Урок 2

Контекст и бины в Spring

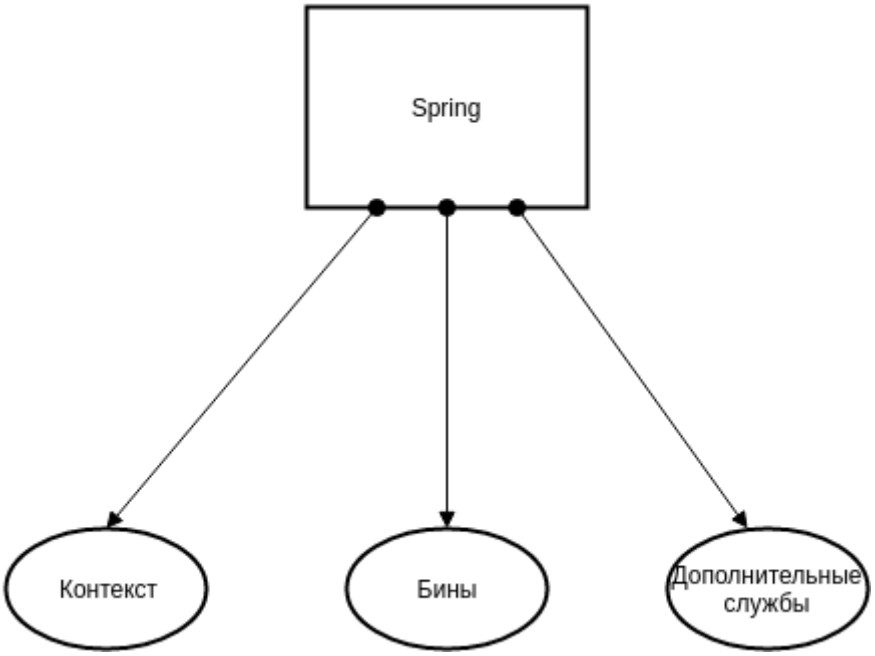
Контекст в Spring. Этапы инициализации контекста. BeanDefinition. BeanFactoryPostProcessor. BeanFactory. BeanPostProcessor. Создание собственных BeanPostProcessor. Жизненный цикл бина.

Введение

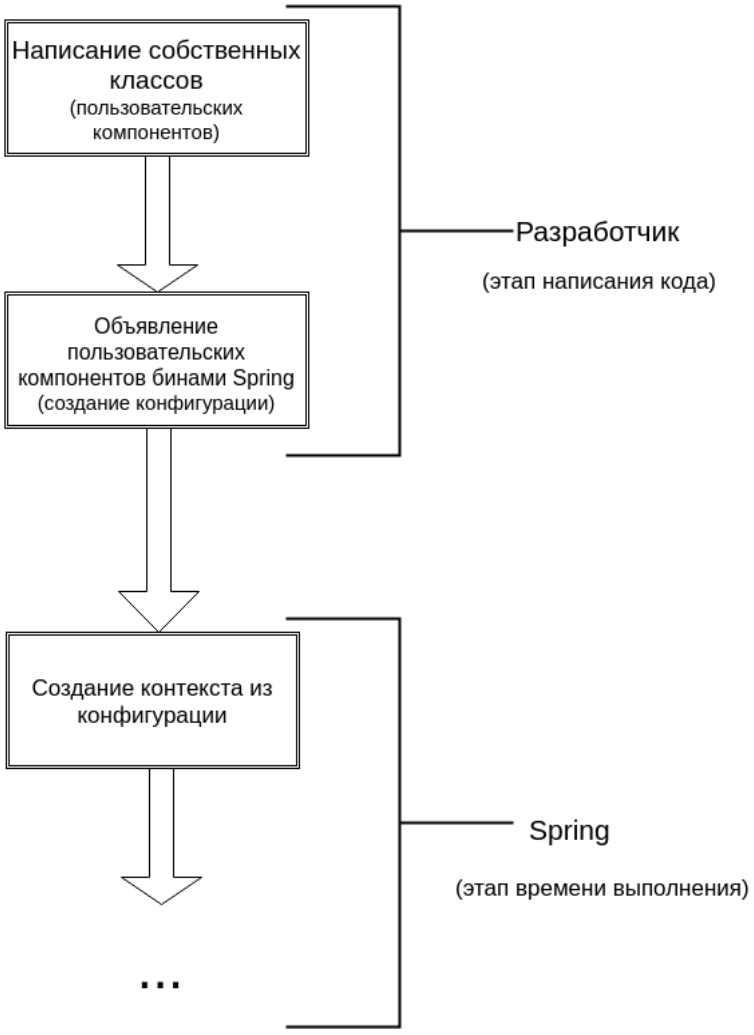
На прошлом уроке мы рассматривали понятие IoC и затронули общие механизмы реализации данной концепции в Spring. На данном этапе изучения фреймворка самыми важными составляющими в Spring для нас являются:

- контекст;
- бины;
- дополнительные службы.

С точки зрения функционала это выглядит так:



Упрощенный порядок работы со Spring можно представить так:



С первыми двумя пунктами этой схемы мы уже знакомы. Обратим внимание на внутренние процессы Spring, научимся вмешиваться в них и посмотрим, чем это будет полезно.

Область видимости бинов

Вспомним пример с фотоаппаратом. В написанной для него программе есть нюанс. Что будет, если клиент случайно сломает фотоаппарат? Времени на починку нет – и он попросит другую камеру.

Модифицируем наш код. Интерфейс фотоаппарата примет вид:

```
public interface Camera {

      CameraRoll getCameraRoll();
      void setCameraRoll(CameraRoll cameraRoll);
      void doPhotograph();
      void breaking();
      boolean isBroken();

}
```

Здесь вызов метода **breaking()** будет имитировать поломку фотоаппарата. А с помощью метода **isBroken()** сможем проверить, является ли фотоаппарат работоспособным. Реализация интерфейса будет выглядеть следующим образом:

