Lesson 29

12.07.2021

```
public class Test1 {
    static boolean foo (char c) {
        System.out.println(c);
        return true;
    }

    public static void main(String[] args) {
        int i = 0;
        for (foo('A'); foo('B') && (i < 3); foo('C')) {
            i++;
            foo('D');
        }
    }
}
```

```
public class Test2 {
    public static void main(String[] args) {
        Boolean[] boo = new Boolean[3];
        boo[0] = new Boolean(Boolean.parseBoolean("tRUE"));
        boo[1] = new Boolean("True");
        boo[2] = new Boolean(false);
    }
}
```


spring

Spring Framework (или коротко **Spring**) — универсальный фреймворк с открытым исходным кодом для Java-платформы.

Фреймворк был впервые выпущен под лицензией Apache 2.0 license в июне 2003 года.

Первая стабильная версия 1.0 была выпущена в марте 2004.

Spring 2.0 был выпущен в октябре 2006,

Spring 2.5 — в ноябре 2007,

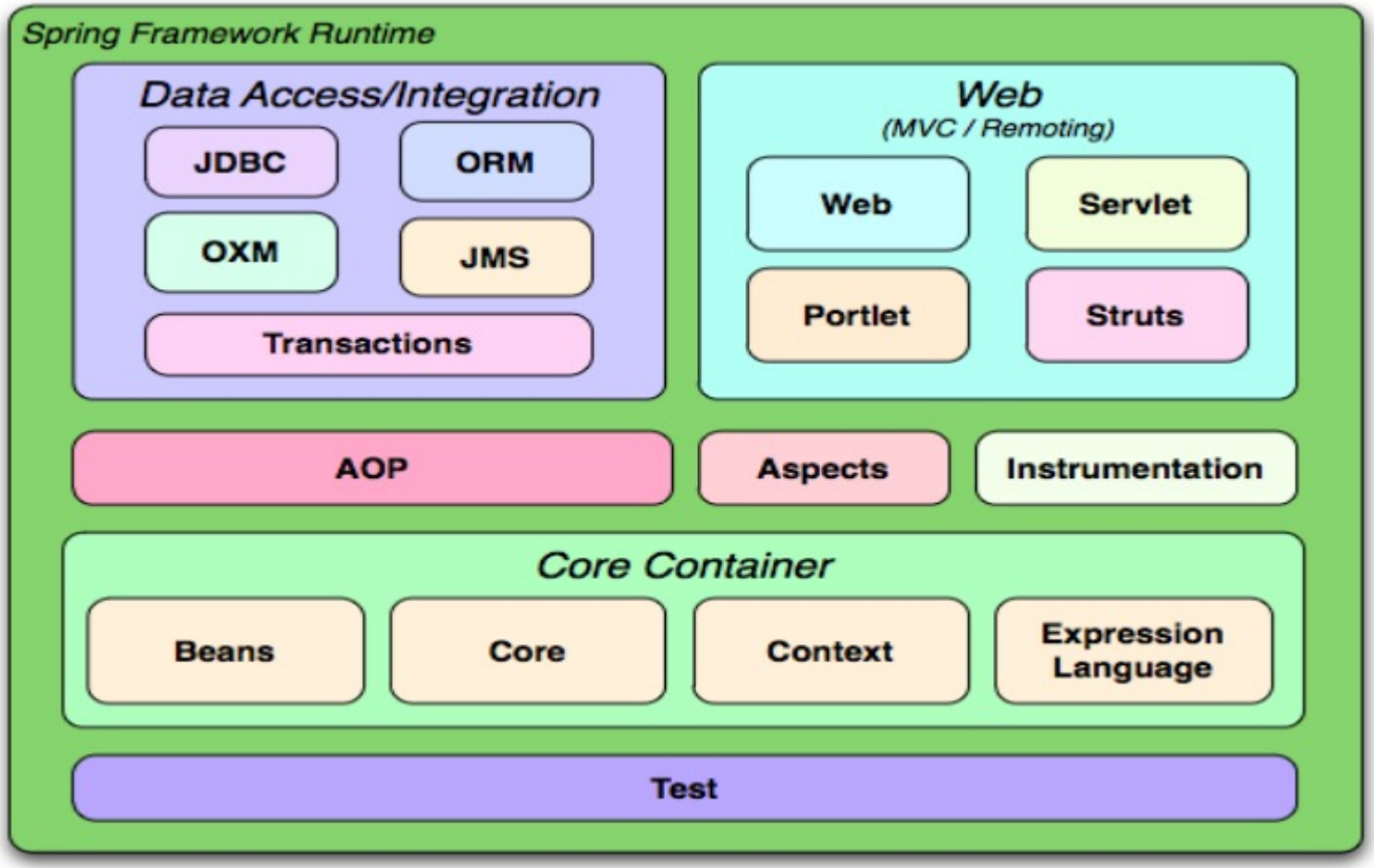
Spring 3.0 в декабре 2009,

Spring 3.1 в декабре 2011.

Текущая версия — 5.2.2.

Несмотря на то, что Spring не обеспечивал какую-либо конкретную модель программирования, он стал широко распространённым в Java-сообществе главным образом как альтернатива и замена модели Enterprise JavaBeans.


Spring MVC
Spring WebFlow
Spring Social
Spring Batch
Spring Framework
Spring Security
Spring Data
Spring Integration
Spring Web Services
And ...


