# 1. Core Technologies

## 1.1 Introduction to the Spring IoC Container and Beans

Dependency Injection это специализированная форма IoC, при которой объекты определяют свои зависимости только через аргументы конструктора, аргументы фабричного метода или свойства, которые установлены в экземпляре объекта после того как он был сконструирован и возвращен от фабричного метода.

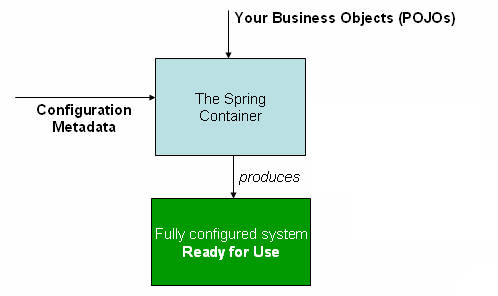
По сути своей этот механизм является обратным, отсюда и название.

BeanFactory интерфейс обеспечивает конфигурационный механизм, способный управлять любым типом объекта. ApplicationContext является наследником BeanFactory. Он добавляет:

* Легкую интеграцию с Spring AOP
* Message resource handling для интернационализации
* Публикации ивентов
* Специфичный контекст на уровне слоя приложения такие как WebApplicationContext для использования в вэб приложениях.

Объекты в Spring, которые формируют скелет приложения и управляются Spring IoC контейнером называются бинами.

## 1.2 Container Overview



### 1.2.1 Configuration metadata

Конфигурационные метаданные могут собираться исходя из разных условий:

1. Из конфигурационного xml файла
2. Из Java аннотаций (на сегодня всё более используемый)
3. Непосредственно из кода Java (@Bean, @DependsOn, @Configuration)

### 1.2.2 Instantiating a Container

Конфигурационные метаданные могут собираться исходя из разных условий:

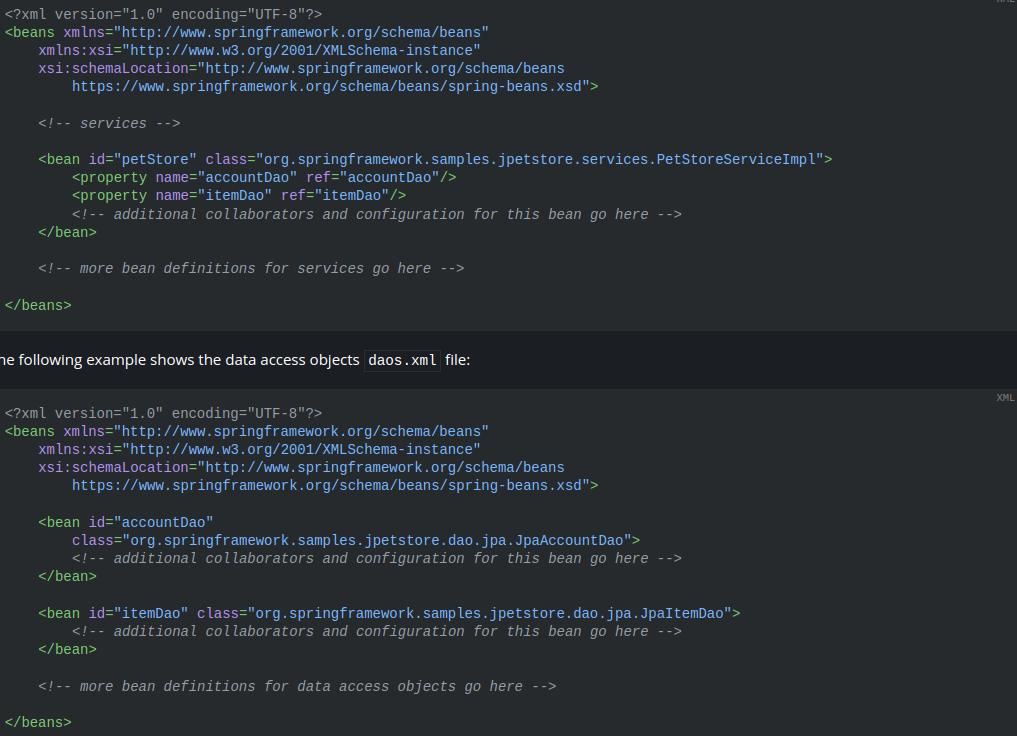
По сути контейнер спринг и его контекст собирается исходя из метаданных, которые могут быть описаны в разных форматах.

Чаще всего используют xml, Java-based форматы. Но возможны несколько других, например Groovy DSL формат, в котором бины и их зависимости описываются языком groovy.