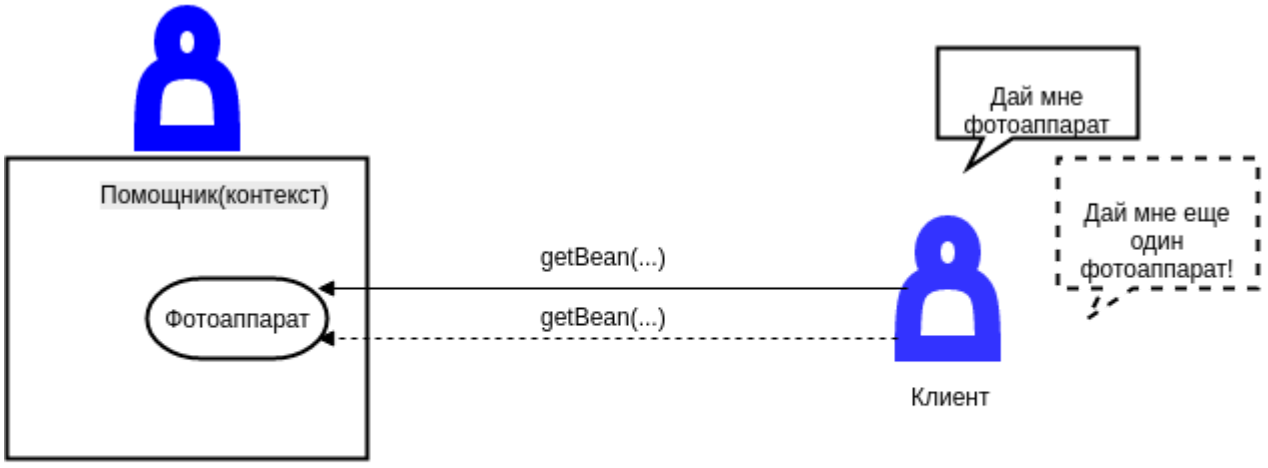
```
@Component("camera")
public class CameraImpl implements Camera {

      @Autowired
      @Qualifier("cameraRoll")
      private CameraRoll cameraRoll;

       @Value("false")
      private boolean broken;

      public CameraRoll getCameraRoll() {
            return cameraRoll;
      }

      public void setCameraRoll(CameraRoll cameraRoll) {
            this.cameraRoll = cameraRoll;
      }

      public boolean isBroken() {
            return broken;
      }

   public void breaking(){

         this.broken=true;
   }

      public void doPhotograph(){

            if(isBroken()){

                  System.out.println("Фотоаппарат сломан!");
                  return;
            }
                System.out.println("Сделана фотография!");
                cameraRoll.processing();

      }
}
```

Мы объявили дополнительное поле **broken**, которое будет иметь логическое значение, позволяющее проверить работоспособность фотоаппарата. Также объявлены два метода: **isBroken()** (проверить, является ли фотоаппарат сломанным) и **breaking()**(сломать фотоаппарат). Кроме того, модифицирован метод **doPhotograph()**, в котором проверяется, работоспособна ли камера.

Смоделируем ситуацию: клиент получает фотоаппарат у своего помощника и случайно его ломает. Сделать фото не получается – он просит еще один фотоаппарат и пытается сделать фото. На языке кода это будет выглядеть так:

```
public class Client {

      public static void main(String[] args) {


      ApplicationContext                                                context=new
AnnotationConfigApplicationContext(AppConfig.class);

       // Получает фотоаппарат
       Camera camera = context.getBean("camera", Camera.class);

      // Ломает фотоаппарат
      camera.breaking();
       // Пытается сделать фото. Неудача!
       camera.doPhotograph();

       // Просит еще один фотоаппарат
       camera = context.getBean("camera", Camera.class);
       // Пытается сделать фото
       camera.doPhotograph();
      }
}
```

Происходит странное: второй фотоаппарат тоже сломан. Причина в том, что это был один и тот же фотоаппарат. И каждый раз, когда клиент будет просить камеру, он будет получать прежнюю.

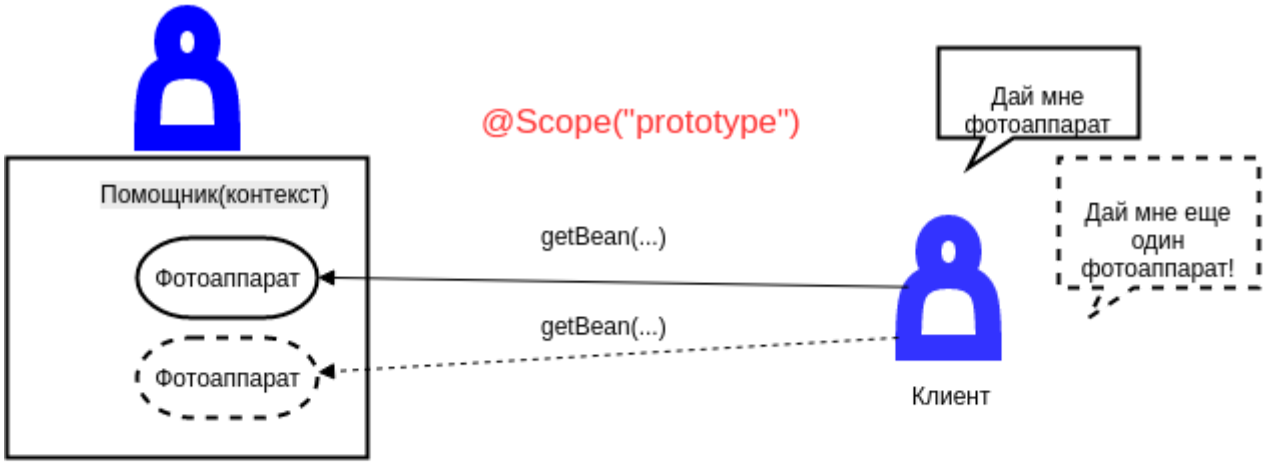


Помощник в данной ситуации не так надежен, как казалось. Дело в том, что фотоаппарат является объектом «одиночка» (Singleton), и все обращения происходят к одному и тому же объекту.

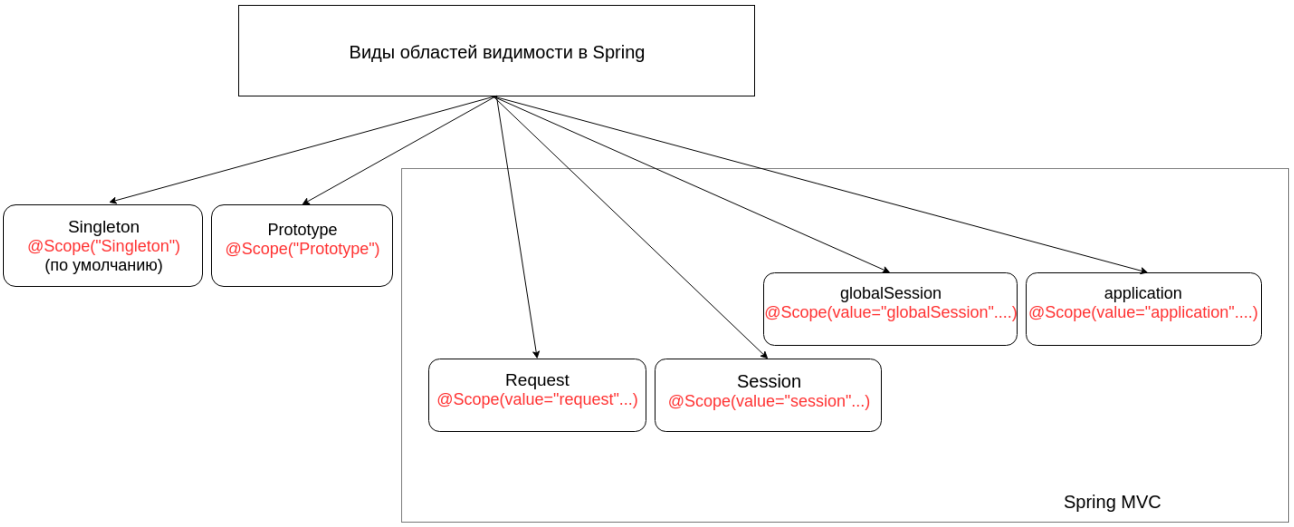
По умолчанию все компоненты Spring являются синглтонами, и в большинстве ситуаций такой подход оптимален, но не в нашем случае. Чтобы переопределить данное поведение, необходимо добавить в класс фотоаппарата следующий код:

```
@Component("camera")
@Scope("prototype")
public class CameraImpl implements Camera {
                  ...

}
```

Мы добавили аннотацию **@Scope(“prototype”)**. Она говорит Spring, что при каждом **getBean** (**camera, Camera.class**) необходимо возвращать новый объект фотоаппарата.