Spring Framework Runtime
Data Access/Integration
JDBC
ORM
OXM
JMS
Transactions
Web
(MVC / Remoting)
Web
Servlet
Portlet
Struts
AOP
Aspects
Instrumentation
Core Container
Beans
Core
Context
Expression Language
Test

Основной контейнер (Core Container) включает в себя Beans, Core, Context и SpEL (expression language).

Beans отвечает за BeanFactory которая является сложной реализацией паттерна Фабрика (GoF).

Модуль ***Core*** обеспечивает ключевые части фреймворка, включая свойства IoC и DI.

Context построен на основе Beans и Core и позволяет получить доступ к любому объекту, который определён в настройках. Ключевым элементом модуля *Context* является интерфейс ApplicationContext.

Модуль ***SpEL*** обеспечивает мощный язык выражений для манипулирования объектами во время исполнения.

Контейнер **Data Access/Integration** состоит из JDBC, ORM, OXM, JMS и модуля Transatcions.

JDBC *обеспечивает* абстрактный слой JDBC и избавляет разработчика от необходимости вручную прописывать монотонный код, связанный с соединением с БД.

ORM обеспечивает интеграцию с такими популярными ORM, как Hibernate, JDO, JPA и т.д.

Модуль ***OXM*** отвечает за связь Объект/XML – XMLBeans, JAXB и т.д.

Модуль ***JMS*** *(Java Messaging Service)* отвечает за создание, передачу и получение сообщений.

Transactions поддерживает управление транзакциями для классов, которые реализуют определённые методы.

Web

Этот слой состоит из Web, Web-MVC, Web-Socket, Web-Portlet

Модуль *Web* обеспечивает такие функции, как загрузка файлов и т.д.

Web-MVC содержит реализацию Spring MVC для веб-приложений.

Web-Socket обеспечивает поддержку связи между клиентом и сервером, используя Web-Socket-ы в веб-приложениях.

Web-Portlet обеспечивает реализацию MVC с среде портлетов.

Spring также включает в себя ряд других важных модулей, таких как AOP, Aspects, Instrumentation, Messaging и Test

AOP реализует аспекто-ориентированное программирование и позволяет использовать весь арсенал возможностей АОП.

Модуль ***Aspects*** обеспечивает интеграцию с AspectJ, которая также является мощным фреймворком АОП.

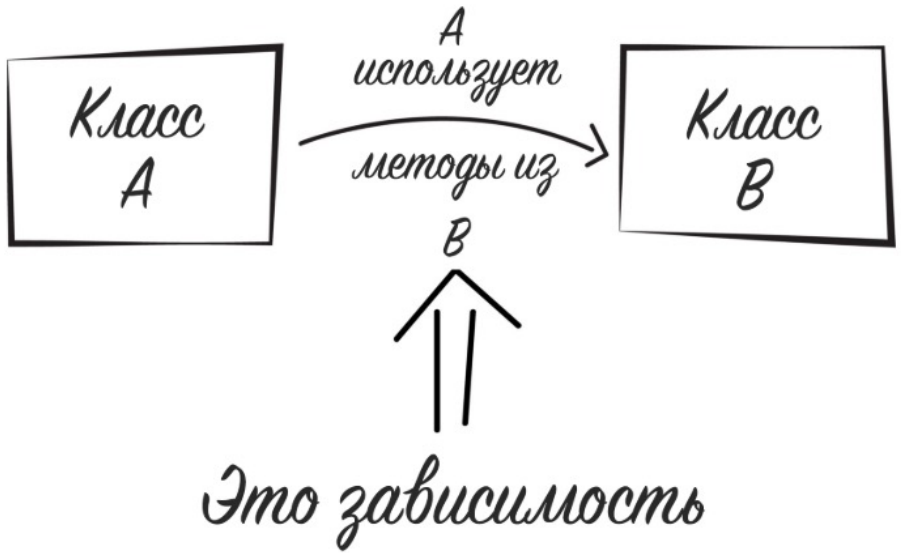
Instrumentation отвечает за поддержку class instrumentation и class loader, которые используются в серверных приложениях.

Модуль ***Messaging*** обеспечивает поддержку STOMP (SimpleText Orientated Messaging Protocol).

И наконец, модуль ***Test*** обеспечивает тестирование с использованием TestNG или JUnit Framework.

В разработке программного обеспечения, внедрение зависимостей это такая техника, где посредством одного объекта (или статического метода) предоставляются зависимости другого объекта. Зависимость — это объект, который может быть использован (как сервис).

Когда класс А использует некоторую функциональность из класса В, тогда говорят, что класс А зависим от класса В.



В Java, прежде чем мы сможем использовать методы других классов, нам необходимо для начала создать экземпляры этого класса (то есть класс A должен создать экземпляр класса B).

Таким образом, передавая задачу создания объекта чему-то другому и прямое использование этой зависимости называется внедрением зависимостей.

```
class Car {
  private Wheels wheel = new MRFWheels();
  private Battery battery = new ExcideBattery();
  ...
  ...
}
```

Перед вами класс Car, отвечающий за создание всех объектов зависимостей. Теперь, что если мы решим избавиться колес компании **MRFWheels** и хотим использовать колеса от **Yokohama** в будущем?

Нам нужно будет воссоздать объект класса Car с новой зависимостью от Yokohama. Но при использовании внедрении зависимостей мы можем изменить колеса во время выполнения программы (потому что зависимости можно внедрять во время выполнения, а не во время компиляции).

Вы можете думать о внедрении зависимостей как о посреднике в нашем коде, который выполняет всю работу по созданию предпочтительного объекта колеса и предоставлению его классу Автомобиль.

Это делает наш класс автомобилей независимым от создания объектов таких как колеса, аккумулятор и т.д.

