Java для молодых специалистов Банка ВТБ

Урок 13

Spring Core

Spring Framework. Понятие внедрения зависимостей и инверсии управления. Что такое bean в Spring? Понятие контекста Spring. Конфигурирование Spring (XML, аннотации)

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

Область видимости бинов

Этапы инициализации контекста

- Этап 1
- Этап 2
- Этап 3

Этап 4

По завершении этапа бины полностью готовы к использованию.

Жизненный цикл бина

Практическое задание

Дополнительные материалы

Используемая литература

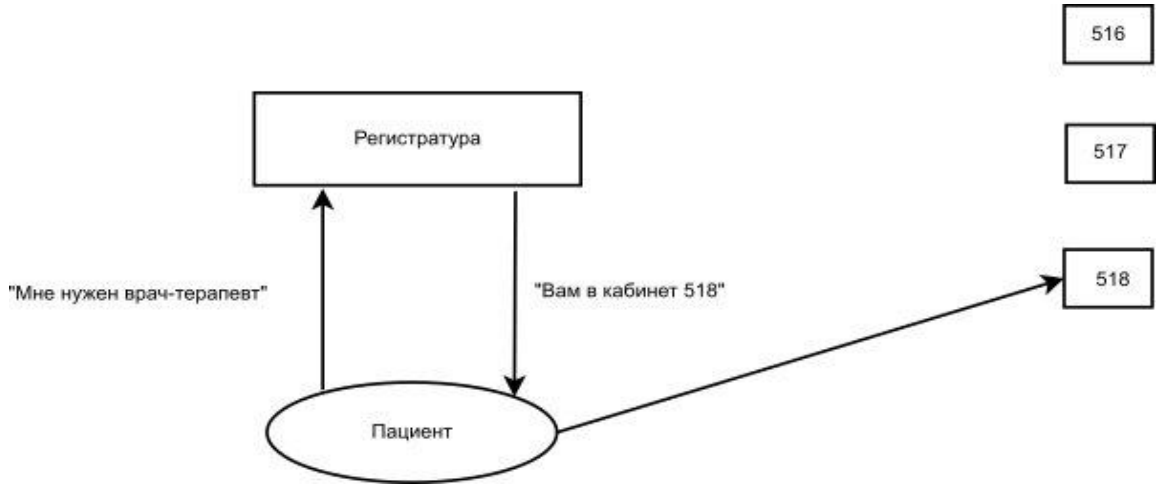
Что такое Spring?

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

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 одну из них — внедрение зависимостей.

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 регистратор (например, «мне нужен терапевт»), вам дадут информацию. Конечный выбор номера кабинета остается не за вами, а за регистратором, — 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 представлен 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. 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:

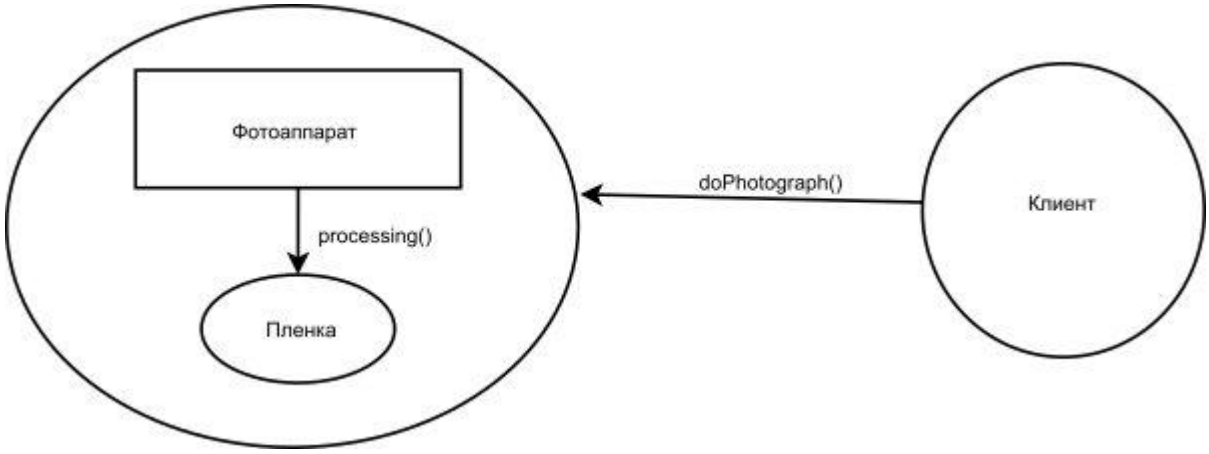
```
public class Camera {
  private CameraRoll cameraRoll;
```

```
  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. При чем 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 получаем. Чтобы сделать снимок, необходима пленка внутри фотоаппарата. Изменим его код, 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 этом любую понравившуюся нам фотопленку, мы больше не сможем поменять ее, даже когда она закончится. А хотелось бы иметь возможность вставлять в аппарат и доставать из него любую фотопленку сколько угодно раз. Используем для этого **get** и **set**. Перепишем 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. А чтобы использовать черно-белую пленку, необходимо написать ее интерфейс и две 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).

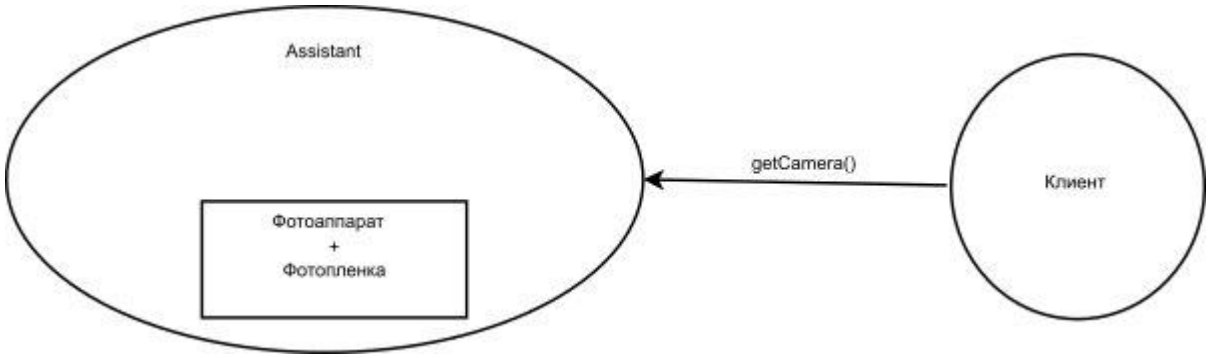
Теперь фотоаппарат умеет работать с несколькими видами пленки, и менять мы ее можем в любое время.

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: создание 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. Подобный контейнер состоит из двух компонентов:

- **контекста**, который хранит созданные объекты с уже внедренными зависимостями (в нашем примере — объект фотоаппарата);
- **дополнительных служб**, благодаря которым создаются объекты и внедряются зависимости. Но есть и множество других служб, которые определяют функциональность Spring. Главная особенность фреймворка в том, что он реализует модульную архитектуру, где каждый модуль представляет собой JAR-файл и наделен определенной функциональностью. Можно подключать только те модули, которые нужны для решения конкретной задачи.

Для начала работы со Spring необходимо создать пустой Maven-проект и подключить зависимости в **pom.xml**. Все зависимости будем подключать с mvnrepository.com.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>5.0.8.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.8.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.

Теперь у нас есть все, чтобы реализовать предыдущий пример, но уже с использованием Spring. В предыдущим коде создадим интерфейс **Camera** с единственным методом **doPhotograph()**, а наш предыдущий класс будет называться **CameraImpl** и 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? Для создания контекста, который играет роль нашего ассистента, необходимо передать 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.

Конфигурирование Spring

В упрощенном виде конфигурация Spring проходит в два этапа:

- инициализация контекста;
- создание, настройка, добавление компонентов (бинов) в контекст.

Виды конфигураций в Spring:

- XML-конфигурация;
- JavaConfig;
- использование аннотаций.

Можно сделать вывод, что Spring обладает двумя самостоятельными способами конфигураций. А третий (использование аннотаций) может применяться только на втором этапе совместно с одним из видов «самостоятельной» конфигурации. То есть можем применять:

- либо XML-конфигурацию + использование аннотаций,
- либо JavaConfig + использование аннотаций,
- или два «самостоятельных» способа конфигурации без аннотаций.

Рассмотрим каждый из этих способов.

Конфигурирование с использованием XML

Конфигурация через XML — первый вариант, который поддерживал Spring. 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="ru.geekbrains.lesson1.ColorCameraRoll" />

  <bean id="camera" class="ru.geekbrains.lesson1.CameraImpl">
    <property name="cameraRoll">
      <ref bean="cameraRoll" />
    </property>
  </bean>
</beans>
```

