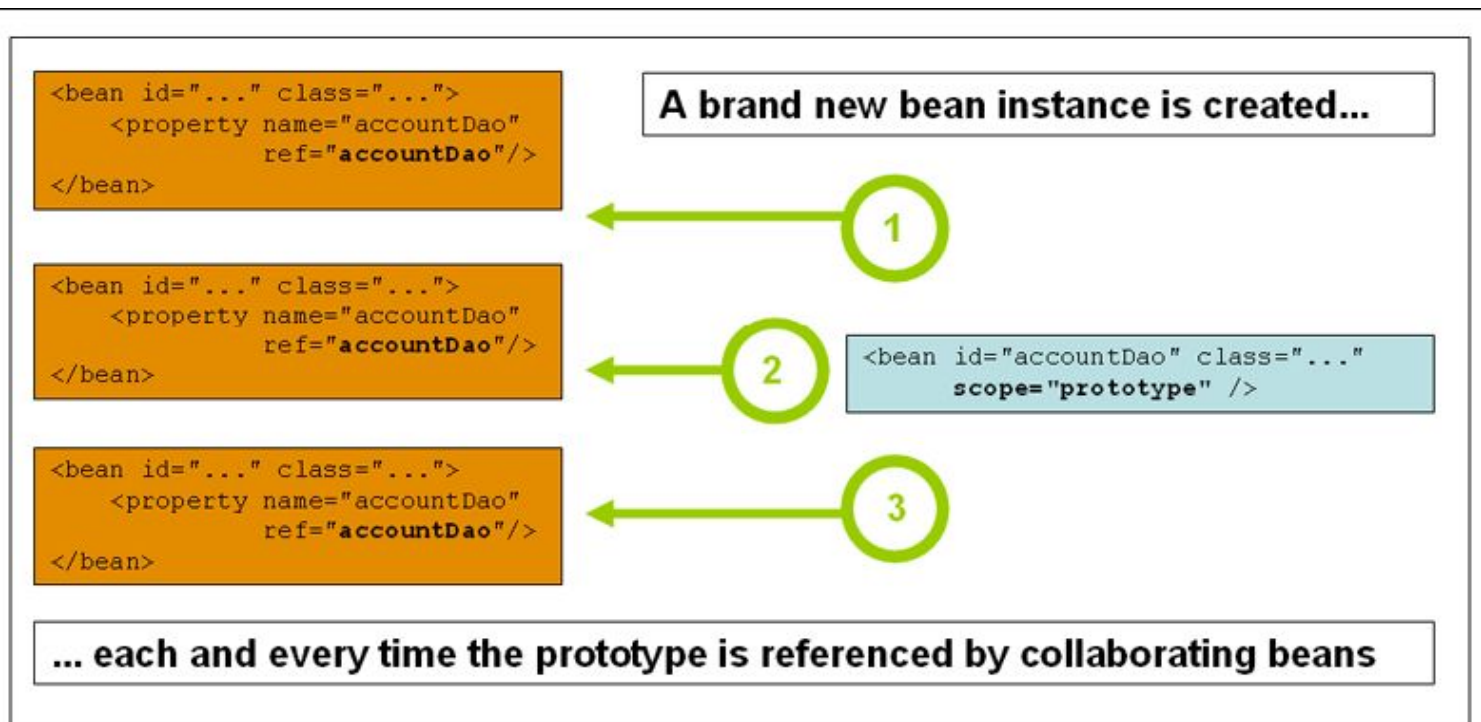


Spring Framework :: scope бинов

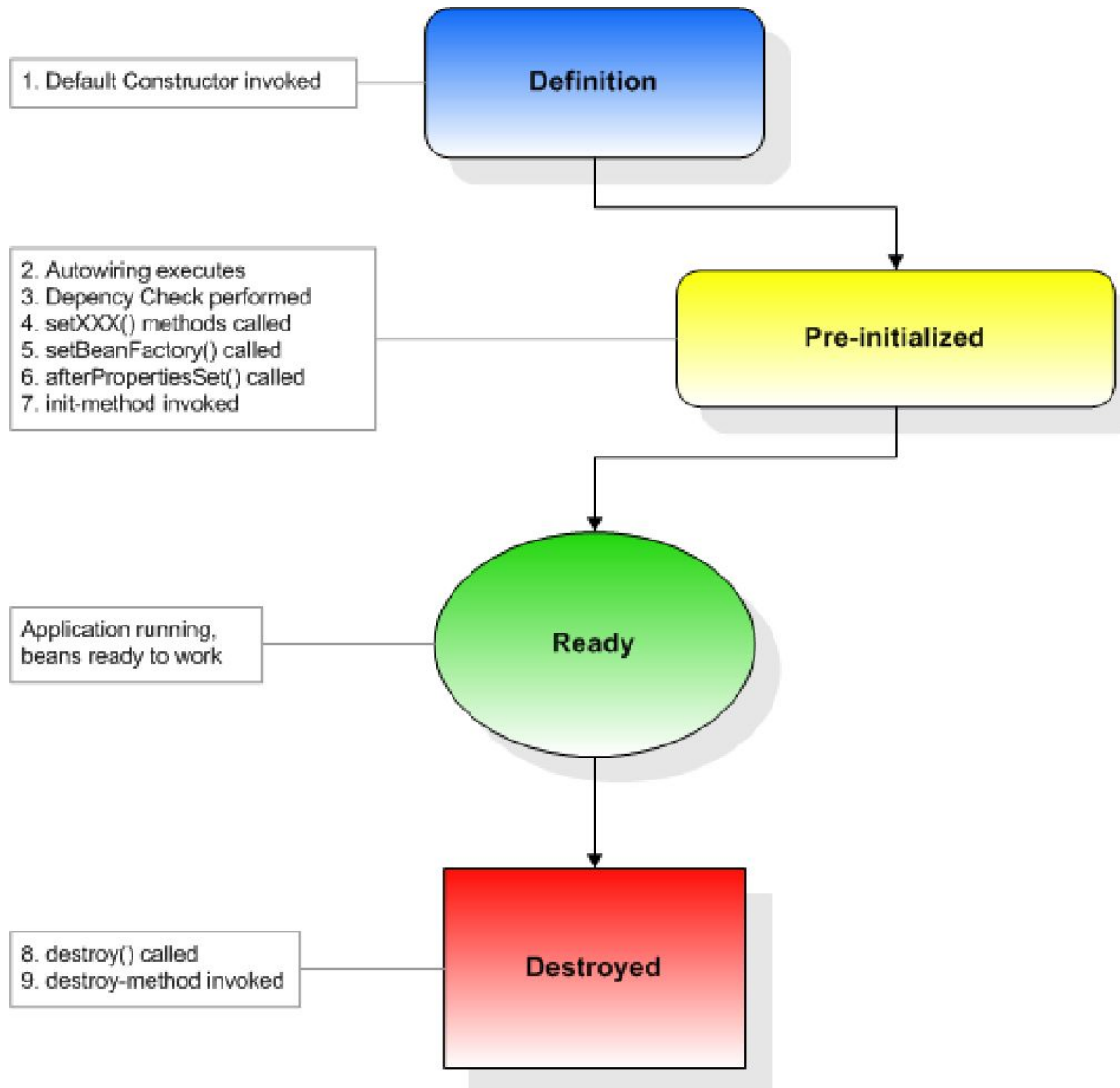
■ Prototype

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

```
<bean name="proto" class="Prototype" scope="prototype" />
```



Spring Framework :: Жизненный цикл бина



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 :: Доступ к 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 :: События

- 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()));
```

Spring Framework :: События

- 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 :: События

Пример: регистрация нового сотрудника в компании.

Возможные получатели события:

- оповещение охранников, чтобы сделали пропуск
- охранники подписываются на событие
- дополнительно могут подписаться бухгалтерия, отдел кадров
- допустим надо добавить новую функциональность:
регистрировать новых сотрудников в базе данных
 - для этого нам достаточно создать класс для регистрации и подписаться на событие добавления сотрудника

Преимущества:

- Получателей может быть как угодно много;
- Добавление получателя не добавляет зависимости: о получателе знает только он сам

Недостатки:

- Приводит к неявному поведению приложения

Spring Framework :: Локализация

- 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 :: Локализация

`messages_en_US.properties`

```
customer.name=Ivan Ivanov, age : {0}, URL : {1}
```

`messages_ru_RU.properties`

```
customer.name=Иван Иванов, возраст : {0}, URL : {1}
```

Папка для расположения файлов:

```
resources\locale\customer\
```

`Locale.xml` :

```
<bean id="messageSource"  
  class="org.springframework.context.support.ResourceBundleMessageSource">  
  <property name="basename">  
    <value>locale\customer\messages</value>  
  </property>  
</bean>
```

Spring Framework :: Локализация

messages_en_US.properties

```
customer.name=Ivan Ivanov, age : {0}, URL : {1}
```

messages_ru_RU.properties

```
customer.name=Иван Иванов, возраст : {0}, URL : {1}
```

```
public static void main(String[] args) {  
    ApplicationContext context  
        = new ClassPathXmlApplicationContext("locale.xml");  
  
    String name = context.getMessage("customer.name",  
        new Object[] { 28, "http://www.luxoft.com" }, Locale.US);  
  
    System.out.println("Customer name (English) : " + name);  
  
    String nameRussian = context.getMessage("customer.name",  
        new Object[] {28, "http://www.luxoft.com" }, Locale.RU);  
  
    System.out.println("Customer name (Russian) : " + nameRussian);  
}
```

Spring Framework :: Локализация



```
public class CustomerService implements MessageSourceAware {  
    private MessageSource messageSource;  
  
    public void setMessageSource(MessageSource messageSource) {  
        this.messageSource = messageSource;  
    }  
  
    public void printMessage(){  
        ApplicationContext context  
            = new ClassPathXmlApplicationContext("locale.xml");  
  
        String name = context.getMessage("customer.name",  
            new Object[] { 28, "http://www.luxoft.com" }, Locale.US);  
  
        System.out.println("Customer name (English) : " + name);  
  
        String nameRussian = context.getMessage("customer.name",  
            new Object[] {28, "http://www.luxoft.com" }, Locale.RU);  
  
        System.out.println("Customer name (Russian) : " + nameRussian);  
    }  
}
```

Spring Framework :: Инициализация коллекций



```
public class Customer {  
    private List<Object> lists;  
    private Set<Object> sets;  
    private Map<Object, Object> maps;  
    private Properties pros;  
}
```

```
<bean id="CustomerBean" class="com.mkyong.common.Customer">
```

```
<!-- java.util.List -->
```

```
<property name="lists">
```

```
    <list>
```

```
        <value>1</value>
```

```
        <ref bean="PersonBean" />
```

```
        <bean class="com.mkyong.common.Person">
```

```
            <property name="name" value="Ivan" />
```

```
            <property name="address" value="address" />
```

```
            <property name="age" value="28" />
```

```
        </bean>
```

```
    </list>
```

```
</property>
```


Spring Framework :: Инициализация коллекций



```
public class Customer {  
    private List<Object> lists;  
    private Set<Object> sets;  
    private Map<Object, Object> maps;  
    private Properties pros;  
}
```

```
<bean id="CustomerBean" class="com.mkyong.common.Customer">
```

```
<!-- java.util.Set -->
```

```
<property name="sets">
```

```
    <set>
```

```
        <value>1</value>
```

```
        <ref bean="PersonBean" />
```

```
        <bean class="com.mkyong.common.Person">
```

```
            <property name="name" value="Ivan" />
```

```
            <property name="address" value="address" />
```

```
            <property name="age" value="28" />
```

```
        </bean>
```

```
    </set>
```

```
</property>
```

Spring Framework :: Инициализация коллекций



```
public class Customer {  
    private List<Object> lists;  
    private Set<Object> sets;  
    private Map<Object, Object> maps;  
    private Properties pros;  
}
```

```
<bean id="CustomerBean" class="com.mkyong.common.Customer">  
  
    <!-- java.util.Map -->  
    <property name="maps">  
        <map>  
            <entry key="Key 1" value="1" />  
            <entry key="Key 2" value-ref="PersonBean" />  
            <entry key="Key 3">  
                <bean class="com.mkyong.common.Person">  
                    <property name="name" value="Ivan" />  
                    <property name="address" value="address" />  
                    <property name="age" value="28" />  
                </bean>  
            </entry>  
        </map>  
    </property>
```

Spring Framework :: Инициализация коллекций



```
public class Customer {  
    private List<Object> lists;  
    private Set<Object> sets;  
    private Map<Object, Object> maps;  
    private Properties pros;  
}
```