Существует три основных типа внедрения зависимостей:

constructor injection: все зависимости передаются через конструктор класса.

setter injection: разработчик добавляет setter-метод, с помощью которого инжектор внедряет зависимость

interface injection: зависимость предоставляет инжектору метод, с помощью которого инжектор передаст зависимость. Разработчики должны реализовать интерфейс, предоставляющий setter-метод, который принимает зависимости

Инверсия управления — концепция, лежащая в основе внедрения зависимости

Это означает, что класс не должен конфигурировать свои зависимости статистически, а должен быть сконфигурирован другим классом извне.

Это пятый принцип **S.O.L.I.D** из пяти основных принципов объектно-ориентированного программирования и разработки от **дяди Боба**, в котором говорится, что класс должен зависеть от абстракции, а не от чего-то конкретного (простыми словами, жестко закодированного).

Согласно принципам, класс должен полностью сосредоточиться на выполнении своих обязанностей, а не на создании объектов, необходимых для выполнения этих обязанностей. И именно здесь начинается внедрение зависимостей: она предоставляет классу требуемые объекты.

Аспекто-ориентированное программирование (Aspect oriented programming – AOP)

Это функции, которые охватывают несколько узлов приложения, называются **cross-cutting concerns** и эти **cross-cutting concerns** отделены от непосредственной бизнес-логики приложения.

Примечание: придётся немного потерпеть и почитать пока непонятные термины, но позже, на практике, всё станет существенно яснее.

В ООП ключевой единицей является класс, в то время, как в АОП ключевым элементом является аспект. DI помогает разделить классы приложения на отдельные модули, и АОП – помогает отделить cross-cutting concerns от объектов на которые они влияют.Более подробно АОП будет рассмотрено далее. Крайней вещью, которая будет рассмотрена в этом руководстве будет непосредственно **Архитектура Spring Framework.** На сегодняшний день Spring разделён на некоторое количество отдельных модулей, что позволяет Вам самим решать, какие из них использовать в Вашем приложении.

Контейнеры IoC

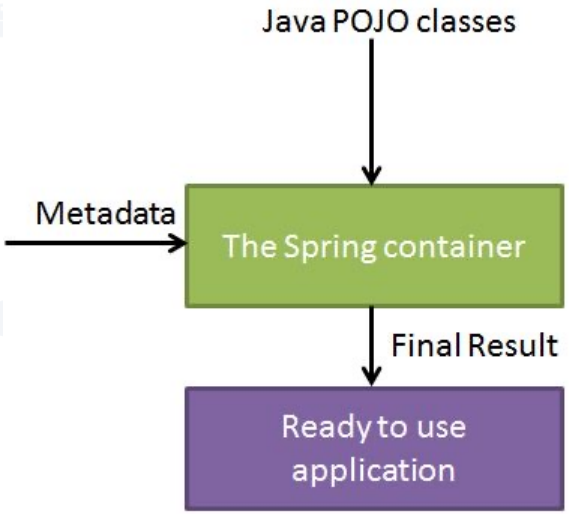
Ключевым элементов Spring Framework является Spring Container. Container создаёт объекты, связывает их вместе, настраивает и управляет ими от создания до момента уничтожения.

Для управления компонентами, из которых состоит приложение, Spring Container использует Внедрение Зависимостей (DI). Эти объекты называются Spring Beans, которые мы обсудим далее.

Spring Container получает инструкции какие объекты инстанциировать и как их конфигурировать через метаданные.

Метаданные могут быть получены 3 способами:

– XML
– Аннотации Java
– Java код



На картинке, которая находится выше схематично показан этот процесс.
Java POJO классы поступает в Spring Container, который на основании инструкций, полученных через метаданные, выдаёт приложение, готовое к использованию.
В Spring имеется 2 различных вида контейнеров:
1. Spring BeanFactory Container;
2. Spring ApplicationContext Container;

Spring ApplicationContext Container

ApplicationContext загружает бины, связывает их вместе и конфигурирует их определённым образом. Но кроме этого, ApplicationContext обладает дополнительной функциональностью: распознание текстовых сообщений из файлов настройки и отображение событий, которые происходят в приложении различными способами. Этот контейнер определяется интерфейсом org.springframework.context.ApplicationContext.

Чаще всего используются следующие реализации AppicationContext:

– **FileSystemXmlApplicationContext**
– **ClassPathXmlApplicationContext**
– **WebXmlApplicationContext**

Bean

Бины – это объекты, которые являются основой приложения и управляются Spring IoC контейнером. Эти объекты создаются с помощью конфигурационных метаданных, которые указываются в контейнере (например, XML-<bean>…</bean>). Я уже говорил о них в предыдущих главах.