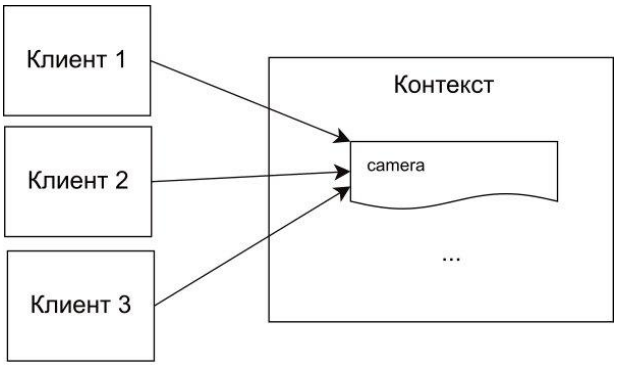
Конфигурационный файл предназначен для указания контейнеру объектов, которыми он должен управлять (бинов). Основной схемой и пространством имен по умолчанию является http://www.springframework.org/schema/beans/spring-beans.xsd и http://www.springframework.org/schema/beans.

Тег **<bean … />** позволяет:

- создать объект определенного класса, который сразу же станет бином;

- внедрить объект другого класса в свойство данного бина, при этом данный 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="ru.geekbrains.lesson1.ColorCameraRoll" />
```

В данной строке с помощью тега **bean** объявляется объект класса **ColorCameraRoll**. 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, но есть особенность. Ранее мы оговаривали, что 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 его 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="ru.geekbrains.lesson1.CameraImpl">
```

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

В данном случае происходит создание объекта класса **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, то между двумя тегами **property** необходимо поместить ссылку (reference) на ранее созданный **bean**. Для этого и используется тег **ref** с атрибутом **bean**, в котором 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="ru.geekbrains.lesson1.CameraImpl">
    <property name="cameraRoll">
      <bean  class="ru.geekbrains.lesson1.ColorCameraRoll" />
    </property>
  </bean>
</beans>
```

Данный вариант эквивалентен следующему Java-коду:

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

Конфигурирование с использованием аннотаций

С версии Spring 2.5 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() {
    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 пространство имен (http://www.springframework.org/schema/context/spring-context.xsd) и схему (http://www.springframework.org/schema/context);

2. Чтобы Spring-контейнер «знал», где искать объявленные с помощью аннотаций компоненты, в config-файл добавляем строку:

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

```
    </property>
  </bean>
</beans>
```

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="ru.geekbrains.lesson1.ColorCameraRoll" />

<bean id="camera" class="ru.geekbrains.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.

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

```
@Configuration
@ComponentScan("ru.geekbrains.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="ru.geekbrains.lesson1" />
<bean id="camera" class="ru.geekbrains.lesson1.CameraImpl">
  <property name="cameraRoll">
    <ref bean="cameraRoll" />
  </property>
</bean>
```

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:

1. Используется Java-код, а значит, не нужно заботиться о xsd-схемах и XML-тегах.
2. Можно выявить ошибки на этапе написания конфигурационного класса.
3. 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 сеттер;
- внедрение на уровне поля.

Почти во всех случаях предпочтительным является внедрение через сеттер. 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 подобное);
- 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 сеттер. Если применять только XML-конфигурацию, объявление бина и обеспечение внедрения зависимости типа String *через конструктор* будет выглядеть следующим образом:

```
<bean id="helloMan" class="ru.geekbrains.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="ru.geekbrains.lesson1.HelloManOnceSay">
  <constructor-arg type="java.lang.String" index="0" value="Yuri" />
</bean>
```

Внедрение *через сеттер* (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 сеттер 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 сеттеру **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 (id, идентификатор) нашего бина.

Для внедрения *через сеттер* применяется следующая конструкция:

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

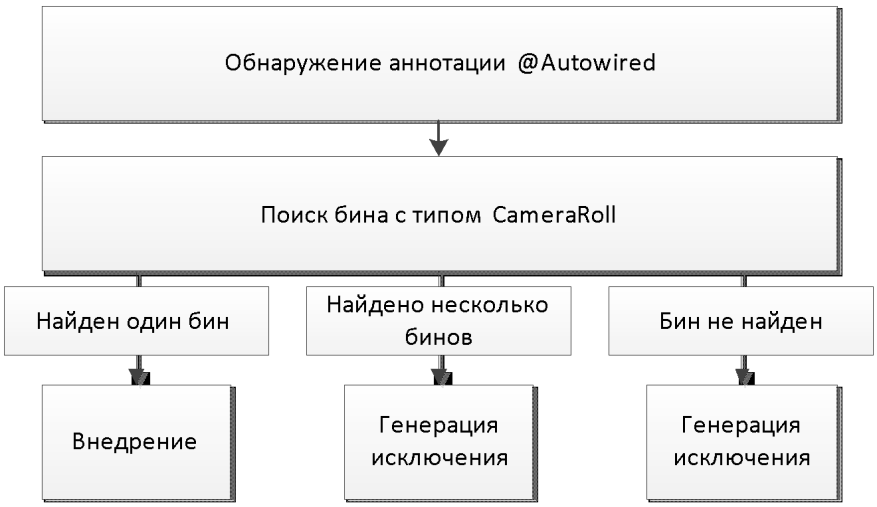
Для внедрения в JavaConfig применяется конструкция, которую мы уже рассматривали:

```
@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 сеттер. 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. Необходимо использовать аннотацию **@Autowired**. Перепишем класс камеры, применяя ее:

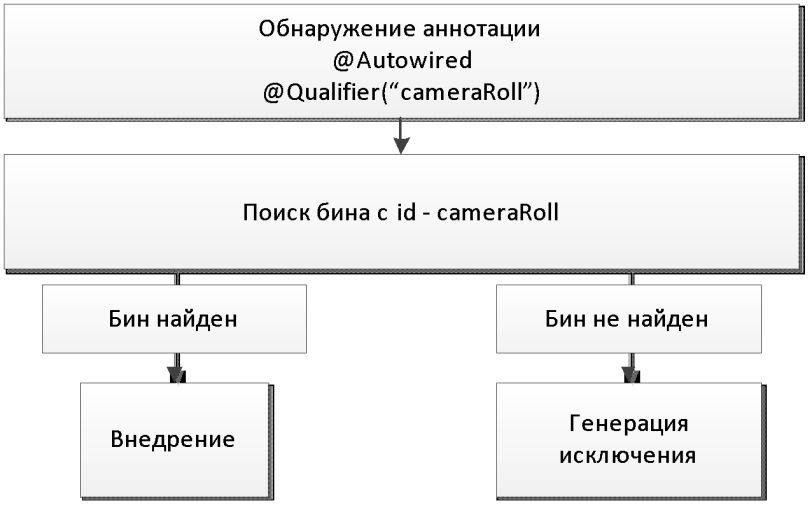
```
@Component("camera")
public class CameraImpl implements Camera {
  private CameraRoll cameraRoll;

  public CameraRoll getCameraRoll() {
    return cameraRoll;
  }

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

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

Эта аннотация может применяться к полю, сеттеру и конструктору. В данной программе алгоритм внедрения будет работать следующим образом:



Из схемы видно, что поиск бина, который станет кандидатом на внедрение, осуществляется по типу. Если в контексте есть несколько бинов заданного типа или бин не будет найден, то генерируется исключение **NoSuchBeanDefinitionException**. В обоих случаях 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 с 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 id бина, который должен быть внедрен.



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:

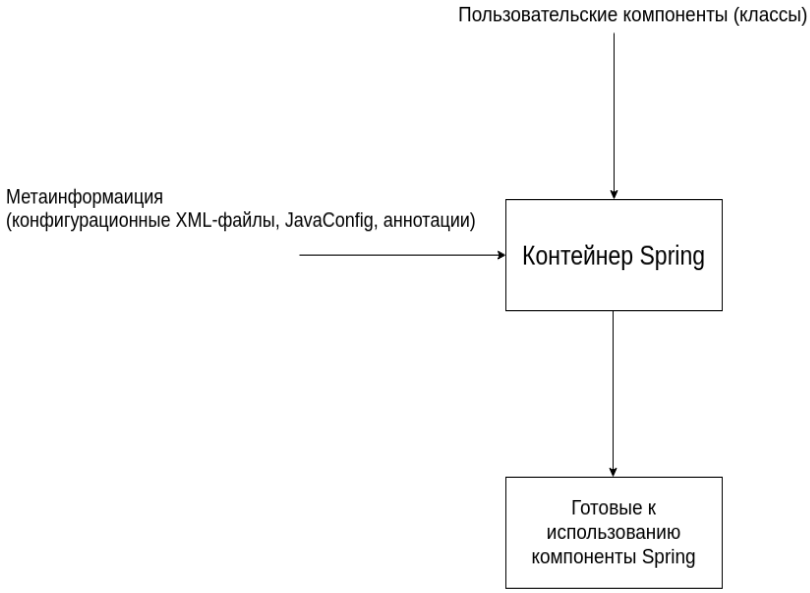
```
@Component("camera")
@Scope("prototype")
public class CameraImpl implements Camera {
    // ...
}
```

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

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 делится на четыре этапа:

Этап 1. Производится чтение данных из конфигурационного файла. На основании полученных данных создаются специальные объекты BeanDefinition, которые несут в себе информацию о бинах, которые надо создать. Эти объекты попадают в BeanFactory.

Этап 2. Производится настройка объектов класса BeanDefinition.

Этап 3. Производится конечная настройка созданных бинов (в том числе, внедрение зависимостей)

Этап 4. BeanFactory создает бины, используя информацию, хранящуюся в объектах BeanDefinition.

В данной схеме присутствуют два этапа настройки (Этап 2 и Этап 4). Именно на этих этапах идет обработка большей части аннотаций Spring.

Этап 1

На первом этапе в зависимости от вида конфигурации используются различные классы, реализующие интерфейс **ApplicationContext**, при создании которых, в конструктор передается ссылка на файл конфигурации(это может быть xml файл, или Java класс). После подготовки объекта типа ApplicationContext, формируются объекты типа **BeanDefinition**, содержащие информацию об объектах, которые необходимо будет создать. Все **BeanDefinition** хранятся в определенном контейнере – **BeanFactory**. Объект **BeanDefinition** содержит следующую информацию:

- **Class** – полный путь к классу, объектом которого будет являться будущий бин;
- **Scope** – область видимости (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.

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**. 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**.

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

