Урок 1

Введение в Spring

Что такое Spring? Понятие внедрения зависимостей и инверсии управления. Каким образом Spring облегчает разработку? Что такое bean в Spring? Понятие контекста Spring. Подключение Spring. Конфигурирование Spring (XML, аннотации). Использование JavaConfig для конфигурирования. DI в Spring (внедрение зависимостей).

Что такое Spring?

Внедрение зависимостей и инверсия управления

Каким образом Spring облегчает разработку?

Конфигурирование Spring

- Конфигурирование с использованием XML
- Конфигурирование с использованием аннотаций
- JavaConfig
- Советы в выборе способа конфигурации

Внедрение зависимостей в Spring

- Внедрение примитивных типов
- Внедрение объектов

Код учебного проекта

Практическое задание

Дополнительные материалы

Используемая литература

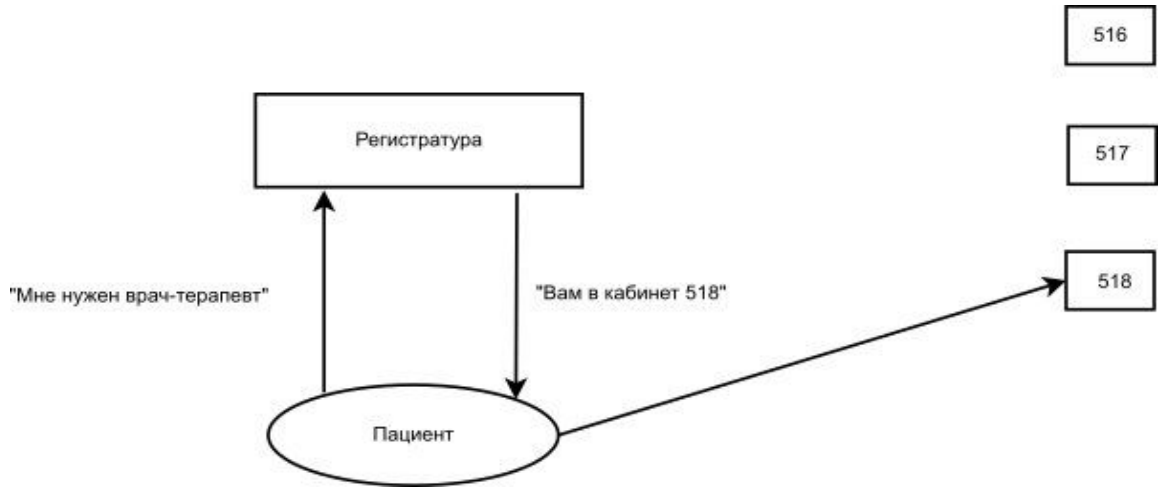
Что такое Spring?

Spring – один из самых популярных фреймворков для разработки веб-приложений на языке Java, главным преимуществом которого является простота и легковесность. Фреймворк Spring представляет собой Inversion of Control (IoC) контейнер.

Внедрение зависимостей и инверсия управления

Inversion of Control (IoC, инверсия управления) – обозначение важного принципа ООП, согласно которому контроль над управлением элементом программы (например, отдельным классом) передается от вас отдельному компоненту. Форм такого контроля может быть множество, но Spring реализует форму «Внедрение зависимостей». Выясним, зачем доверять управление классами какому-то отдельному компоненту и что-то внедрять.

В повседневной жизни много примеров инверсии управления и внедрения зависимостей. Вы приходите в больницу на прием к врачу, и чтобы узнать номер кабинета, обращаетесь в регистратуру. На основе тех данных, которые получит от вас регистратура (например, «мне нужен терапевт»), вам дадут информацию. Конечный выбор номера кабинета остается не за вами, а за регистратурой – произошла инверсия управления. Да и зачем самостоятельно искать нужный кабинет? Вы знаете только то, что вам нужно к определенному специалисту, который выполняет конкретные функции (а значит, реализует определенный интерфейс). Графически этот процесс изображен на рисунке:



Плюсы от такого подхода в реальной жизни очевидны.

Разберемся с инверсией управления в коде. Представим, что должны написать программу, которая умеет делать фото с помощью пленочного фотоаппарата. Исходя из задания, нам нужно разработать два класса: Camera (фотоаппарат) и CameraRoll (фотопленка). Учитывая, что фотопленка «заряжается» в фотоаппарат, наш класс Camera будет содержать поле типа CameraRoll.

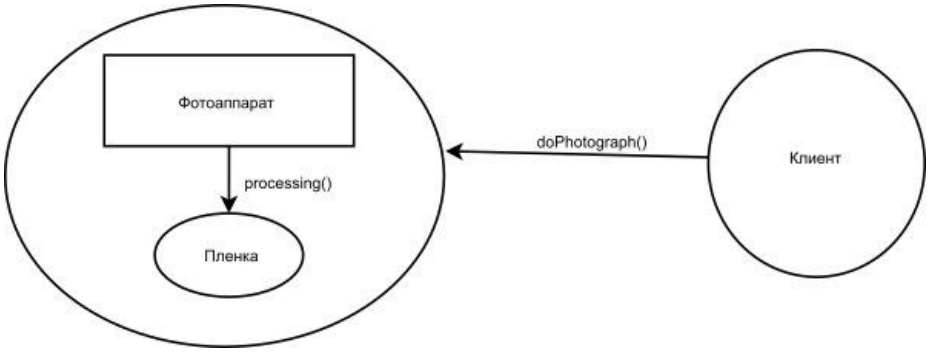
Класс Camera будет иметь следующий вид:

```
public class Camera {
  private CameraRoll cameraRoll;

  public void doPhotograph() {
    System.out.println("Щелк!");
    cameraRoll.processing();
  }
}
```

Ничего необычного в этом классе не происходит. При вызове метода doPhotograph() имитируется процесс создания фотографии (слышен звук щелчка), а с помощью метода processing() происходит отработка кадра на пленке. Класс CameraRoll будет выглядеть так:

```
public class CameraRoll {
  public void processing() {
      System.out.println("-1 кадр");
  }
}
```

Таким образом имитируется следующее отношение:



Все выглядит вполне просто. Причем здесь инверсия управления и внедрение зависимостей?

Попробуем сделать фотографию и напишем следующий клиентский код:

```
public class Client {
  public static void main(String[] args) {
    Camera camera = new Camera();
    camera.doPhotograph();
  }
}
```

В данном коде мы создаем объект фотоаппарата и пытаемся сделать фотографию. Что в итоге? Слышим щелчок, но фото не сделано. Чтобы сделать фотографию, необходима пленка внутри фотоаппарата. Изменим код фотоаппарата, ведь он полностью зависит от фотопленки и без нее работать не будет.

Первое, что сделали бы некоторые:

```
public class Camera {
  private CameraRoll cameraRoll = new CameraRoll();

  public void doPhotograph(){
    System.out.println("Щелк!");
    cameraRoll.processing();
  }
}
```

Фотоаппарат заработал, но где вы видели камеры, которые производятся со встроенной пленкой? И зачем так увязывать фотоаппарат с конкретной фотопленкой? Перепишем иначе:

```
public class Camera {
  private CameraRoll cameraRoll;

  public Camera(CameraRoll cameraRoll) {
    this.cameraRoll = cameraRoll;
  }

  public void doPhotograph() {
    System.out.println("Щелк!");
    cameraRoll.processing();
  }
}
```

Все будет работать. В данном листинге мы наблюдаем внедрение зависимости через конструктор. Да, мы сделали предыдущий код более гибким, и теперь сами выбираем, какой объект класса фотопленки вставить в фотоаппарат. Но получается, что создав фотоаппарат единожды и применив при создании любую понравившуюся нам фотопленку, мы больше не сможем поменять ее, даже когда она закончится. А хотелось бы иметь возможность вставлять и доставать любую фотопленку из фотоаппарата любое количество раз. Используем для этого **get** и **set**. Перепишем наш код:

```
public class Camera {
	private CameraRoll cameraRoll;
	public CameraRoll getCameraRoll() {
		return cameraRoll;
	}
	public void setCameraRoll(CameraRoll cameraRoll) {
		this.cameraRoll = cameraRoll;
	}
	public void doPhotograph(){

		System.out.println("Щелк!");
		cameraRoll.processing();

	}
}
```

Теперь можем сами менять фотопленку в любое время. Но вдруг нам захочется использовать черно-белую пленку? Чтобы это стало возможным, необходимо написать интерфейс фотопленки и две ее реализации: цветную и черно-белую.