Определение бина содержит метаданные конфигурации, которые необходимы управляющему контейнеру для получения следующей информации:

Как создать бин;
Информацию о жизненном цикле бина;
Зависимости бина.

class

Этот атрибут является обязательным и указывает конкретный класс Java-приложения, который будет использоваться для создания бина.

name

Уникальный идентификатор бина. В случае конфигурации с помощью xml-файла, вы можете использовать свойство “id” и/или “name” для идентификации бина.

scope

Это свойство определяет область видимости создаваемых объектов. (Прим. Отсутствие в русском языке достойного перевода этого свойства бинов могут вызвать затруднения, но более подробно оно будет рассмотрено далее).

constructor-arg

Определяет конструктор, использующийся для внедрения зависимости. Более подробно – далее.

properties

Определяет свойства внедрения зависимости. Более подробно рассмотрим далее.

initialization method

Здесь определяется метод инициализации бина

destruction method

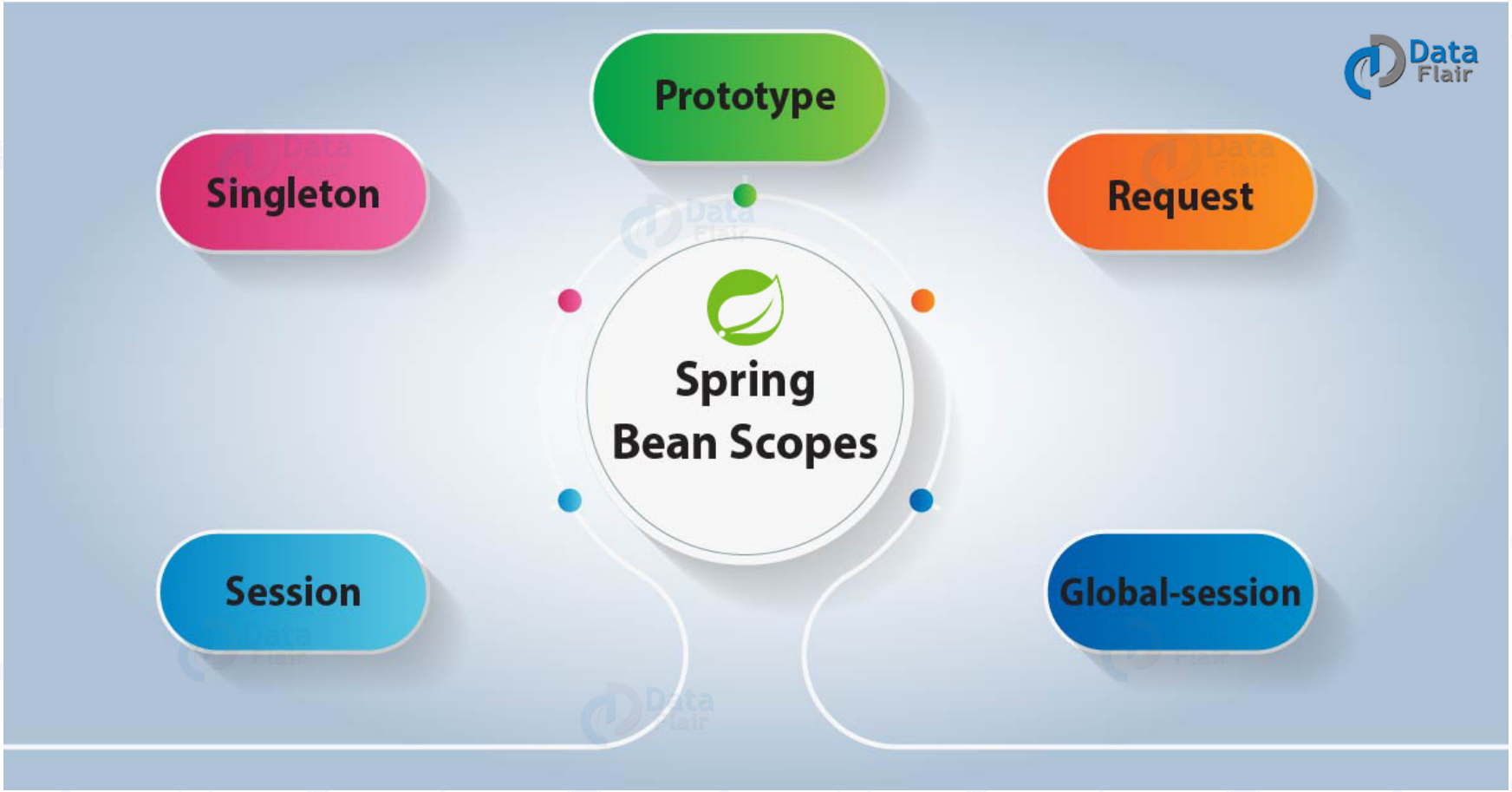
Метод уничтожения бина, который будет использоваться при уничтожении контейнера, содержащего бин.

autowiring mode

Определяет режим автоматического связывания при внедрении зависимости. Более подробно рассмотрим далее.

lazy-initialization mode

Режим ленивой инициализации даёт контейнеру IoC команду создавать экземпляр бина при первом запросе, а не при запуске приложения.