Кроме того, в Spring присутствуют и другие области видимости:

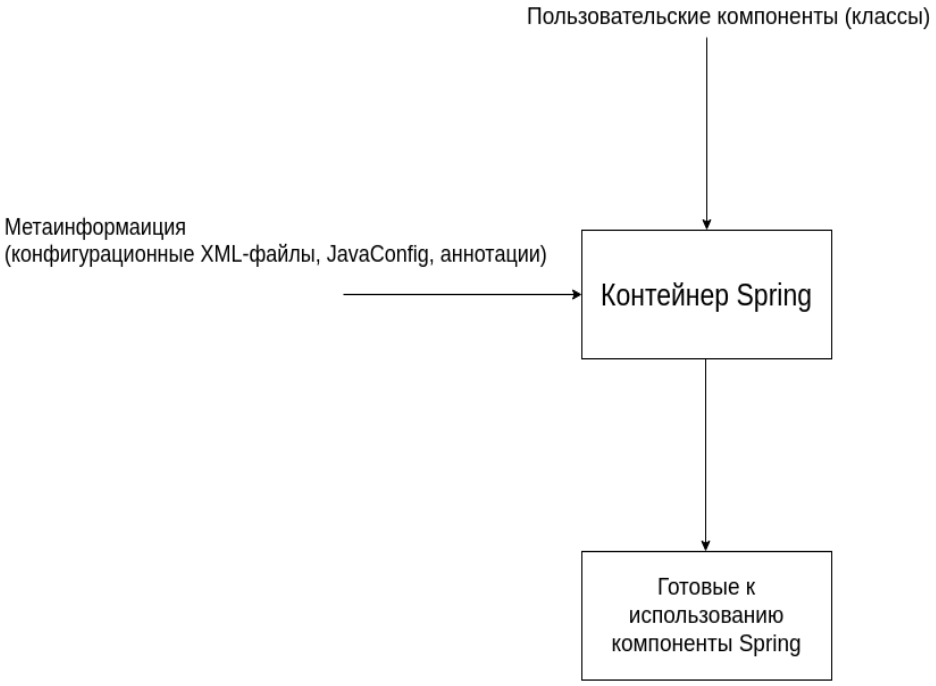


Области видимости **Request**, **Session**, **globalSession**, application относятся к **Spring MVC**. Рассмотрим их в последующих уроках.

Все бины, создаваемые в **JavaConfig** с помощью аннотации **@Bean**, тоже являются синглтонами (аналогично и в XML). Это поведение можно изменить в обоих видах конфигурационных файлов. Подробнее об этом читайте в дополнительном материале к уроку.

Этапы инициализации контекста

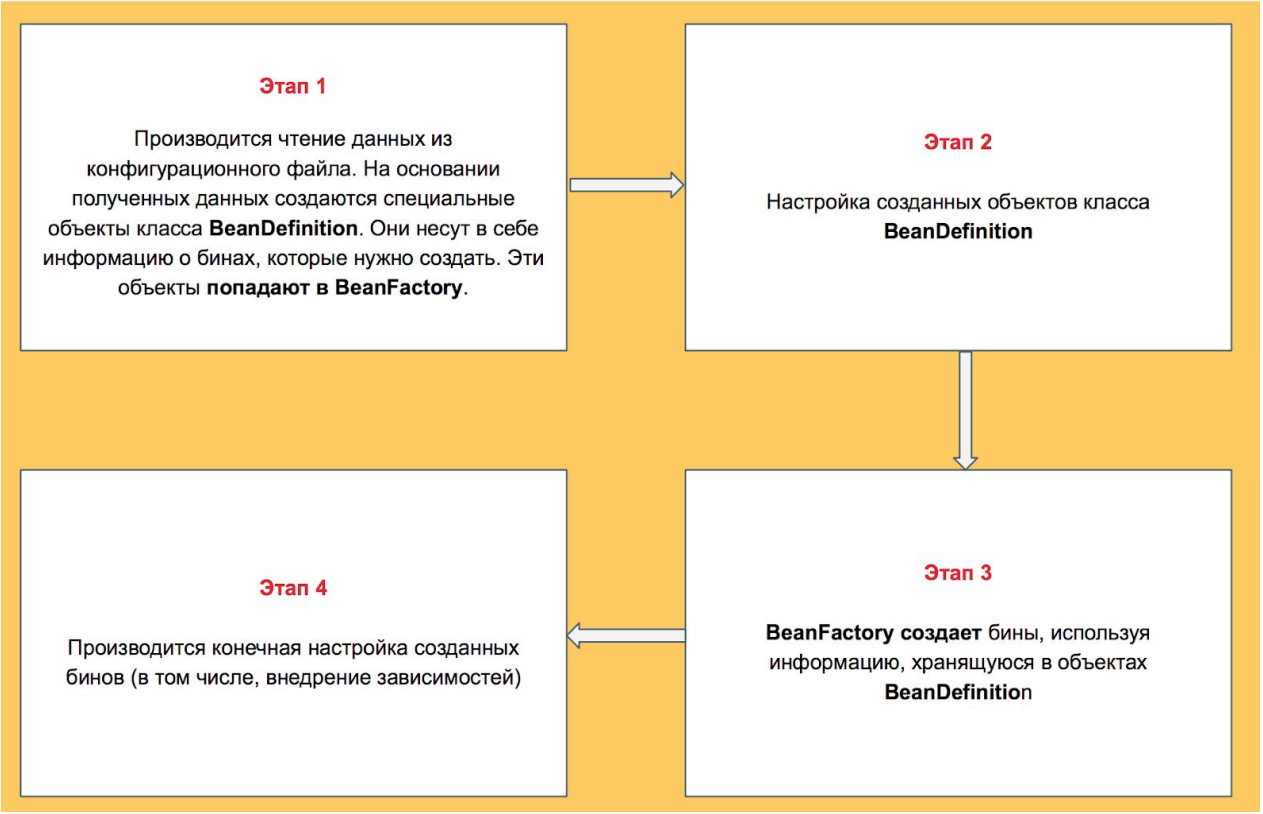
Прежде, чем использовать богатый функционал Spring, компонентам необходимо попасть в контейнер в фреймворке. Если смотреть глобально, то это выглядит следующим образом:



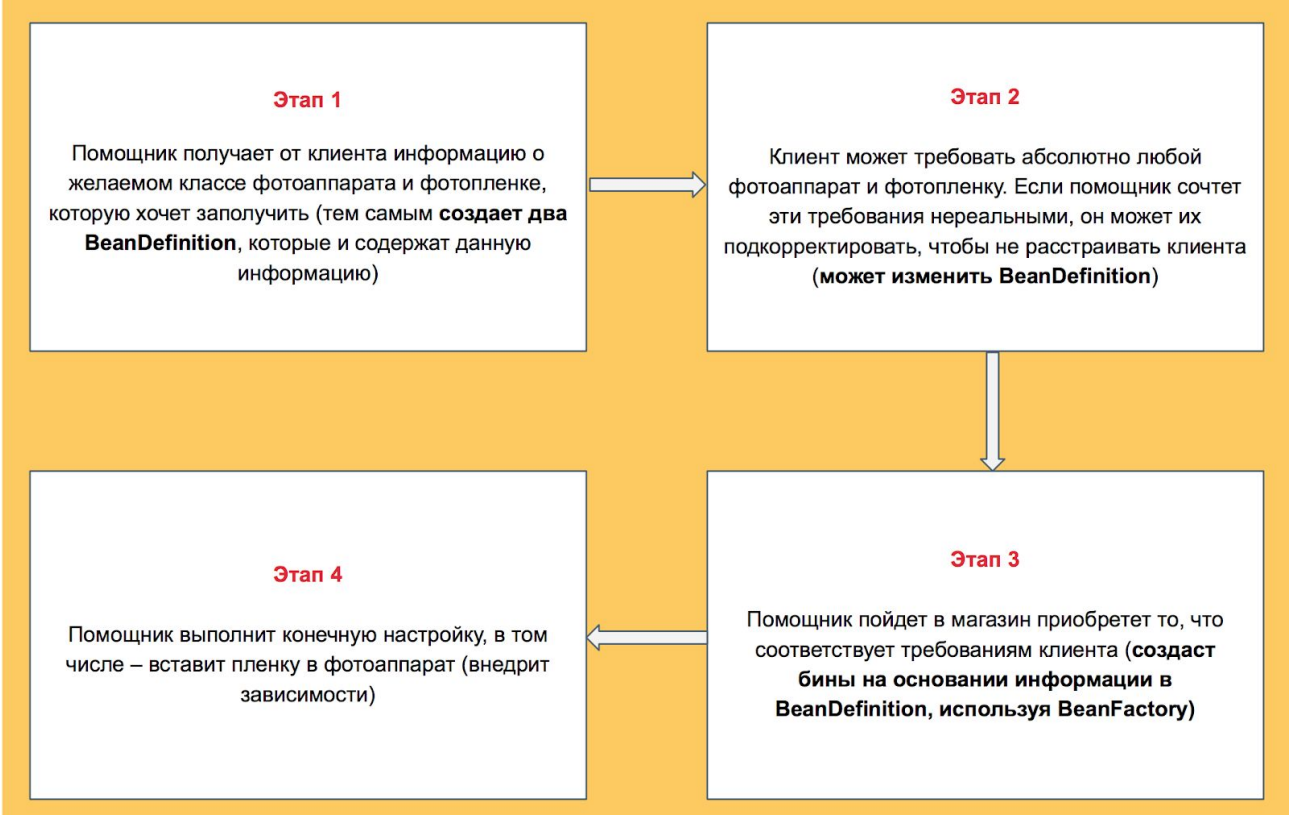
В итоге все бины оказываются в контексте, который является объектом класса **ApplicationContext**.

Но прежде чем туда попасть, они проходят немалый путь, на определенных этапах которого мы можем внести свой вклад в создание бина.

Весь процесс создания и инициализации бина изображен на следующем рисунке:



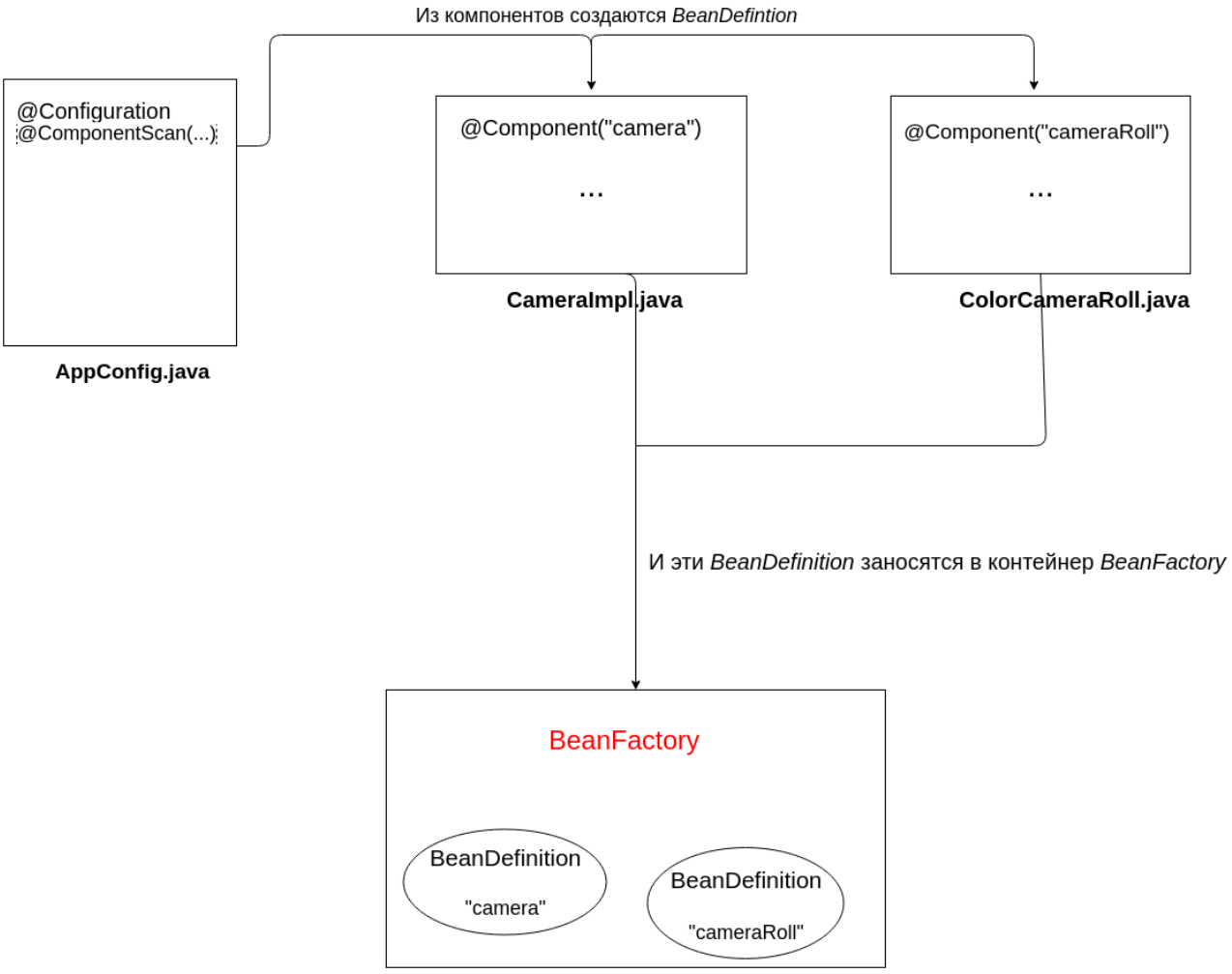
В данной схеме присутствуют два этапа настройки (Этап 2 и Этап 4). Именно на этих этапах идет обработка большей части аннотаций Spring. Чтобы прояснить это, вернемся к примеру с фотоаппаратом. Отразим данный процесс в реальной жизненной ситуации:



Этап 1

На первом этапе в зависимости от вида конфигурации используются различные классы, реализующие интерфейс **ApplicationContext**: **AnnotationConfigApplicationContext**(в случае с **JavaConfig**), **ClassPathXMLApplicationContext** (в случае с **XML**) и т.п. Есть и другие виды конфигурации, но они менее распространены. В конструктор передается параметр, который содержит имя файла конфигурации. Этот процесс полностью соответствует аналогии, согласно которой помощник получает информацию в устной или письменной форме, а итогом является создание **BeanDefinition**, содержащих информацию об объекте, который необходимо будет создать. Все **BeanDefinition** хранятся в определенном контейнере – **BeanFactory**.