Интерфейс и две его реализации будут содержать следующий код:

```
public interface CameraRoll {
	public void processing();
}
public class ColorCameraRoll implements CameraRoll {
	@Override
	public void processing() {

		System.out.println("-1 цветной кадр");
	}
}
public class BlackAndWhiteCameraRoll implements CameraRoll {
	public void processing(){
		System.out.println("-1 черно-белый кадр");
	}
}
```

С первого варианта кода до данного момента мы «отвязали» фотоаппарат не просто от конкретного объекта класса пленки, но и от конкретного вида самой пленки. Все это получилось благодаря использованию **get**- и **set**-метода и интерфейса (внедрение через сеттер).

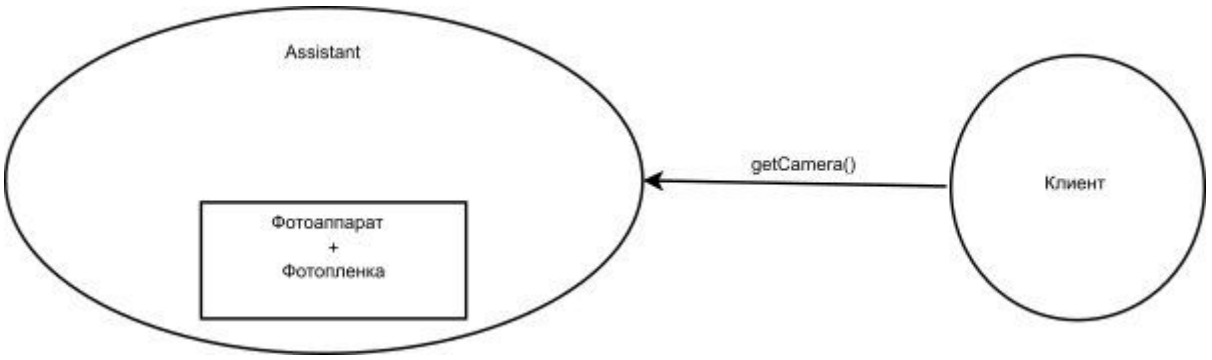
Теперь фотоаппарат умеет работать с несколькими видами пленки, и менять мы ее можем в любое время.

Взглянем теперь, какие действия необходимо произвести клиенту, чтобы сделать фотографию. Клиентский код будет иметь следующий вид:

```
public class Client {
	public static void main(String[] args) {
		Camera camera = new Camera();
		CameraRoll cameraRoll = new ColorCameraRoll();
		camera.setCameraRoll(cameraRoll);
		camera.doPhotograph();
	}
}
```

Фотоаппарат теперь отлично настраивается. В этом коде присутствуют вполне шаблонные действия: создания объекта фотоаппарата, фотопленки, внедрение зависимости. Но зачем клиенту заботиться об этом? Ему просто нужно сделать фотографию, а созданием и внедрением пусть занимается тот, кто в дальнейшем отдаст настроенный фотоаппарат клиенту, чтобы он сделал фото.

В идеале это должно выглядеть следующим образом:



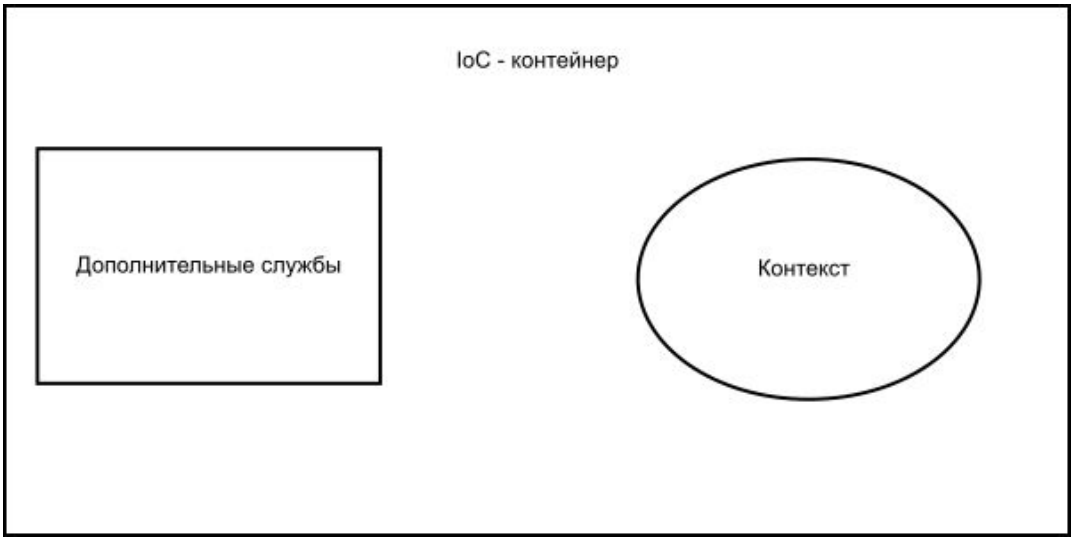
И клиентский код теперь выглядит так:

```
public class Client {

	public static void main(String[] args) {

		Assistant assistant = new Assistant();
		Camera camera = assistant.getCamera();
		camera.doPhotograph();
	}
}
```

Теперь у клиента есть помощник, который делает всю второстепенную работу за него. Он управляет процессом создания и настройки фотоаппарата – произошла инверсия управления. Основной функцией помощника является вставка фотопленки в фотоаппарат – это основная форма инверсии управления, которая называется внедрением зависимости. В данном случае, класс **Assistant** является для нас подобием IoC-контейнера. Здесь и кроется идея Spring!

Каким образом Spring облегчает разработку?

Spring является реализацией IoC-контейнера, а также предоставляет дополнительные службы для содержащихся в нем объектов. Для реализации механизма предоставления этих служб необходима архитектура, отличающаяся от той, что вы видели в предыдущей главе. Эта архитектура напоминает нечто подобное:



Как видите, подобный контейнер состоит из двух компонентов:

- **Контекст** – хранит созданные объекты с уже внедренными зависимостями (в нашем примере – объект фотоаппарата);
- **Дополнительные службы** – благодаря им создаются объекты и внедряются зависимости, но есть и огромное количество других служб, которые определяют функционал Spring. Главная особенность Spring заключается в том, что он реализует модульную архитектуру, где каждый модуль представляет собой JAR-файл и наделен определенным функционалом. Можно подключать только те модули, которые нужны для решения конкретной задачи.