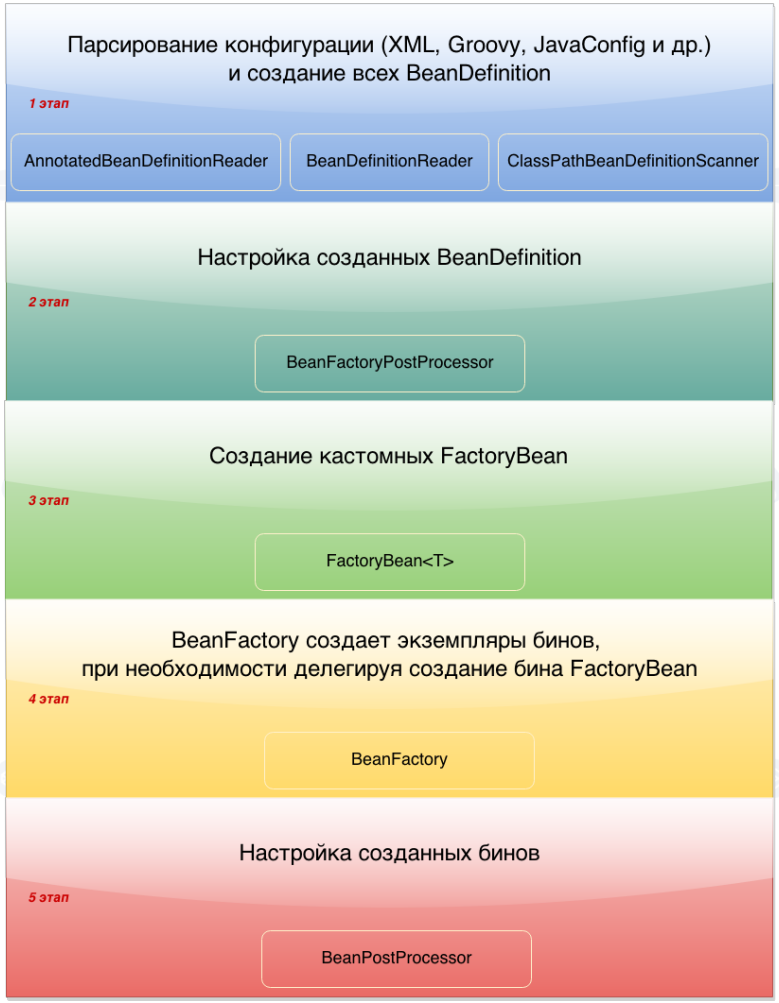

Prototype
Singleton
Request
Spring
Bean Scopes
Session
Global-session
Data
Flair

Несмотря на кажущуюся сложность, жизненный цикл бинов крайне прост и лёгок для понимания. После создания экземпляра бина, могут понадобиться некоторые действия для того, чтобы сделать его работоспособным. Также при удалении бина из контейнера, необходима очистка.

В документации указан список того, что происходит с бином от момента создания до момента уничтожения. Но для начального понимания нам необходимо разобраться в двух наиболее важных методах, которые вызываются во время инициализации бина и его уничтожения.

Для управления созданием и уничтожением бина у нас есть параметры **init-method** и **destroy-method**.

Spring изнутри. Этапы инициализации контекста



1. Парсирование конфигурации и создание BeanDefinition

- **Xml конфигурация** — ClassPathXmlApplicationContext(“context.xml”)
- Конфигурация через аннотации с указанием пакета для сканирования — **AnnotationConfigApplicationContext(“package.name”)**
- Конфигурация через аннотации с указанием класса (или массива классов) помеченного аннотацией @Configuration - **AnnotationConfigApplicationContext(JavaConfig.class).** Этот способ конфигурации называется — JavaConfig.
- **Groovy конфигурация** — GenericGroovyApplicationContext(“context.groovy”)

Цель первого этапа — это создание всех *BeanDefinition*.

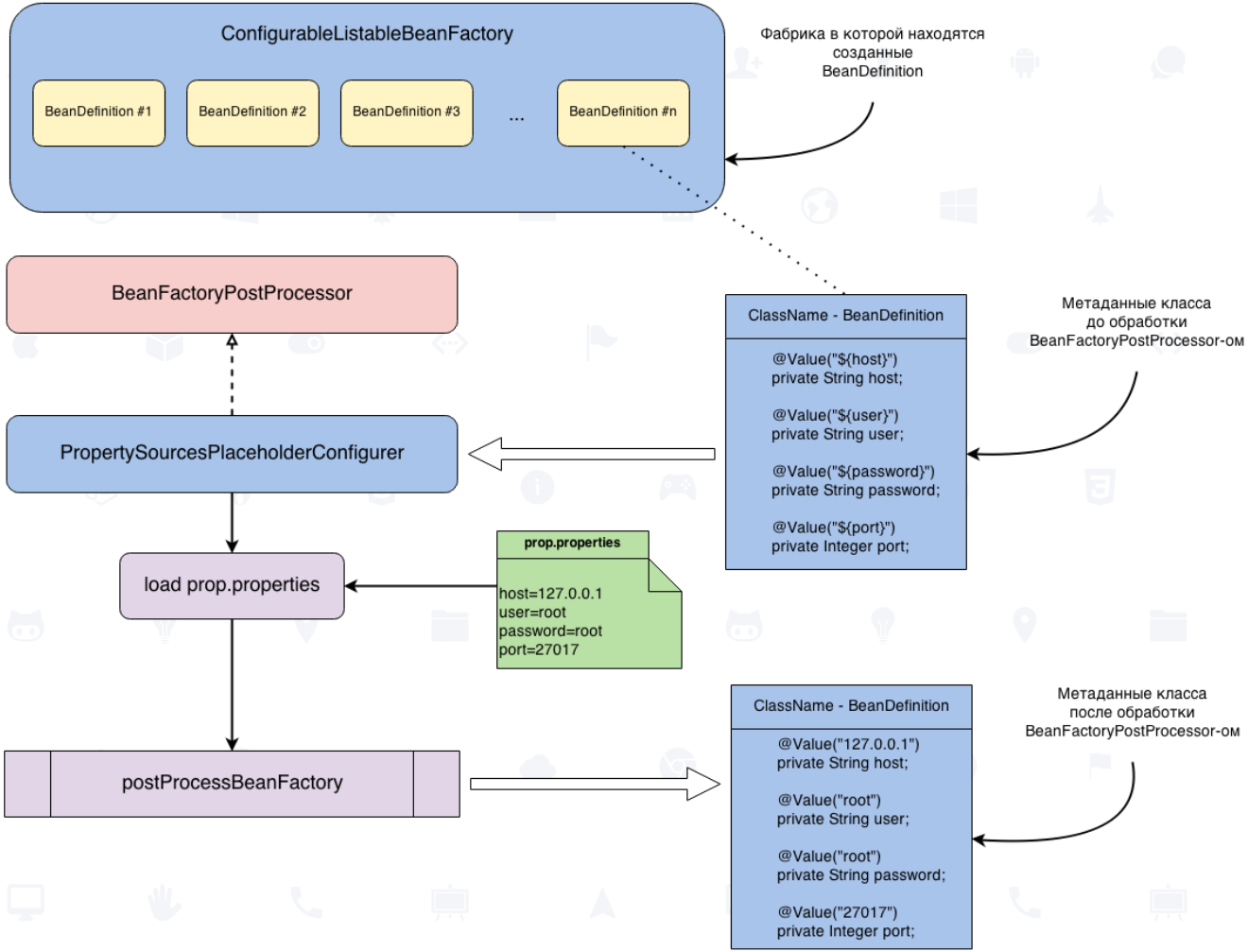
BeanDefinition — это специальный интерфейс, через который можно получить доступ к метаданным будущего бина. В зависимости от того, какая у вас конфигурация, будет использоваться тот или иной механизм парсирования конфигурации.