```
@Component
public class TestBeanFactoryPostProc 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
      BeanDefinition beanDefinition = beanFactory.getBeanDefinition(name);
      // Вывод информации о BeanDefinition
      System.out.println(beanDefinition.toString());
    }
  }
}
```

При запуске можно увидеть, что объектов **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 **TestBeanFactoryPostProc**, который приведен выше).

*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) {
        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. 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**.

Практическое задание

1. Создайте класс Product (id, title, cost);
2. Создайте компонент ProductService, который хранит в себе List<Product>, допустим с 10 видами товаров. В ProductService должен быть метод вывода всех товаров в консоль printAll(), получения ссылки на Product по имени findByTitle(String title);
3. Создайте компонент Cart (корзина) с возможностью добавления туда товаров add(Product product);
4. Создайте компонент OrderService, позволяющий из корзины сформировать заказ. Под формирование заказа подразумевается распечатка всех позиций в консоли, с выводом итоговой стоимости выбранных товаров.

Не надо прописывать ввод данных с консоли. Достаточно набросать немного кода, демонстрирующего выполнение действий с указанными выше компонентами. Базу данных прикручивать к проекту тоже не надо.

В ProductService у вас будет List<Product> для его заполнения не стоит использовать конструктор, как бы мы это делали в обычном проекте. Вместо этого пропишите метод с аннотацией @PostConstruct, который сработает после подготовки бина к работе, и в нем сделайте всю необходимую подготовительную работу.

Дополнительные материалы

1. Кей С. Хорстманн, Гари Корнелл Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. — М.: Вильямс, 2014. — 864 с.
2. Брюс Эккель. Философия Java // 4-е изд.: Пер. с англ. — СПб.: Питер, 2016. — 1 168 с.
3. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.
4. Г. Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 720 с.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.