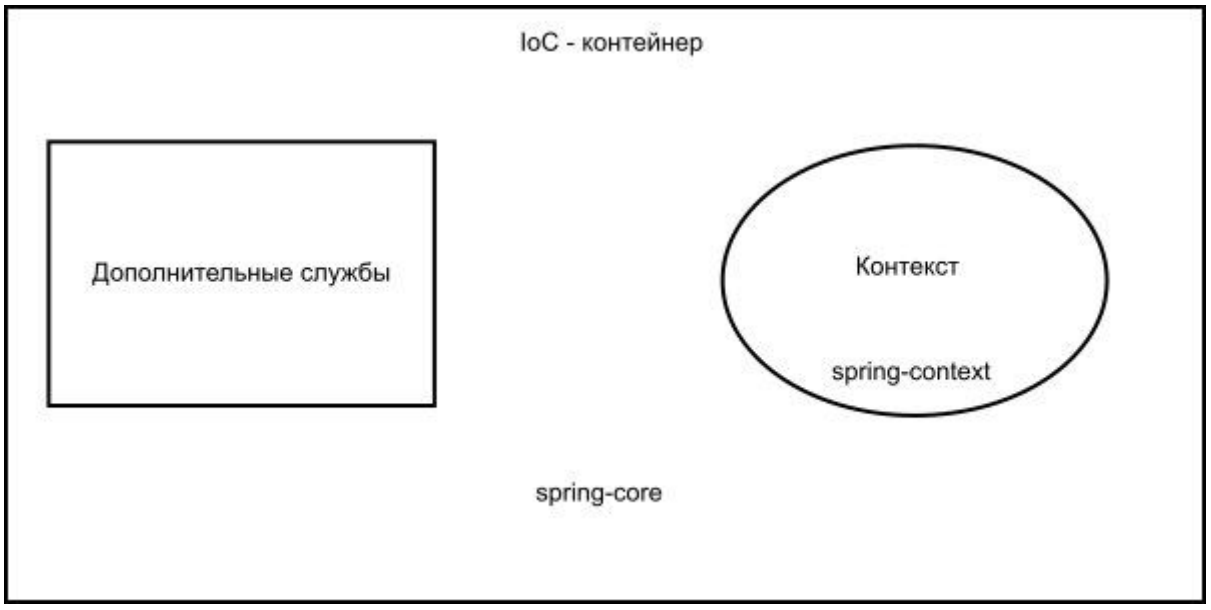
Перепишем код из предыдущей главы, но уже с использованием Spring. Первое, что необходимо сделать – создать пустой **Maven**-проект и подключить зависимости (модули Spring) в файл **pom.xml**. Все зависимости будем подключать с **mvnrepository.com**.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-core -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>4.3.3.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.3.3.RELEASE</version>
</dependency>
```

Spring-core – это основной модуль, необходимый абсолютно для всех приложений Spring. Он предоставляет классы для использования другим модулям.

Spring-context обеспечивает работу контекста IoC-контейнера.

Для окончательного понимания визуализируем эти модули:



Теперь у нас есть все, чтобы реализовать предыдущий пример, но уже с использованием Spring. Единственное, что изменим в предыдущим коде – создадим интерфейс **Camera** с единственным методом **doPhotograph()**, а наш предыдущий класс будет называться **CameraImpl** и будет

реализовывать данный интерфейс. Делаем так, следуя «правилу хорошего тона разработки», позволяющему в дальнейшем расширять приложение. Но основная причина связана с особенностями контекста Spring.

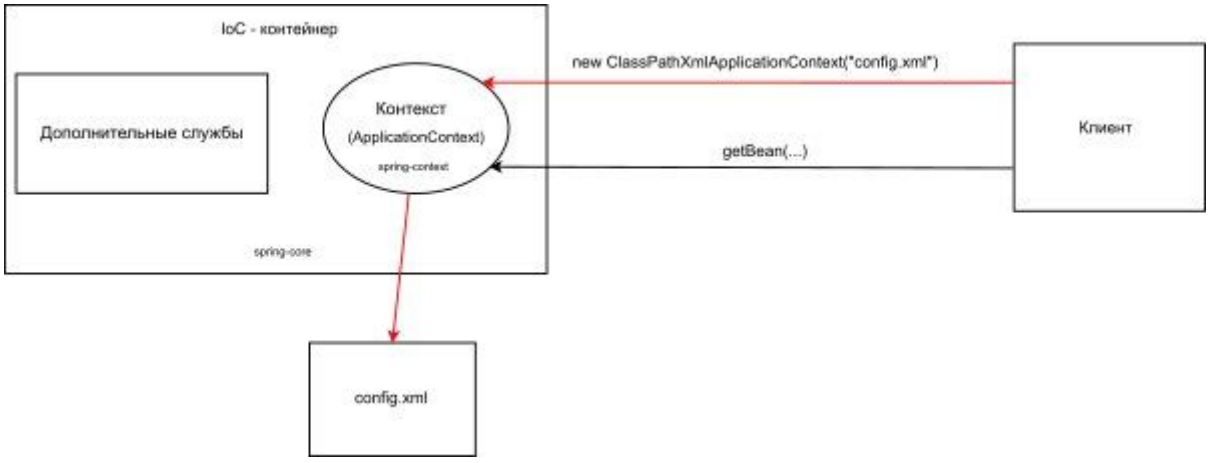
Клиентский код программы:

```
public class Client {
      public static void main(String[] args) {
      ApplicationContext context = new
ClassPathXmlApplicationContext("config.xml");
            Camera camera = context.getBean("camera",Camera.class );
            camera.doPhotograph();
      }
}
```

Теперь в роли класса **Assistant** выступает интерфейс **ApplicationContext**. Его метод **getBean** предоставляет клиенту настроенную камеру.

Bean – это объект любого класса, который управляется контейнером Spring (подробнее о **bean** – в следующем уроке).

Каковы особенности данного кода по сравнению с аналогичным, написанным без использования Spring? Во-первых, для создания контекста, который играет роль нашего ассистента, необходимо передать некие параметры. В предыдущем примере мы ничего не передавали, так как точно знали, что класс **Assistant** нужен только для настройки и получения объекта фотоаппарата. А так как Spring является фреймворком, то и управлять он может абсолютно любыми объектами. Чтобы указывать Spring, какими именно объектами следует управлять (поместить в контекст) и какие зависимости необходимо удовлетворить, контексту передается определенная информация. В данном случае она находится во внешнем конфигурационном файле **config.xml**. Для чтения этой информации из конфигурационного файла (создания контекста) используется конкретная реализация **ApplicationContext** – **ClassPathXmlApplicationContext**. Она прекрасно «понимает» XML-язык. Итоговая схема данного процесса выглядит следующим образом:

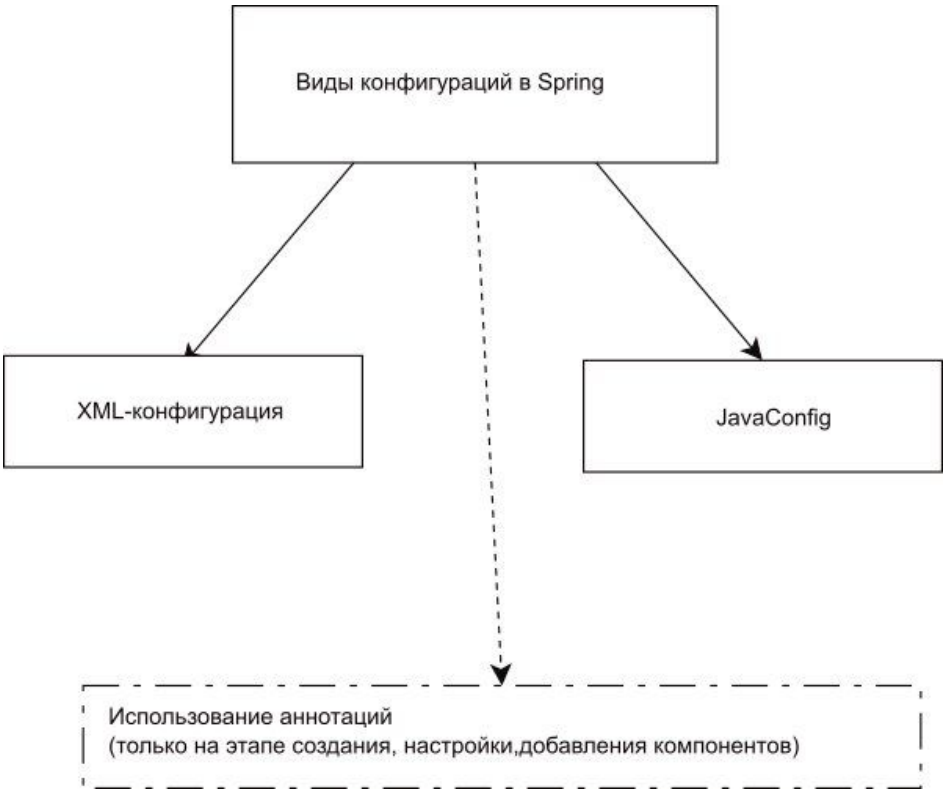


Сравните ее с приведенной ранее схемой, которая отражает аналогичный процесс без использования Spring. Они похожи, но в данном случае в роли класса **Assistant** выступает **ApplicationContext**, которому передается информация о том, какими конкретно объектами стоит управлять. Из данной схемы видно, что пока мы не используем дополнительных служб, а работаем лишь с контекстом. Чтобы понять механизм создания контекста, заглянем в конфигурационный файл **config.xml**.

Конфигурирование Spring

В упрощенном виде конфигурация представлена на схеме:

Виды конфигураций в Spring:



Можно сделать вывод, что Spring обладает двумя самостоятельными видами (способами) конфигураций. А третья (использование аннотаций) может применяться только на этапе 2 совместно с одним из видов «самостоятельной» конфигурации: можем применять либо **XML**-конфигурацию + использование аннотаций, либо **JavaConfig** + использование аннотаций. Или два «самостоятельных» способа конфигурации могут применяться без аннотаций. Рассмотрим каждый из способов конфигурации.

Конфигурирование с использованием XML

Конфигурация через XML – первый вариант, который поддерживал Spring, и по сей день самый популярный. Посмотрим в конфигурационный файл **config.xml**:

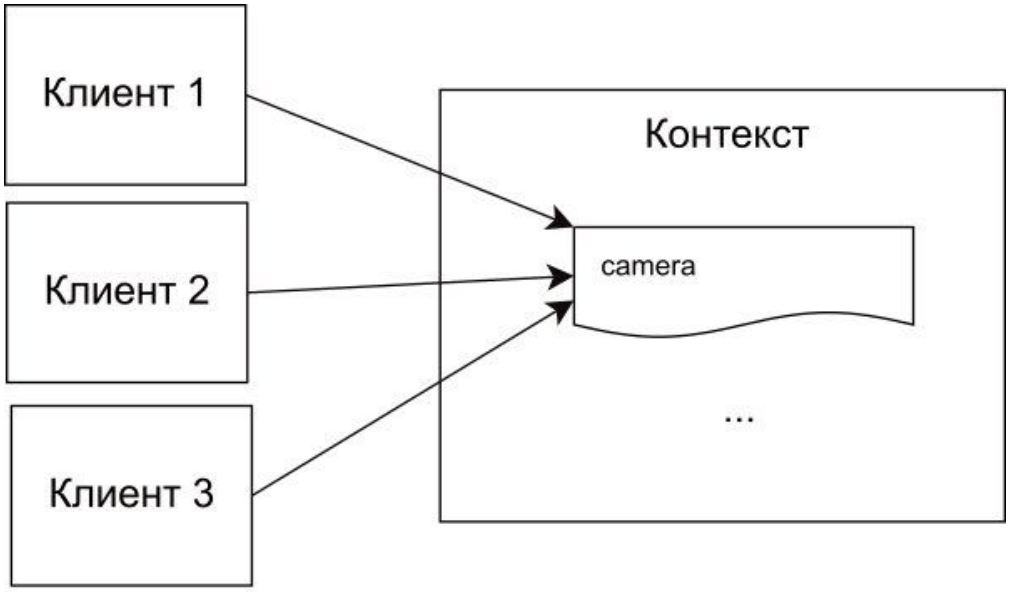
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
      <bean id="cameraRoll" class="net.zt.funcode.lesson1.ColorCameraRoll" />
      <bean id="camera" class="net.zt.funcode.lesson1.CameraImpl">
         <property name="cameraRoll">
            <ref bean="cameraRoll" />
         </property>
      </bean>
</beans>
```