2. Настройка созданных BeanDefinition

После первого этапа у нас имеется Map, в котором хранятся BeanDefinition. Архитектура спринга построена таким образом, что у нас есть возможность повлиять на то, какими будут наши бины еще до их фактического создания, иначе говоря мы имеем доступ к метаданным класса. Для этого существует специальный интерфейс BeanFactoryPostProcessor, реализовав который, мы получаем доступ к созданным BeanDefinition и можем их изменять

```
public interface BeanFactoryPostProcessor {
        void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException;
}
```

Метод postProcessBeanFactory принимает параметром Configurable ListableBeanFactory. Данная фабрика содержит много полезных методов, в том числе getBeanDefinitionNames, через который мы можем получить все BeanDefinitionNames, а уже потом по конкретному имени получить BeanDefinition для дальнейшей обработки метаданных.

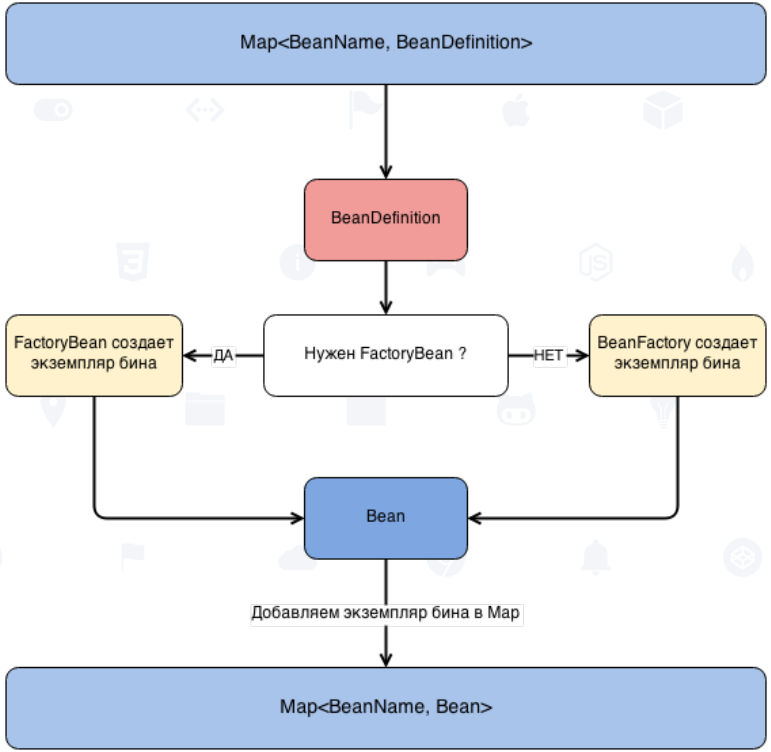

ConfigurableListableBeanFactory
BeanDefinition #1
BeanDefinition #2
BeanDefinition #3
...
BeanDefinition #n
Фабрика в которой находятся созданные BeanDefinition
BeanFactoryPostProcessor
PropertySourcesPlaceholderConfigurer
load prop.properties
postProcessBeanFactory
prop.properties
host=127.0.0.1
user=root
password=root
port=27017
ClassName - BeanDefinition
@Value("${host}")
private String host;
@Value("${user}")
private String user;
@Value("${password}")
private String password;
@Value("${port}")
private Integer port;
Метаданные класса до обработки BeanFactoryPostProcessor-ом
ClassName - BeanDefinition
@Value("127.0.0.1")
private String host;
@Value("root")
private String user;
@Value("root")
private String password;
@Value("27017")
private Integer port;
Метаданные класса после обработки BeanFactoryPostProcessor-ом

3. Создание кастомных FactoryBean

FactoryBean — это generic интерфейс, которому можно делегировать процесс создания бинов типа . В те времена, когда конфигурация была исключительно в xml, разработчикам был необходим механизм с помощью которого они бы могли управлять процессом создания бинов. Именно для этого и был сделан этот интерфейс. Для того что бы лучше понять проблему, приведу пример xml конфигурации.

4. Создание экземпляров бинов

Созданием экземпляров бинов занимается *BeanFactory* при этом, если нужно, делегирует это кастомным *FactoryBean*. Экземпляры бинов создаются на основе ранее созданных *BeanDefinition*.