Для нашего примера данный процесс изображен на следующем рисунке:



Объект **BeanDefinition** содержит следующую информацию:

- **Class** – полный путь к классу, объектом которого будет являться будущий бин;
- **Scope** – область видимости, о которой мы говорили ранее (по умолчанию **singleton**);
- **Abstract** – указывает, является ли бин абстрактным. Такой бин может быть объектом абстрактного класса, но в случае с XML-конфигурацией класс можно не указывать – достаточно задать лишь свойства. Бины, являющиеся абстрактными, используются только для создания дочерних бинов;
- и другие.

О других свойствах можно прочитать в дополнительном материале к уроку.

Кроме **BeanDefinition**, описывающих пользовательские компоненты, в этом контейнере будут содержаться и **BeanDefinition**, описывающие служебные бины Spring.

Именно на данном этапе Spring обрабатывает уже известные нам аннотации: **@Configuration**, **@ComponentScan**, **@Component**, **@Bean** и другие.

На этом этапе вмешиваться в работу Spring не стоит. Итогом этапа является определенное количество созданных объектов **BeanDefinition**, содержащих информацию о том, какие объекты и с какими параметрами необходимо создать.

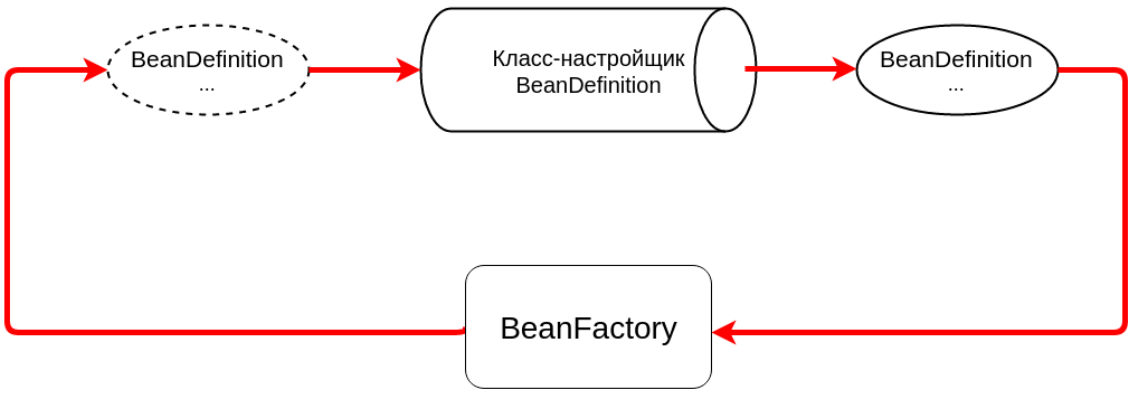
Этап 2

На втором этапе происходит настройка созданных **BeanDefinition**. Обрабатываются только те **BeanDefinition**, классы которых помечены определенными аннотациями, или аннотации, имеющие определенные параметры – например, **@Value("${property.password}"** (подробнее о том, что означает данный параметр, мы поговорим на следующих уроках).

Каким образом происходит эта настройка? Что необходимо сделать, чтобы получить доступ ко всем **BeanDefinition**? Нужен доступ к контейнеру, в котором они хранятся – **BeanFactory** (хотя правильнее будет сказать, что он является объектом класса, реализующим интерфейс **BeanFactory**). Для настройки потребуется класс-настройщик, написанный нами. Но каким образом в него попадет **BeanFactory**? Нам не придется выискивать данный контейнер – Spring сам предоставит **BeanFactory** классу-настройщику. Для этого необходимо выполнить два условия:

- Класс-настройщик должен быть компонентом Spring (т.е. иметь аннотацию **@Component**, а путь к данному классу должен содержаться в конфигурационном файле или в классе нашего приложения);
- Реализовывать интерфейс **BeanFactoryPostProcessor** и его единственный метод **postProcessBeanFactory**, который и принимает в качестве параметра **BeanFactory**.

В общем случае упрощенная схема процесса настройки **BeanDefinition** выглядит так:



Ниже приведен пример реализации собственного класса-настройщика, который выводит информацию о всех **BeanDefinition**, содержащихся в BeanFactory:

```
@Component
public class TestBFPP implements BeanFactoryPostProcessor {

      public      void      postProcessBeanFactory(ConfigurableListableBeanFactory
beanFactory) throws BeansException {
          // Получение имен  BeanDefinition всех бинов, объявленных пользователем
          String[] beanDefinitionNames = beanFactory.getBeanDefinitionNames();
          // Перебор массива для получения доступа к каждому имени
        for(String name: beanDefinitionNames){
          // Получение BeanDefinition по имени (иначе никак!)
          BeanDefinition beanDefinition = beanFactory.getBeanDefinition(name);

          // Вывод информации о BeanDefinition
          System.out.println(beanDefinition.toString());
        }
   }
}
```

При запуске можно увидеть, что объект **BeanDefinition** на самом деле больше, чем ожидалось. Это связано с тем, что Spring использует собственные служебные бины для настройки компонентов, помеченных определенными аннотациями (например, **BeanDefinition**, описывающего бин класса **TestBFPP**, который приведен выше).

*Главное, что стоит запомнить новичку на данном этапе – создание классов, реализующих **BeanFactoryPostProcessor**, позволит повлиять на то, каким будет бин еще до его создания.*

Вернемся к примеру с фотоаппаратом – чтобы понять, как Spring на данном этапе обрабатывает некоторые из своих аннотаций, а также ощутить всю мощь **BeanFactoryPostProcessor**.

Представим ситуацию: клиент в своих требованиях указывает помощнику, что хочет фотографировать на черно-белую пленку. Но она уже давно не производится – вместо нее все применяют цветную. Помощник это понимает и покупать будет именно цветную пленку, тем самым вмешиваясь в требования клиента.