Конфигурационный файл предназначен для указания контейнеру объектов, которыми он должен управлять (бинов). Основной схемой и пространством имен по умолчанию является http://www.springframework.org/schema/beans/spring-beans.xsd и http://www.springframework.org/schema/beans.

Благодаря тегу **<bean …….. />** мы можем:

- Создать объект определенного класса, который сразу же станет бином;
- Внедрить объект другого класса в свойство данного бина, при том этот объект тоже должен быть объявлен в конфигурационном файле.

Spring по умолчанию создает одиночные объекты, и все вызовы **getBean(...)** будут возвращать один и тот же созданный контейнером объект с указанным в скобках идентификатором. Схема доступа к данному объекту изображена на следующей схеме:



: ...

Чем это может быть полезно и как переопределить данное поведение – узнаем позже.

Теперь разберем каждую запись построчно:

```
<bean id="cameraRoll" class="net.zt.funcode.lesson1.ColorCameraRoll" />
```

В данной строке с помощью тега **bean** объявляется объект класса. Для него, как и в случае создания объекта в Java-коде через **new**, необходимо имя, с помощью которого можно обращаться к нему в дальнейшем. В XML-конфигурации таким именем будет значение атрибута ***id***. По данном идентификатору к бину можно обращаться непосредственно в данном файле конфигурации или получать его из контекста, передавая имя в метод **getBean(....)**. Для указания класса создаваемого объекта необходимо задать путь к классу в атрибуте ***class***. Все просто, не так ли? Но есть одна небольшая особенность. Ранее мы оговаривали, что наш класс **ColorCameraRoll** реализует интерфейс **CameraRoll**. Соответственно, в Java-коде возможны два варианта создания данного объекта:

```
ColorCameraRoll cameraRoll = new ColorCameraRoll();
```

В данном варианте мы создаем тип объекта, который является реализацией интерфейса, но не типом самого интерфейса.

Второй вариант:

```
CameraRoll cameraRoll = new ColorCameraRoll();
```

На самом деле, отличия между этими двумя вариантами незначительны, но Spring реализует именно второй. И при извлечении бина из контекста с помощью метода **getBean(.....)** в параметры данного метода нужно передавать интерфейс, а не конкретную реализацию. Сейчас это надо запомнить, а причины разберем позже.

Перейдем ко второму объявлению конфигурационного файла ***config.xml***:

```
<bean id="camera" class="net.zt.funcode.lesson1.CameraImpl">
         <property name="cameraRoll">
            <ref bean="cameraRoll" />
         </property>
</bean>
```

В данном случае происходит создание объекта класс **CameraImp** с идентификатором «**camera**». Чтобы Spring-контейнер самостоятельно вставил фотопленку в фотоаппарат, используем:

```
<property name="cameraRoll">
            <ref bean="cameraRoll" />
</property>
```

Тег **property** дает доступ к свойству нашего объекта класса **camera**. В атрибуте **name** указывается имя данного свойства, а так как пленку мы уже создали ранее, то между двумя тегами **property** необходимо поместить ссылку (reference) на ранее созданный **bean**. Для этого и используется тег **ref** с атрибутом **bean**, в которым указывается **id** ранее созданного бина.

Все это эквивалентно Java-коду в предыдущем варианте, в котором Spring не использовался:

```
    CameraRoll cameraRoll = new ColorCameraRoll();
      Camera camera = new Camera();
      camera.setCameraRoll(cameraRoll);
```

Если некоторый бин необходим для одноразового внедрения и получать его из контекста не требуется, то XML-конфигурацию можно изменить следующим образом:

```
<?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.xsd">
      <bean id="camera" class="net.zt.funcode.lesson1.CameraImpl">
         <property name="cameraRoll">
            <bean  class="net.zt.funcode.lesson1.ColorCameraRoll" />
         </property>
      </bean>
</beans>
```

Данный вариант эквивалентен следующему Java-коду:

```
      Camera camera = new Camera();
      camera.setCameraRoll(new ColorCameraRoll());
```

Конфигурирование с использованием аннотаций

С выходом более новых версий Spring стало возможным убрать часть кода из XML-конфигурации с помощью специальных аннотаций.

Например, чтобы в Spring-контейнере оказался объект фотопленки, достаточно над классом фотопленки указать аннотацию:

```
@Component("cameraRoll")
public class ColorCameraRoll implements CameraRoll {
      public void processing() {
            // TODO Auto-generated method stub
            System.out.println("-1 цветной кадр");
      }
```

Первая аннотация заставляет Spring-контейнер создать объект класса, к которому применена аннотация, с ***id***, указанным в скобках. Данный объект сразу же будет являться бином – а значит компонентом, управляемым контейнером.