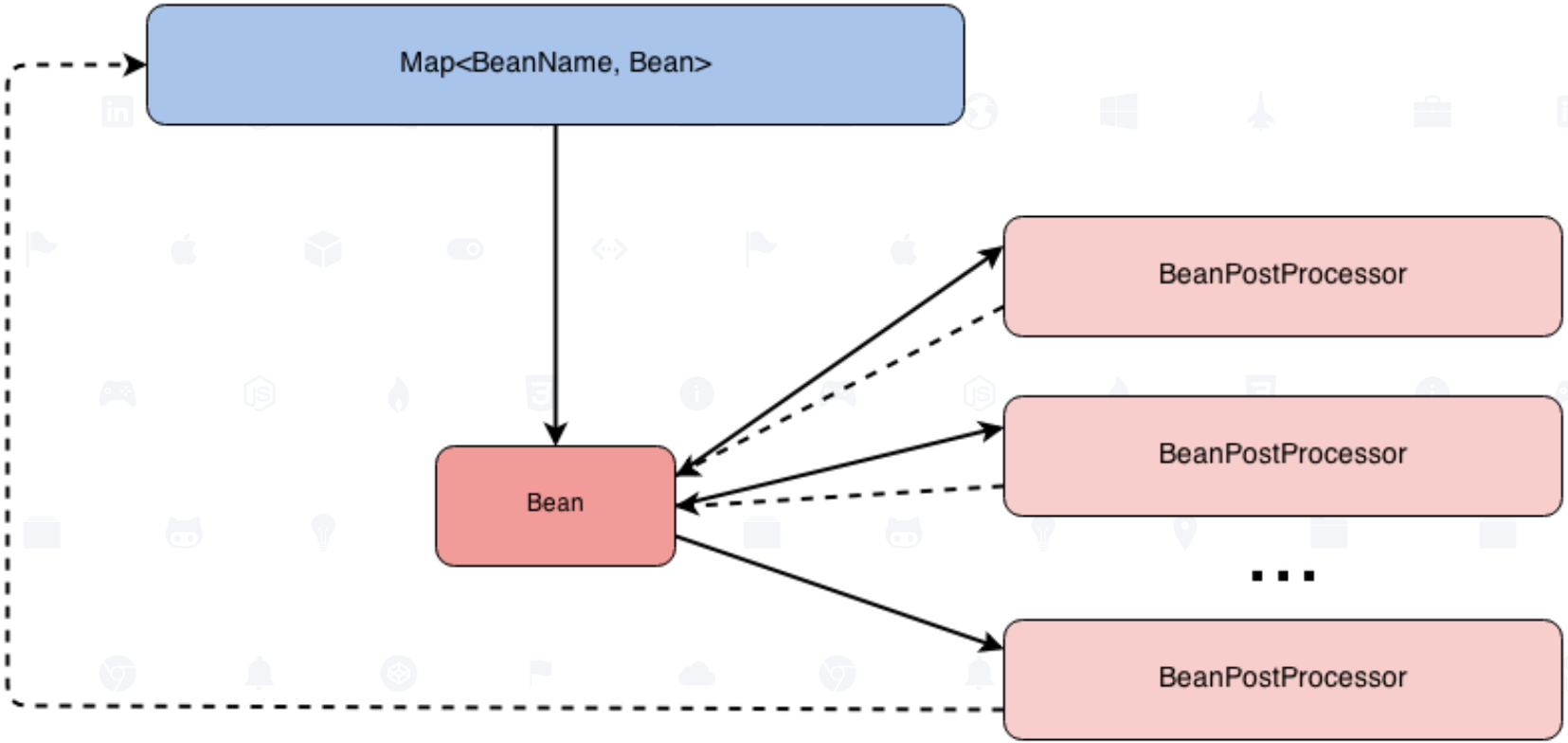
5. Настройка созданных бинов

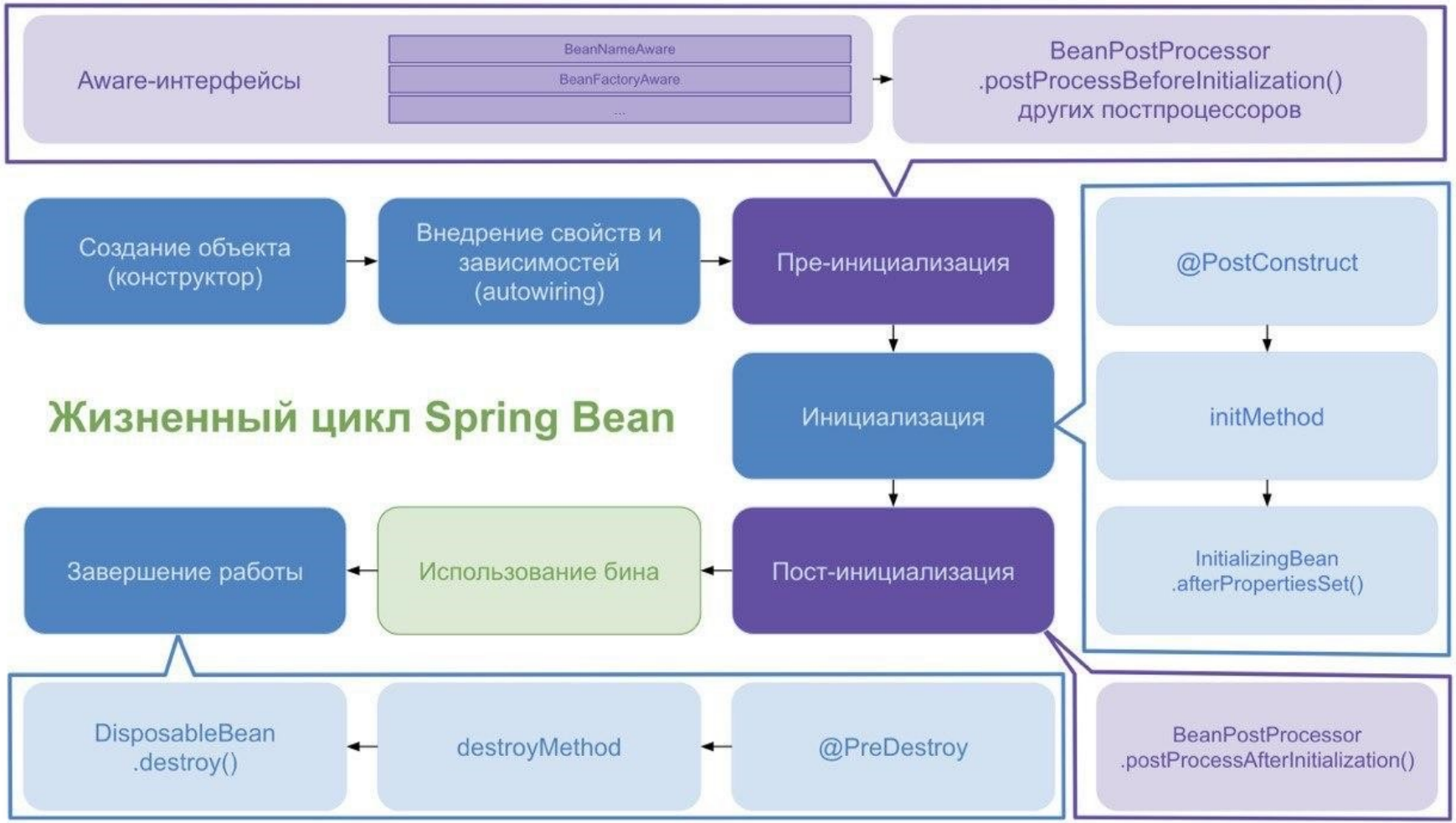
Интерфейс BeanPostProcessor позволяет вклиниться в процесс настройки ваших бинов до того, как они попадут в контейнер. Интерфейс несет в себе несколько методов.

```
public interface BeanPostProcessor {
        Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException;
        Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException;
}
```

Оба метода вызываются для каждого бина. У обоих методов параметры абсолютно одинаковые. Разница только в порядке их вызова. Первый вызывается до init-метода, второй, после. Важно понимать, что на данном этапе экземпляр бина уже создан и идет его донастройка.

Оба метода в итоге должны вернуть бин. Если в методе вы вернете null, то при получении этого бина из контекста вы получите null, а поскольку через бинпостпроцессор проходят все бины, после поднятия контекста, при запросе любого бина вы будете получать фиг, в смысле null.


Map<BeanName, Bean>
BeanPostProcessor
BeanPostProcessor
Bean
...
BeanPostProcessor


Aware-интерфейсы
BeanNameAware
BeanFactoryAware
...
BeanPostProcessor
.postProcessBeforeInitialization()
других постпроцессоров
Создание объекта
(конструктор)
Внедрение свойств и
зависимостей
(autowiring)
Пре-инициализация
@PostConstruct
Жизненный цикл Spring Bean
Инициализация
initMethod
Завершение работы
Использование бина
Пост-инициализация
InitializingBean
.afterPropertiesSet()
DisposableBean
.destroy()
destroyMethod
@PreDestroy
BeanPostProcessor
.postProcessAfterInitialization()