```
<bean id="CustomerBean" class="com.mkyong.common.Customer">
```

```
<!-- java.util.Properties -->
```

```
<property name="pros">
```

```
<props>
```

```
<prop key="admin">admin@nospam.com</prop>
```

```
<prop key="support">support@nospam.com</prop>
```

```
</props>
```

```
</property>
```

Упражнения

- Работа со схемой Spring IoC
 - 20 мин – самостоятельная работа;
 - 10 мин – обсуждение;

Spring Framework :: Наследование свойств



```
<bean id="inheritedTestBean" abstract="true"  
      class="org.springframework.beans.TestBean">
```

```
  <property name="name" value="parent" />
```

```
  <property name="age" value="1" />
```

```
</bean>
```

```
<bean id="inheritsWithDifferentClass"  
      class="org.springframework.beans.DerivedTestBean"  
      parent="inheritedTestBean" init-method="initialize">
```

```
  <property name="name" value="override" />
```

```
  <!-- the age property value of 1 will be inherited from parent -->
```

```
</bean>
```

Spring Framework :: Объединение коллекций



```
<beans>
  <bean id="parent" abstract="true" class="example.ComplexObject">
    <property name="adminEmails">
      <props>
        <prop key="administrator">administrator@example.com</prop>
        <prop key="support">support@example.com</prop>
      </props>
    </property>
  </bean>
  <bean id="child" parent="parent">
    <property name="adminEmails">
      <!-- the merge is specified on the *child* collection definition -->
      <props merge="true">
        <prop key="sales">sales@example.com</prop>
        <prop key="support">support@example.co.uk</prop>
      </props>
    </property>
  </bean>
</beans>
```

child.adminEmails= administrator=administrator@example.com
sales=sales@example.com
support=support@example.co.uk

Применимо к properties, list, set, map.

Spring Framework :: Пустые и null значения



```
<bean class="ExampleBean">  
  <property name="email" value="" />  
</bean>
```

```
<bean class="ExampleBean">  
  <property name="email"><null /></property>  
</bean>
```

Spring Framework :: p-namespace

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:p="http://www.springframework.org/schema/p">

  <bean name="classic" class="com.example.ExampleBean">
    <property name="email" value="foo@bar.com" />
  </bean>

  <bean name="p-namespace" class="com.example.ExampleBean"
    p:email="foo@bar.com" />

  <bean name="john-classic" class="com.example.Person">
    <property name="name" value="John Doe" />
    <property name="spouse" ref="jane" />
  </bean>

  <bean name="john-modern" class="com.example.Person"
    p:name="John Doe"
    p:spouse-ref="jane" />

  <bean name="jane" class="com.example.Person">
    <property name="name" value="Jane Doe" />
  </bean>
</beans>
```


Spring Framework :: Профили конфигурации



```
<beans profile="dev">
  <jdbc:embedded-database id="dataSource">
    <jdbc:script location="classpath:com/bank/config/sql/schema.sql" />
    <jdbc:script location="classpath:com/bank/config/sql/test-data.sql" />
  </jdbc:embedded-database>
</beans>
```

```
<beans profile="production">
  <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"
</beans>
```

Указание профиля и загрузка конфигурации в Java-коде:

```
GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
ctx.getEnvironment().setActiveProfiles("dev");
ctx.load("classpath:/com/bank/config/xml/*-config.xml");
ctx.refresh();
```

Указание профиля в параметрах командной строки:

```
-Dspring.profiles.active="profile1,profile2"
```

Spring Framework :: Java-based конфигурация



@Configuration

@Profile("dev")

```
public class TransferServiceConfig {  
    @Autowired DataSource dataSource;  
    @Bean  
    public TransferService transferService() {  
        return new DefaultTransferService(accountRepository(), feePolicy());  
    }  
    @Bean  
    public AccountRepository accountRepository() {  
        return new JdbcAccountRepository(dataSource);  
    }  
    @Bean  
    public FeePolicy feePolicy() {  
        return new ZeroFeePolicy();  
    }  
}
```

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();  
ctx.getEnvironment().setActiveProfiles("dev");  
// find and register all @Configuration classes within  
ctx.scan("com.bank.config.code");  
ctx.refresh();
```

Упражнения

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

Вопросы!?