Это позволит Spring-контейнеру создать объект данного класса, сделать его бином и поместить в контекст. Id бина указывается в аннотации в скобках, здесь это **cameraRoll**. По этому id объект будет доступен в контексте, и к нему можно обращаться в XML-конфигурации. Но как Spring увидит данную аннотацию? Стоит отметить, что клиентский код при таком подходе не изменится: нам все так же придется создавать контекст с помощью внешнего конфигурационного XML-файла. Но благодаря этой аннотации можно избавиться от тегов **<bean…>** в файле **config.xml**. В нем же необходимо указать Spring-контейнеру, где искать классы, помеченные данной аннотацией.

Сделать это можно следующим образом:

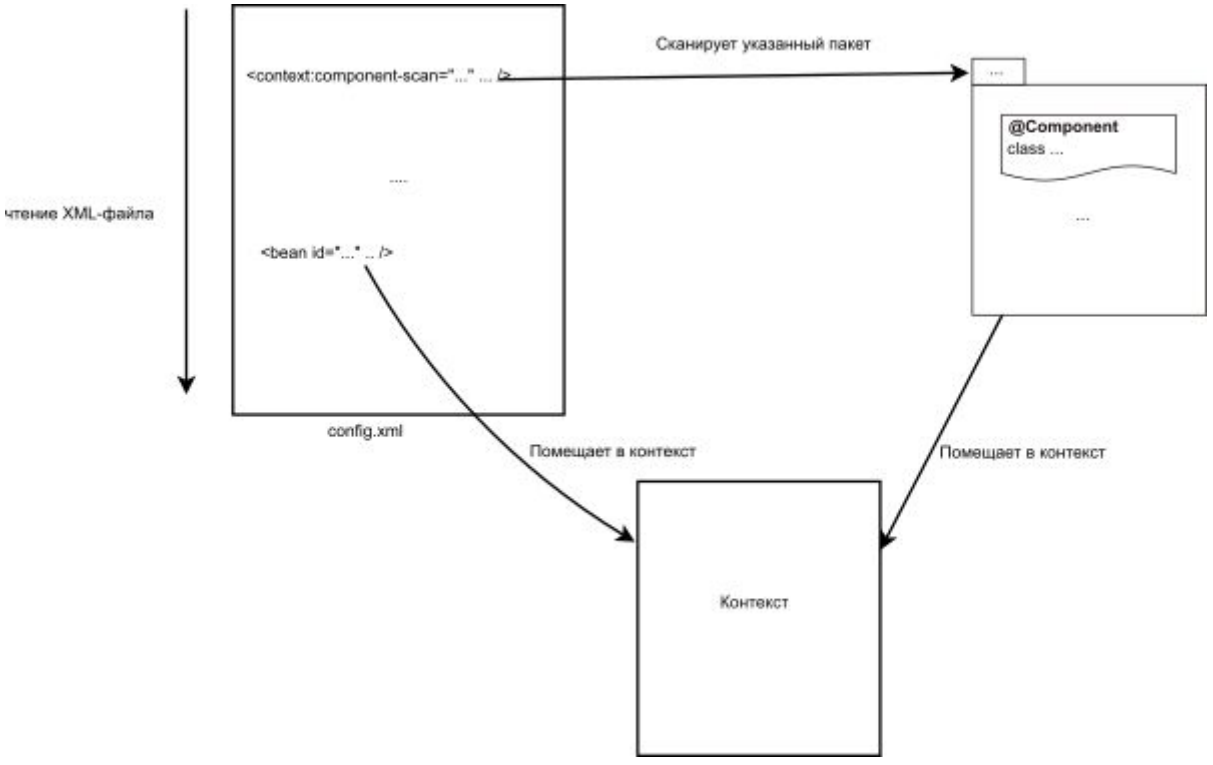
1. В **config.xml** подключаем новое пространство имен и схему;
2. Чтобы прочитав XML-конфигурацию, Spring контейнер «знал», где искать объявленные с помощью аннотаций компоненты, в конфиг-файл добавляем строку:

```
<context:component-scan base-package="net.zt.funcode.lesson1"    />
```

Теперь объект класса **CameraRoll** создается и помещается в контекст с помощью аннотаций. Конфигурационный файл приобрел следующий вид:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd">
<context:component-scan base-package="net.zt.funcode.lesson1"    />
    <bean id="camera" class="net.zt.funcode.lesson1.CameraImpl">
       <property name="cameraRoll">
          <ref bean="cameraRoll" />
       </property>
    </bean>
</beans>
```

Процесс чтения этого конфигурационного файла:



Это был пример применения XML-конфигурации совместно с использованием аннотаций.

JavaConfig

Создание контекста Spring может происходить вообще без использования XML-конфигурации. Всю конфигурацию можно вынести в отдельный Java-класс, помеченный аннотацией **@Configuration**:

```
@Configuration
public class AppConfig {
          @Bean(name="cameraRoll")
          public CameraRoll cameraRoll() {
              return new ColorCameraRoll() ;
          }
@Bean(name="camera")
          public Camera camera(CameraRoll cameraRoll){
            Camera camera = new CameraImpl();
            camera.setCameraRoll(cameraRoll);
            return camera;
          }
}
```

В данном листинге присутствуют две новые аннотации:

- **@Configuration** – аннотация, указывающая на то, что данный Java-класс является классом конфигурации;
- **@Bean** – используется для аннотирования методов, создающих бины в классе, помеченном аннотацией @Configuration. Аналог тега <bean…../> в XML-конфигурации.

Следующий листинг этого Java-кода:

```
        @Bean(name="cameraRoll")
          public CameraRoll cameraRoll() {
              return new ColorCameraRoll() ;
          }

          @Bean(name="camera")
          public Camera camera(CameraRoll cameraRoll){

            Camera camera = new CameraImpl();
            camera.setCameraRoll(cameraRoll);
            return camera;
          }
```

...можно представить как:

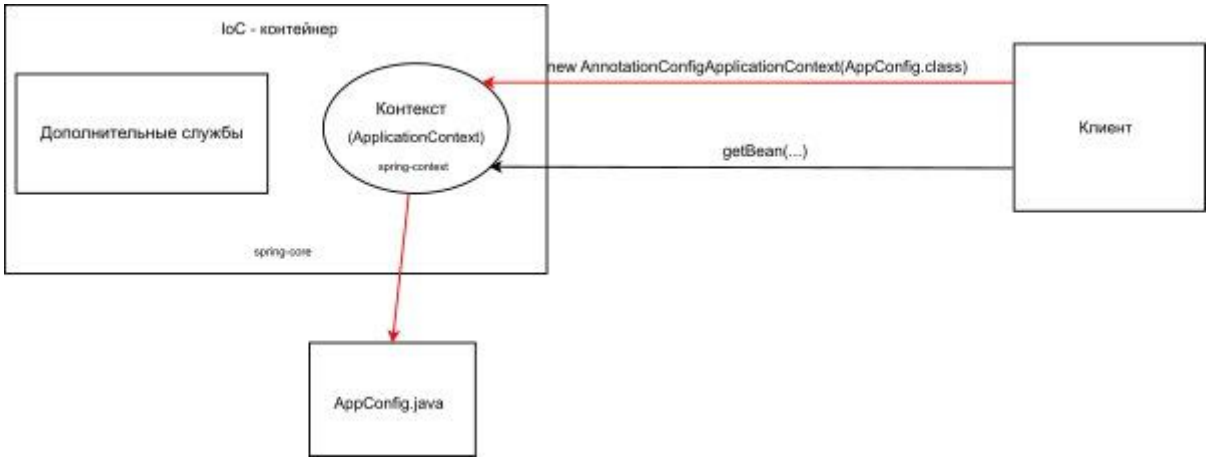
```
<bean id="cameraRoll" class="net.zt.funcode.lesson1.ColorCameraRoll" />