Beans – центральный объект заботы Spring Framework. За кулисами фреймворка с ними происходит множество процессов. Во многие из них можно вмешаться, добавив собственную логику в разные этапы жизненного цикла. Через следующие этапы проходит каждый отдельно взятый бин:

1. Инстанцирование объекта. Техническое начало жизни бина, работа конструктора его класса;

2. Установка свойств из конфигурации бина, внедрение зависимостей;

3. Нотификация aware-интерфейсов. BeanNameAware, BeanFactoryAware и другие.. Технически, выполняется системными подтипами BeanPostProcessor, и совпадает с шагом 4;

4. Пре-инициализация – метод postProcessBeforeInitialization() интерфейса BeanPostProcessor;

5. Инициализация. Разные способы применяются в таком порядке:

- Метод бина с аннотацией @PostConstruct из стандарта JSR-250 (рекомендуемый способ);
- Метод afterPropertiesSet() бина под интерфейсом InitializingBean;
- Init-метод. Для отдельного бина его имя устанавливается в параметре определения initMethod. В xml-конфигурации можно установить для всех бинов сразу, с помощью default-init-method;

6. Пост-инициализация – метод postProcessAfterInitialization() интерфейса BeanPostProcessor.

Когда IoC-контейнер завершает свою работу, мы можем кастомизировать этап штатного уничтожения бина. Как со всеми способами финализации в Java, при жестком выключении (kill -9) гарантии вызова этого этапа нет. Три альтернативных способа «деинициализации» вызываются в том же порядке, что симметричные им методы инициализации:

1. Метод с аннотацией @PreDestroy;
2. Метод с именем, которое указано в свойстве destroyMethod определния бина (или в глобальном default-destroy-method);
3. Метод destroy() интерфейса DisposableBean.

Не следует путать жизненный цикл отдельного бина с жизненным циклом контекста и этапами подготовки фабрик бинов