Отредактируем наш код. Раз клиент требует черно-белую пленку, то аннотацию **@Component** стоит **удалить** из класса **ColorCameraRoll** и **добавить** в класс **BlackAndWhiteCameraRoll**. Ведь клиенту цветная пленка совсем не интересна – значит, не обязательно класс **ColorCameraRoll** объявлять компонентом Spring:

```
@Component("cameraRoll")
public class BlackAndWhiteCameraRoll implements CameraRoll {


      public void processing(){

            System.out.println("-1 черно-белый кадр");

      }
}
```

Не каждый помощник будет знать о том, что черно-белая пленка уже давно не производится. Значит, класс этой фотопленки должен быть как-то помечен. Единственным верным способом будет создание

аннотации. Она должна недвусмысленно говорить помощнику о том, что этот вид фотопленки больше не производится и нужно искать другую.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface UnproducableCameraRoll {

	Class usingCameraRollClass();
}
```

Применение данной аннотации означает, что фотопленка этого класса больше не производится. Значение поля **usingCameraRollClass** должно указывать на класс пленки, которую следует использовать вместо данной.

Применим эту аннотацию к классу черно-белой фотопленки:

```
@Component("cameraRoll")
@UnproducableCameraRoll(usingCameraRollClass=ColorCameraRoll.class)
public class BlackAndWhiteCameraRoll implements CameraRoll {


	public void processing(){

		System.out.println("-1 черно-белый кадр");

	}
}
```

Значением поля **usingCameraRollClass** будет класс цветной фотопленки.

Теперь необходимо написать класс-настройщик, который будет обнаруживать данную аннотацию и изменять класс фотопленки, указанный в **BeanDefinition**, на класс, указанный в параметре аннотации **@UnproducableCameraRoll**. С учетом тех требований, которые выдвигаются к подобному классу, он будет иметь следующий вид:

```
@Component
public class UnproducableCameraRollBeanFactoryPostProcessor implements
BeanFactoryPostProcessor {
      public void postProcessBeanFactory(ConfigurableListableBeanFactory
beanFactory) throws BeansException {
            // Получаем имена всех BeanDefinition для доступа к каждому из них
            String[] beanDefinitionNames = beanFactory.getBeanDefinitionNames();
            // Перебираем все имена
            for(String name: beanDefinitionNames){
            // Получаем BeanDefinition по имени
                  BeanDefinition beanDefinition =
beanFactory.getBeanDefinition(name);

            /*Получаем имя класса создаваемого бина, чтобы проверить,
             * содержит ли он аннотацию UnproducableCameraRoll
             */
                  String className = beanDefinition.getBeanClassName();

                  try {
            // Получаем класс по имени
                        Class<?> beanClass = Class.forName(className);

            /*Пытаемся получить объект аннотации и ее значение,
             * если  класс не содержит данную аннотацию, то  метод вернет null
             */
            UnproducableCameraRoll annotation =
(UnproducableCameraRoll)beanClass.getAnnotation(UnproducableCameraRoll.class);

            // Проверяем, содержал ли класс эту аннотацию
                        if(annotation!=null){

            // Получаем значение, указанное в параметрах аннотации (класс
пленки, которую необходимо использовать)
                  Class usingCameraRollName =annotation.usingCameraRollClass();

            // Меняем класс будущего бина

beanDefinition.setBeanClassName(usingCameraRollName.getName());
                        }

                  } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
                        e.printStackTrace();
                  }

            }
      }
}
```

Запустив клиентский код, вы сможете убедиться, что помощник действительно подменил класс фотопленки, вопреки требованиям клиента.

Для реализации подобных трюков есть одно очень важное условие. Еще раз взглянем на объявление атрибута, в который «вживляется» фотопленка:

```
@Component("camera")
public class CameraImpl implements Camera {
      @Autowired
      @Qualifier("cameraRoll")
      private CameraRoll cameraRoll;
      …
}
```

Данный атрибут имеет тип интерфейса **CameraRoll**. Его реализуют оба класса фотопленки – значит, независимо от того, какую фотопленку в итоге придется вставить в фотоаппарат, внедрение пройдет успешно.

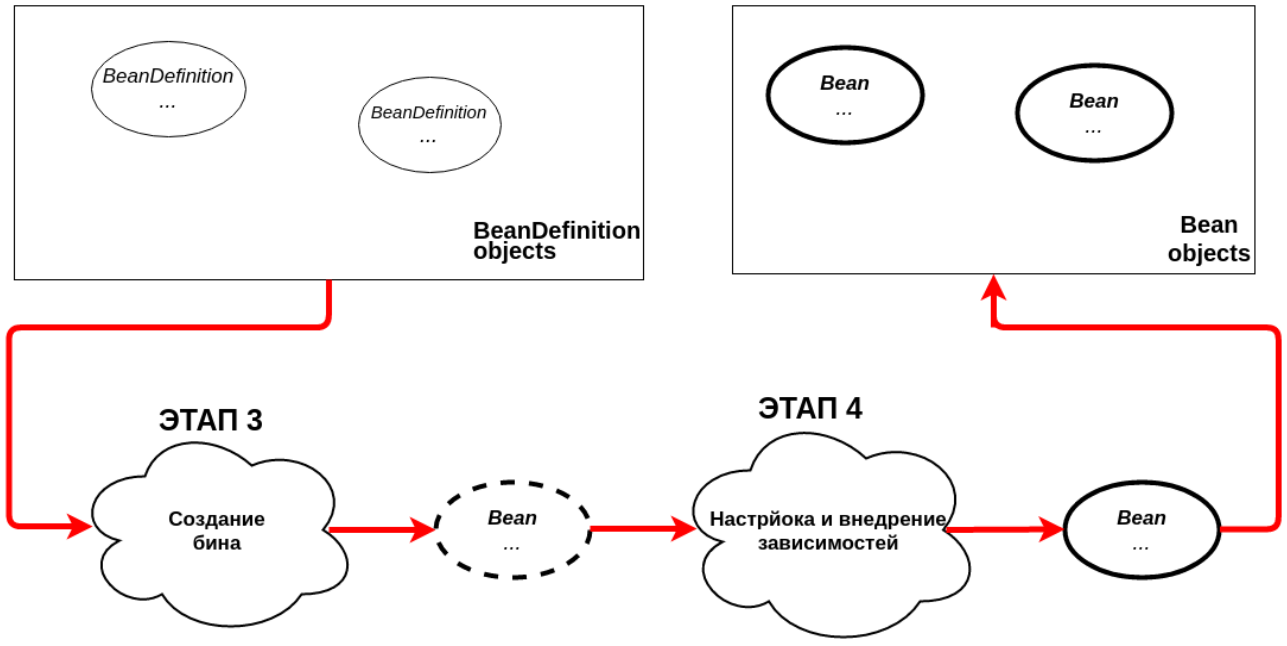
Теперь вспомним строку из нашего клиентского кода:

```
    Camera camera = context.getBean("camera", Camera.class);
```

Бин **camera** класса **CameraImpl** извлекается из контекста по интерфейсу, который реализует данный класс. Это помогает нам не получить исключение в случае, если помощник подменит класс фотокамеры другим классом, который реализует данный интерфейс.

Этап 3

На этом этапе **BeanFactory**, используя **BeanDefinition**, создает бины. Данный процесс является внутренним процессом Spring, и нет смысла влиять на него. Важно отметить, что на этой стадии бины лишь создаются, но их зависимости еще не удовлетворены.



Этап 4

На четвертом этапе мы имеем созданные бины, но зависимости еще не внедрены, в том числе и простые значения, внедряемые аннотацией **@Value**. Значит, бины еще не готовы к использованию. Чтобы внедрить зависимости, Spring использует классы-настройщики, которые и обрабатывают

аннотации **@Value**, **@Autowired** и другие. Мы без проблем можем реализовать собственный класс-настройщик, но для этого он должен выполнять два условия:

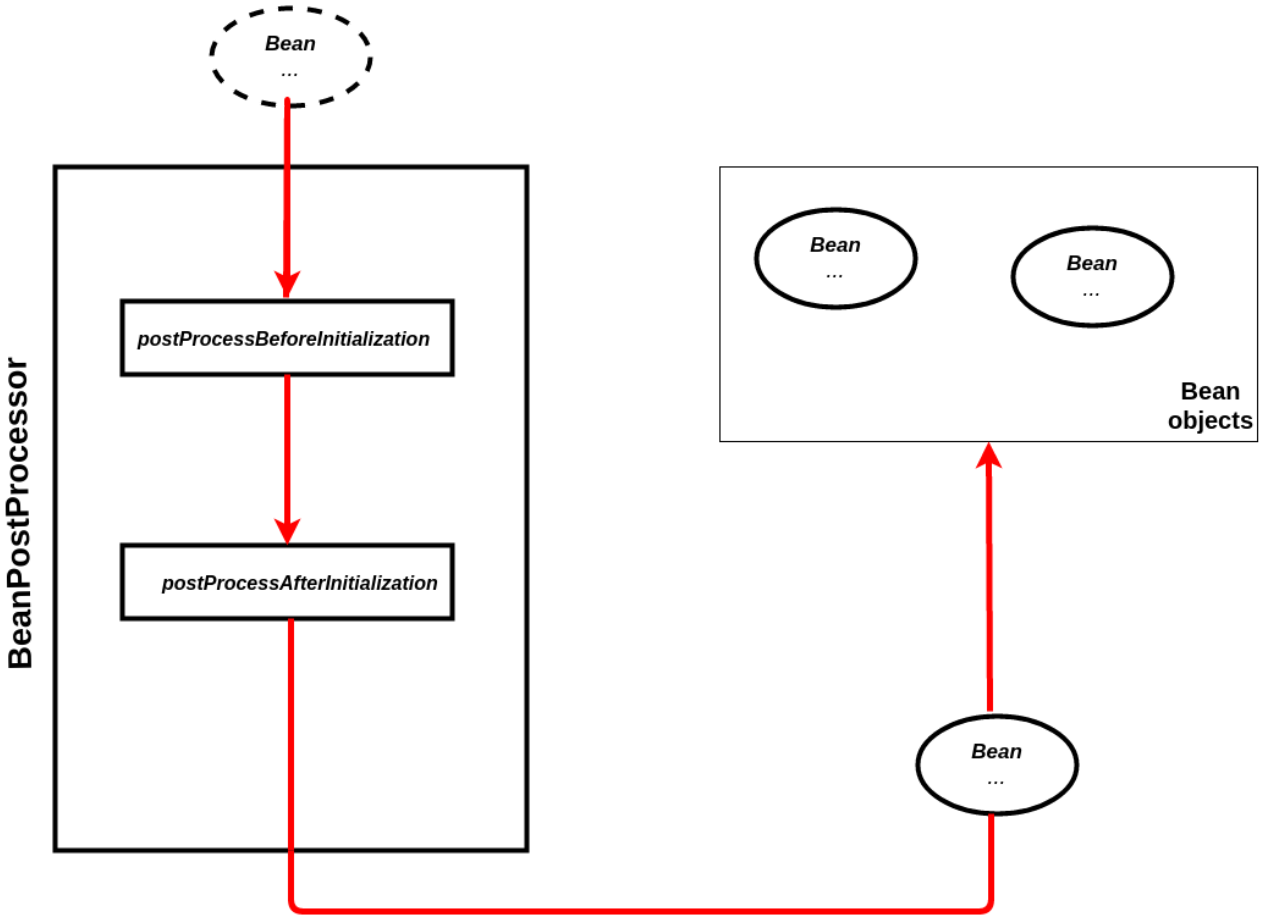
- Класс-настройщик должен являться компонентом Spring (иметь аннотацию **@Component**, а путь к данному классу должен содержаться в конфигурационном файле или классе нашего приложения);
- Реализовывать интерфейс **BeanPostProcessor** и его методы:

```
public interface BeanPostProcessor {

      Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException;

      Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException;

}
```

В отличие от класса, реализующего интерфейс **BeanFactoryPostProcessor**, реализация этого интерфейса позволит получать доступ к каждому бину поочередно. В каждый из методов передается сам бин и его имя. Данные методы имеют следующее описание:

- **postProcessBeforeInitialization** – метод, вызываемый до инициализации бина (термин «инициализация» в данном контексте довольно относителен: для Spring это означает вызов пользовательского init-метода, о котором будет рассказано далее). На данном этапе бин создан, и в него уже внедрены зависимости, которые помечены аннотациями Spring (**@Autowired**, **@Value** и т.п.). Классы-настройщики Spring всегда будут вызываться раньше, чем реализованные пользователем;

- **postProcessAfterInitialization** – метод, который выполняется после инициализации (после вызова init-метода). После него настроенные бины попадают непосредственно в контейнер бинов.



Предположим, что помощник перед покупкой фотоаппарата решил проверить его работоспособность. Для этого ему необходимо самому попробовать сделать фотографию. Для осуществления этой затеи реализуем собственный **BeanPostProcessor**:

```
@Component
public class PhotocameraTestBeanPostProcessor implements BeanPostProcessor {

      public    Object    postProcessBeforeInitialization(Object    bean,    String
beanName) throws BeansException {
            // В данном методе просто возвращаем бин
            return bean;
}

      public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {

            // Находим бин класса фотокамеры
            if(bean instanceof Camera){

            System.out.println("Делаю пробное фото!");
            // Делаем пробное фото
            ((Camera) bean).doPhotograph();
            System.out.println("Отлично! Работает!");
            }
            return bean;
      }
}
```

По завершении этапа бины полностью готовы к использованию.

Жизненный цикл бина

Все, что было рассмотрено ранее, относится к этапам инициализации контекста. Но сам бин об этих этапах ничего не знает. Как это исправить?

Бины Spring обладают жизненным циклом. Фактически, это дает возможность вызывать собственные методы бина на его жизненных этапах. Чтобы подобное стало возможным, необходимо как-то пометить метод и время его вызова. На практике используется несколько подходов.

Первый подход использует аннотации:

- **@PostConstruct** – метод инициализации, вызываемый после создания объекта и внедрения зависимостей (т.е. между методами **postProcessBeforeInitialization** и **postProcessAfterInitialization** интерфейса **BeanPostProcessor**);
- **@PreDestroy** – метод, вызываемый перед уничтожением бина.

Второй подход использует XML-атрибуты тега **<bean>**:

- **init-method**;
- **destroy-method**.

Оба подхода являются полными аналогами друг друга. Но если в вашем приложении используются оба подхода для одного и того же бина, то первыми будут вызываться методы, помеченные аннотацией.

Применение этих аннотаций возможно благодаря поддержке Spring стандарта JSR-250. Соответственно, для использования **@PostConstruct** и **@PreDestroy** необходимо подключать дополнительную зависимость.

В примере с фотокамерой реализуем метод инициализации, который выводит уведомление о том, что фотоаппарат готов к использованию.

Чтобы использовать аннотации жизненного цикла (в частности, **@PostConstruct**), необходимо добавить в файл **pom.xml** следующую зависимость:

```
 <dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>jsr250-api</artifactId>
    <version>1.0</version>
</dependency>
```

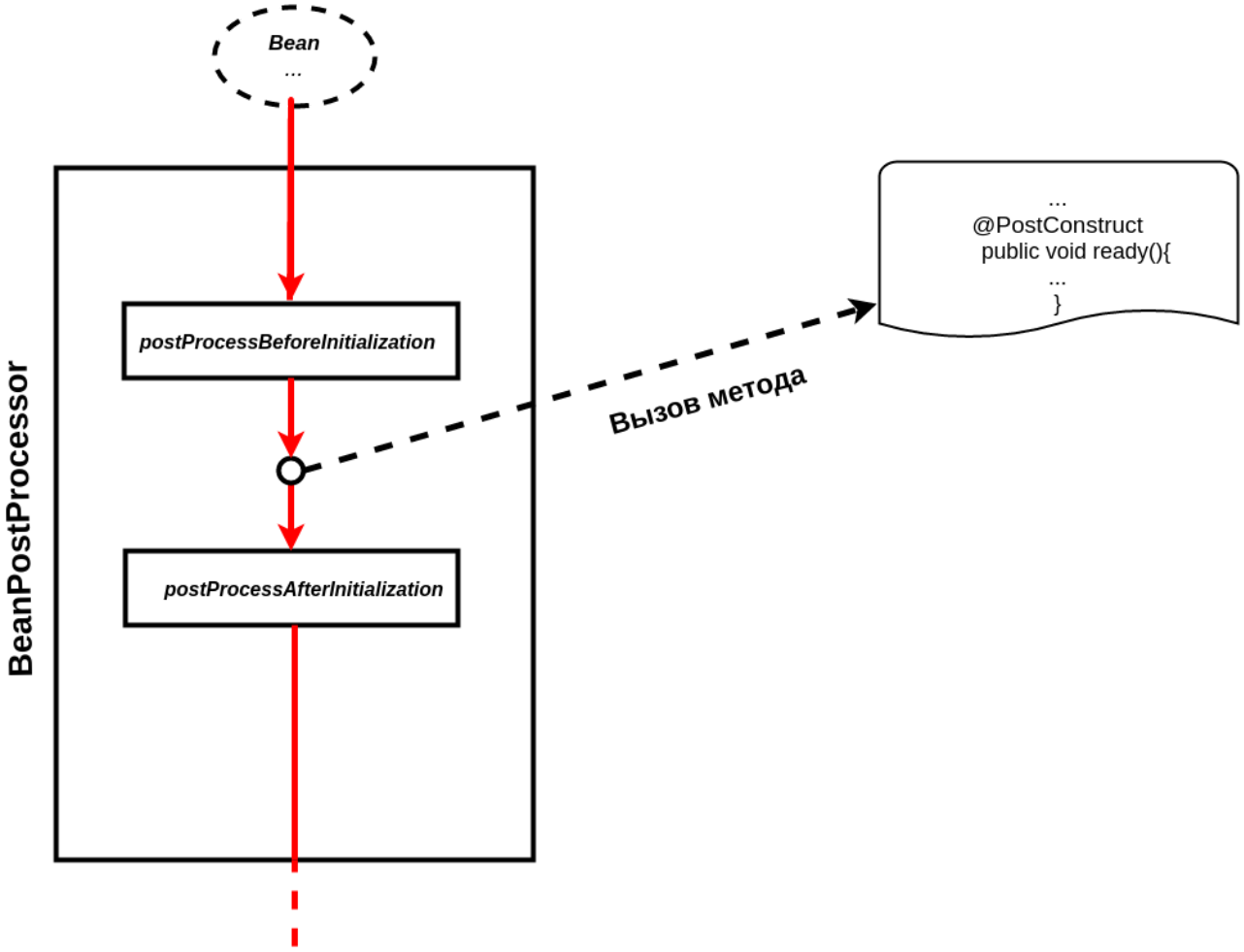
В интерфейсе **Camera** объявим новый метод:

```
public void ready();
```

Реализуем этот метод в классе **CameraImpl** и пометим его аннотацией:

```
@Component("camera")
public class CameraImpl implements Camera {

	@Autowired
	private CameraRoll cameraRoll;

	@Value("false")
	private boolean broken;

	public CameraRoll getCameraRoll() {
		return cameraRoll;
	}

	public void setCameraRoll(CameraRoll cameraRoll) {
		this.cameraRoll = cameraRoll;
	}

	public boolean isBroken() {
		return broken;
	}

   public void breaking(){

	   this.broken=true;
   }

	public void doPhotograph(){

		if(isBroken()){

			System.out.println("Фотоаппарат сломан!");
			return;
		}
		    System.out.println("Сделана фотография!");
		    cameraRoll.processing();
	}
	@PostConstruct
	public void ready() {

		System.out.println("Фотоаппарат готов к использованию!");
	}
}
```

При запуске клиентского кода получим оповещение о готовности к использованию. Данный метод будет вызван между двумя методами интерфейса **BeanPostProcessor** следующим образом:



Заключение

В этом уроке мы рассмотрели много вещей, которые зачастую остаются «за кадром» для рядового разработчика. Не всегда вмешательство в данные процессы является целесообразным. Но знание того, что происходит «под капотом» отличает разработчика-профессионала от программиста-любителя. Если вы понимаете, каким образом работает фреймворк, то Spring в ваших руках – это не просто инструмент, а грозное оружие! К тому же, материал этого урока может выручить вас на собеседованиях. В следующих уроках мы будем создавать реальное приложение, и эти знания позволят вам чувствовать себя вполне уверенно.

Код учебного проекта

Конечный вариант кода программы, разработанной в ходе данного урока: https://github.com/Firstmol/Geekbrains-lesson2

В коде учебного проекта мы полностью отказались от XML-конфигурации, потому файл **config.xml** был удален из исходного кода.

Практическое задание

1. Запустить код учебного проекта и убедиться в правильности его работы.
2. Изменить код таким образом, чтобы клиент требовал цветную фотопленку, а класс-настройщик менял класс бина на черно-белую.
3. * Реализовать собственный подкласс фотоаппарата, написать аннотацию и класс-настройщик (реализующий интерфейс **BeanFactoryPostProcessor**), который будет подменять класс фотоаппарата подобно тому, как это сделано с фотопленкой в учебном примере данного урока.

Дополнительный материал

1. Этапы инициализации контекста: https://habrahabr.ru/post/222579/;
2. Видеолекция «Spring-потрошитель»: https://www.youtube.com/watch?v=BmBr5diz8WA;
3. Подробнее о жизненном цикле и интерфейсе **InitializingBeans**: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html;
4. Крис Шефер, Кларенс Хо. 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.