          <bean id="camera" class="net.zt.funcode.lesson1.CameraImpl">
               <property name="cameraRoll">
                 <ref bean="cameraRoll" />
               </property>
</bean>
```

Клиентский код:

```
ApplicationContext                                                    context=new
AnnotationConfigApplicationContext(AppConfig.class);
          Camera camera = context.getBean("camera",Camera.class);
          camera.doPhotograph();
```

Единственным изменением стало то, что теперь для создания контекста применяется другая реализация **ApplicationContext**, которая имеет имя **AnnotationConfigApplicationContext**. В параметр конструктора передается класс, который был описан ранее.

Теперь процесс создания контекста примет такой вид:



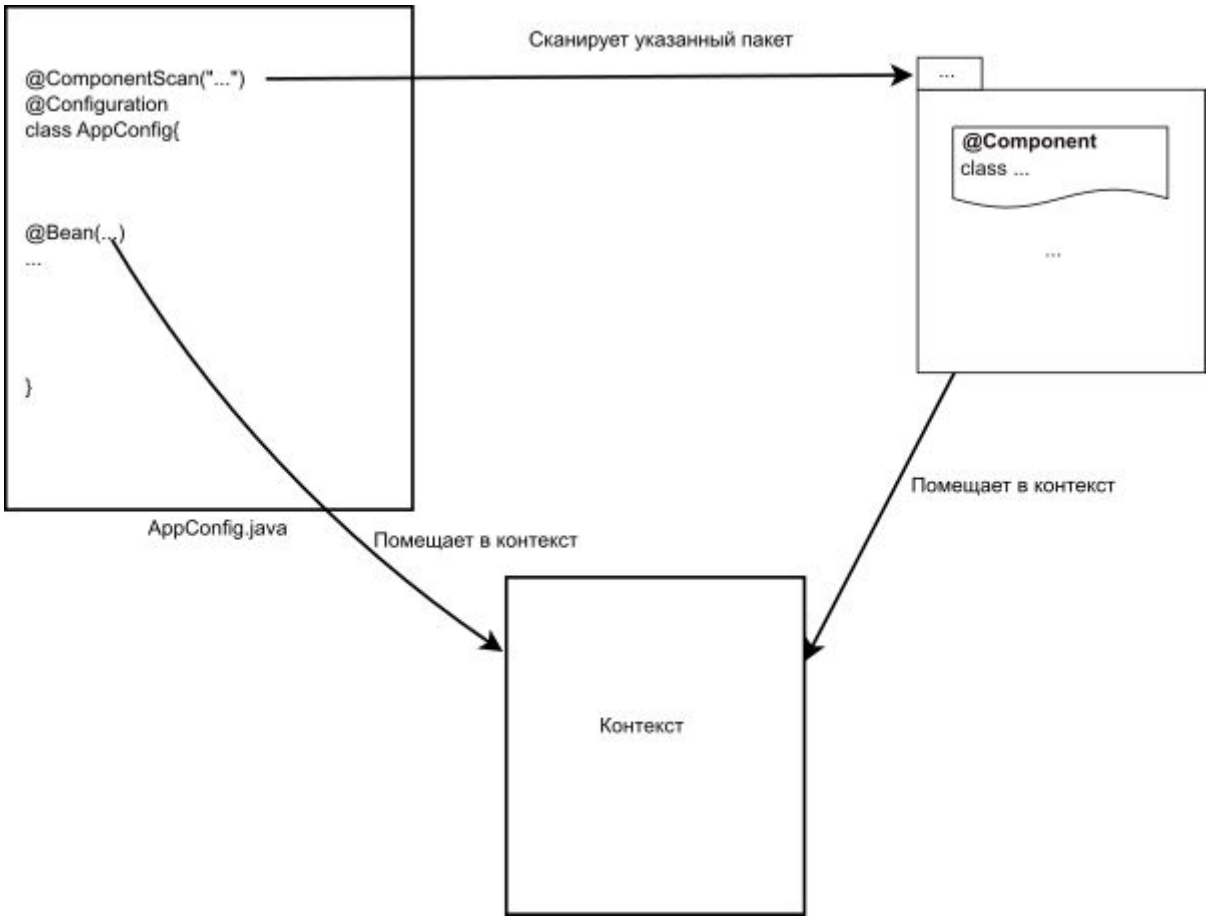
Совместно применять JavaConfig и аннотации можно, добавляя к классу конфигурации аннотации ***@ComponentScan(....)***. В нашем случае это будет выглядеть следующим образом:

```
@Configuration
@ComponentScan("net.zt.funcode.lesson1")
public class AppConfig {
          @Bean(name="camera")
          public Camera camera(CameraRoll cameraRoll){
            Camera camera = new CameraImpl();
            camera.setCameraRoll(cameraRoll);
            return camera;
          }
}
```

Это будет полностью эквивалентно следующему XML:

```
    <context:component-scan base-package="net.zt.funcode.lesson1"    />
    <bean id="camera" class="net.zt.funcode.lesson1.CameraImpl">
       <property name="cameraRoll">
          <ref bean="cameraRoll" />
       </property>
    </bean>
```

Процесс конфигурирования при совместном применении JavaConfig и аннотаций показан на следующей схеме:



Советы в выборе способа конфигурации

XML-конфигурация и JavaConfig являются двумя равнозначными способами конфигурации в Spring. Но JavaConfig все же обладает весомыми преимуществами перед XML-конфигурацией:

- Используется Java-код, а значит не нужно заботиться о xsd-схемах и XML-тегах;
- Можно выявить ошибки на этапе написания конфигурационного класса;
- Больше гибкости за счет работы с объектами и методами.

Вывод: наилучшим способом конфигурации является JavaConfig, но по возможности на протяжении курса будет приводиться и XML-аналог конфигурации.

Стоит ли применять конфигурацию с помощью аннотаций совместно с JavaConfig? Однозначно стоит. Оптимальный способ – условие, согласно которому все бины, необходимые для инфраструктуры приложения (источники данных, менеджеры транзакций), объявляются непосредственно в классе **JavaConfig** путем создания метода и применения к нему аннотации ***@Bean***. А ко всем классам, реализующим написанную нами бизнес-логику (либо к классам, предназначенным для хранения какой-либо информации), применяются специальные аннотации (***@Component***, ***@Service***, ***@Repository*** и т.п.). Рассмотрим их в следующем уроке.

Внедрение зависимостей в Spring

Вернемся к теории и вспомним виды Dependency Injection:

- внедрение через конструктор;
- внедрение через метод установки или внедрение на уровне поля.

Почти во всех случаях предпочтительным является внедрение через метод установки. Именно так мы и делали, но выполняли это непосредственно в классе конфигурации (в случае JavaConfig), либо в XML-файле (в случае XML-конфигурации). Попрактикуемся в возможностях, которые поддерживает Spring для внедрения зависимостей.

Что будем внедрять:

- значения примитивных типов (строки, целочисленные значения и т.п.);
- объекты.

Как будем внедрять?

- с помощью специальных тегов в XML-конфигурации;
- с помощью методов в JavaConfig;
- непосредственно в коде, используя аннотацию **@Autowired**.

Внедрение примитивных типов

Представим, что у нас есть интерфейс **HelloMan**, который объявляет единственный метод приветствия:

```
public interface HelloMan {
      public void helloSay();
      public String getName();
      public void setName(String name);
}
```

И его реализация **HelloManOnceSay**:

```
public class HelloManOnceSay implements HelloMan {
      private String name;
      public HelloManOnceSay(){
      }
      public HelloManOnceSay(String name){
            this.name=name;
      }
      public String getName() {
            return name;
      }
      public void setName(String name) {
            this.name = name;
      }
      public void helloSay(){
            System.out.println("Hello,"+this.name);
      }
}
```

Для работы метода **helloSay()** нам необходимо обеспечить зависимость класса в поле **name**, которая имеет тип **String**.

Внедрить зависимости при использовании XML-конфигурации мы можем несколькими путями: через конструктор и через метод установки. Если применять только XML-конфигурацию, объявление бина и обеспечение внедрения зависимости типа String ***через конструктор*** будет выглядеть следующим образом:

```
   <bean id="helloMan" class="net.zt.funcode.lesson1.HelloManOnceSay">
       <constructor-arg  value="Yuri" />
    </bean>
```

Для тега **constructor-arg** возможно определение двух дополнительных атрибутов: ***type*** и ***index***. Они указывают на тип внедряемого значения и его порядковый номер в конструкторе класса.

```
   <bean id="helloMan" class="net.zt.funcode.lesson1.HelloManOnceSay">
       <constructor-arg  value="Yuri" />
    </bean>
```

Внедрение ***через метод установки*** (предпочтительный способ):

```
  <bean id="helloMan" class="net.zt.funcode.lesson1.HelloManOnceSay">
       <property name="name" value="Yuri" />
   </bean>
```

При использовании **JavaConfig** внедрение ***через конструктор*** выглядит довольно просто:

```
@Bean(name="helloMan")
          public HelloMan helloMan(@Value("Yuri") String name){
            HelloMan helloMan = new HelloManOnceSay(name);
            return helloMan;
          }
```

Аннотация **@Value** внедряет примитивное значение в элемент, к которому применена данная аннотация. Для внедрения зависимости **name** через метод установки достаточно воспользоваться пустым конструктором и использовать метод **setName(name)**.

Но ранее мы упоминали, что бины классов, реализующих бизнес-логику, объявляются вне конфигурации с помощью аннотации **@Component** и подобных. Но как быть с зависимостями данного бина? Применяем ту же аннотацию **@Value**:

```
@Component
public class HelloManOnceSay implements HelloMan {
	@Value("Yuri")
	private String name;
	public HelloManOnceSay(){
	}
	public HelloManOnceSay(String name){
		this.name=name;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public void helloSay(){

		System.out.println("Hello,"+this.name);
	}
}
```

Кроме того, аннотацию **@Value** можно применить и к параметру конструктора, и к методу установки **setName(...)**. Внедрение числовых типов происходит аналогичным образом.

Внедрение объектов

Для внедрения объектов применяется такой же подход. Но в случае применения XML-конфигурации и ***внедрения через конструктор*** вместо такого тега:

```
    <constructor-arg  value="..." />
```

… используется следующая конструкция:

```
     <constructor-arg  ref="..." />
```

Здесь значением **ref** будет имя (идентификатор) нашего бина.

Для внедрения ***через метод установки*** применяется следующая конструкция:

```
    <property name="cameraRoll">
		<ref bean="cameraRoll" />
      </property>
```

Для внедрения в JavaConfig применяется конструкция, которую мы уже рассматривали:

```
@Bean(name="cameraRoll")
    public CameraRoll cameraRoll() {
        return new ColorCameraRoll() ;
          }
@Bean(name="camera")
    public Camera camera(CameraRoll cameraRoll){
             Camera camera = new CameraImpl();
             camera.setCameraRoll(cameraRoll);
             return camera;
           }
```

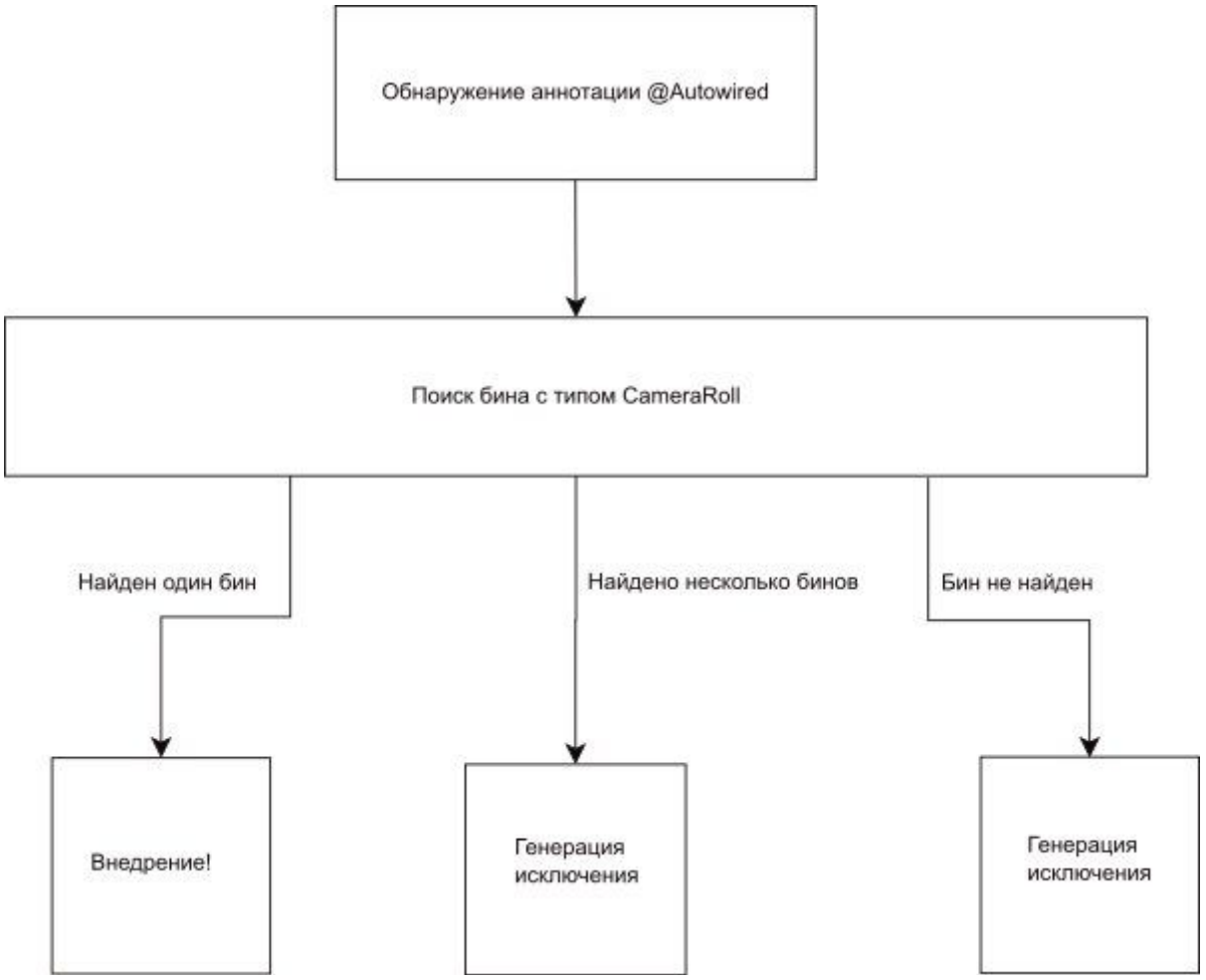
В этом листинге происходит внедрение через метод установки. Чтобы сделать подобное через конструктор, необходимо передать параметр **cameraRoll** в параметры оператора **new** (если класс имеет конструктор с данной сигнатурой).

Самое интересное нас ждет при внедрении объектов через аннотации (в данном примере это оптимальный способ). Необходимо использовать аннотацию **@Autowired**. Перепишем класс камеры, применяя ее:

```
@Component("camera")
public class CameraImpl implements Camera {
      @Autowired
      private CameraRoll cameraRoll;
      public CameraRoll getCameraRoll() {
            return cameraRoll;
      }
      public void setCameraRoll(CameraRoll cameraRoll) {
            this.cameraRoll = cameraRoll;
      }
      public void doPhotograph(){
            System.out.println("Сделана фотография!");
            cameraRoll.processing();
      }
}
```

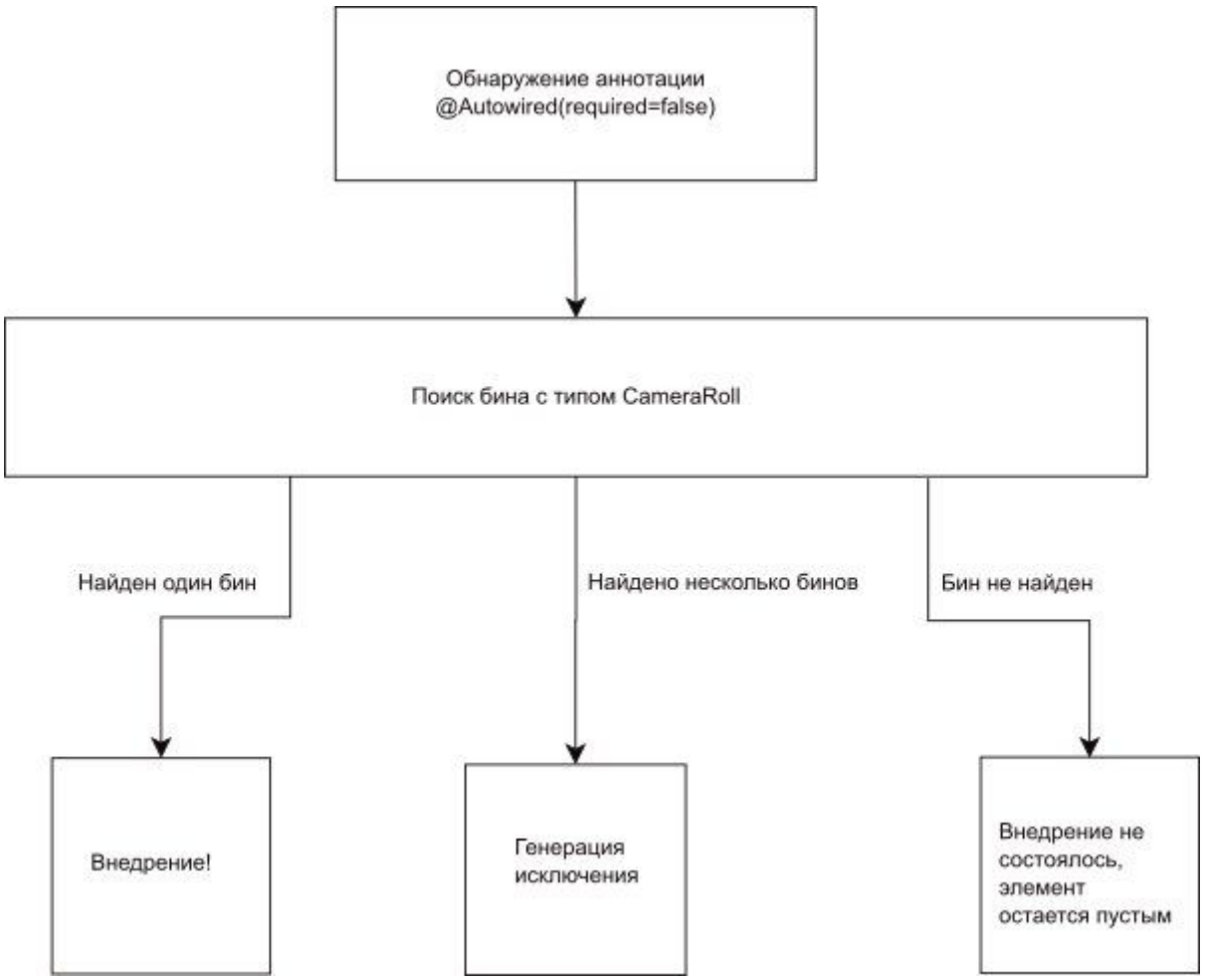
Эта аннотация может применяться к полю, методу установки и конструктору.

В данной программе алгоритм внедрения будет работать следующим образом:

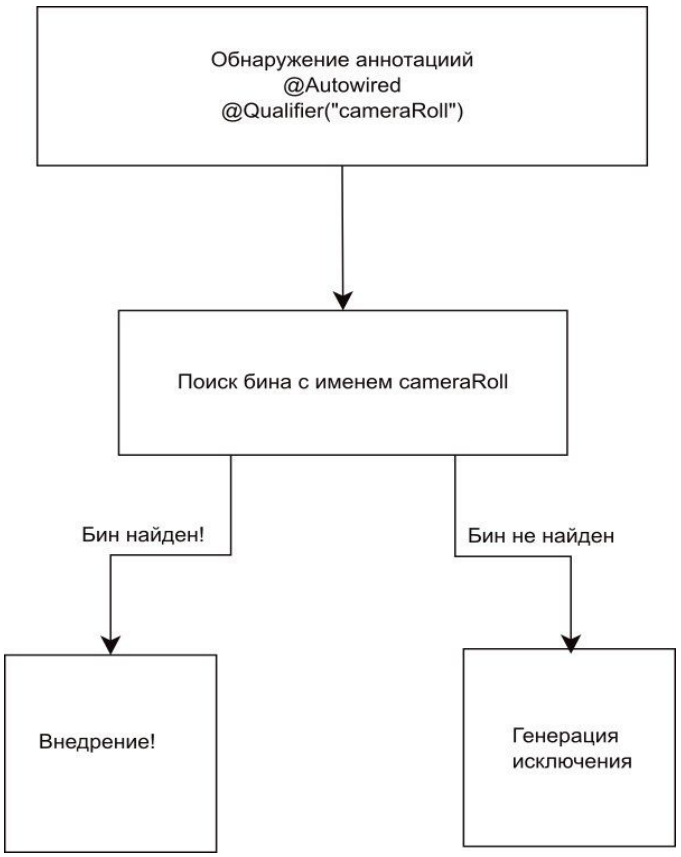


Из схемы видно, что поиск бина-«кандидата на внедрение» осуществляется по типу. Если в контексте имеется несколько бинов заданного типа, или бин не будет найден, то генерируется исключение **NoSuchBeanDefinitionException**.

В обеих ситуациях исключения можно избежать. Например, если атрибут **required** аннотации **@Autowired** установить в значение **false**: **@Autowired(required=false)**. Тогда в случае, если нужный бин отсутствует, в элемент, аннотированный данной аннотацией, значение внедрено не будет.



Чтобы избежать исключения, когда в контексте имеется несколько бинов данного типа, необходимо использовать аннотацию **@Qualifier(“beanName”)**. Она позволяет указать имя бина, который должен быть внедрен.



Код учебного проекта

Конечный вариант кода программы, разработанной в ходе данного урока: https://github.com/Firstmol/Geekbrains-lesson1

Учебный проект содержит два вида конфигурации. Для выбора конкретного варианта необходимо раскомментировать соответствующую строку в классе **Client.java**.

Практическое задание

1. Скачать с официального сайта и установить **Eclipse IDE** (ссылка на скачивание).
2. Запустить Eclipse IDE, нажать на вкладку ***Help -> Eclipse Marketplace***. В появившемся окне в поле ***Find*** ввести «**Spring Tool Suite**». Нажать ***Install***.
3. Создать Maven-проект. Перейти в ***File -> New -> Maven Project***, отметить ***«Create a simple project…»***. Нажать ***Next*** и заполнить все данные для создания учебного проекта.
4. В **pom.xml** подключить зависимости, указанные в материале данной методички.
5. В своем проекте написать программу, аналогичную той, которая разрабатывалась на уроке. Попробовать запустить ее. Выбор способа конфигурации остается за вами.
6. * Разработать интерфейсы и классы для моделирования процесса получения клиентом ружья и выстрела из него. Внедрение патронов в ружье производить с помощью аннотаций. При разработке опираться на учебный проект из данного урока.

Дополнительные материалы

1. Singleton и Prototype: http://docs.spring.io/spring/docs/3.0.0.M3/reference/html/ch04s04.html;
2. Beans и Factory method: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html;
3. Крис Шефер, Кларенс Хо, Роб Харроп. Spring 4 для профессионалов.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Крис Шефер, Кларенс Хо, Роб Харроп. Spring 4 для профессионалов (4-е издание);
2. Крейг Уоллс. Spring в действии.