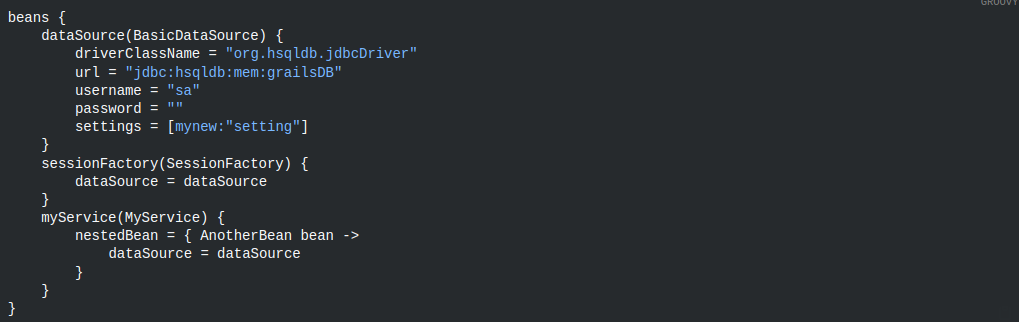
Контекст может быть сформирован только из одного формата данных, либо можно использовать контекст, который имеет возможность загружать метаданные из всех форматов метаданных

XML формат:

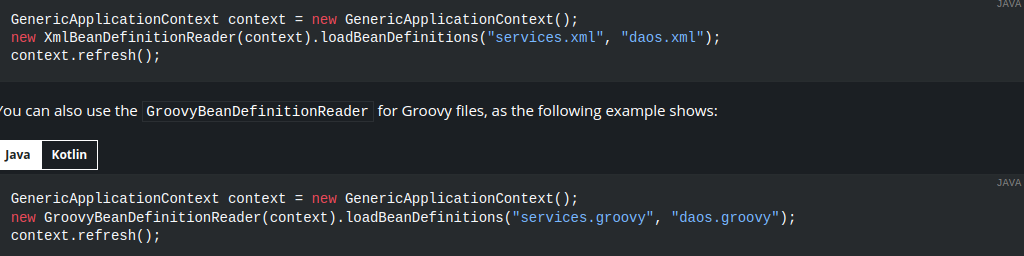




Groovy Bean Definition DSL формат:



 Использование комбинированного контекста:



Можно использовать метод getBean, который присутствует в интерфейсе ApplicationContext. При помощи него можно получить бин из конгтейнера. Но в коде приложения вообще не должно быть использование Spring Api. Напротив, ваши объекты должны работать самостоятельно и использовать зависимости, которые описаны в конфигурационных метаданных для автоматического их использования.

## 1.3 Bean Overview

Спринг IoC контейнер управляет одним или более бинами. Эти бины создаются из конфигурационных метаданных, которые вы поставляете в контейнер, например, через xml конфигурацию.

Внутри самого контейнера эти бины представлены как BeanDefinition объектами, которые содержат (среди прочей информации) следующие метаданные:

1. Наименование класса с его пакетом.
2. Конфигурационные элементы поведения бины, которые определяют как бин должен себя вести в контейнере (область видимости, коллбэки жизненного цикла и т. д.)
3. Ссылки на другие бины, которые нужны, чтобы этот бин работал. Такие бины так же называются соавторами или зависимостями.
4. Другие конфигурационные настройки, которые необходимо установить во вновь созданном объекте.

### 1.3.1 Naming Beans

Каждый бин имеет свой один или более идентификаторов. Эти идентификаторы должны быть уникальны в пределах контейнера, который их содержит. Обычно у бина только один идентификатор.

Бины обычно именуются с маленькой буквы в кэмел кейсе и отражают название класса (applicationService, acceptDao и т. д.).

Можно создавать псевдонимы для бинов, т. е. условно его второе имя для использования в других сервисах. Например, для xml конфигурации можно использовать как:



### 1.3.2 Instantiating Beans

По сути своей bean definition это рецепт для создания одного или более объектов. Контейнер смотрит на рецепт для запрашиваемого бина и использует конфигурационные метаданные инкапсулированные bean definition чтобы создать или получить актуальный объект.

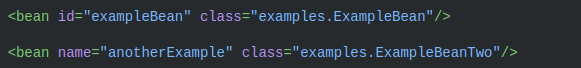
Если использовать xml конфигурацию, обычно определяется класс в атрибуте class <bean/> элемента. Этот атрибут Class (который внутренне является свойством Class в Bean Definition) обычно обязательный. Можно использовать Class по одному из двух способов:

* Обычно чтобы указать класс бина для его создания, в случае, если контейнер сам непосредственно создает bean компонент, рефлексивно вызывая его конструктор, что в некторой степени эквивалентно new оператору.
* Чтобы указать фактический класс, содержащий статический фабричный метод, который вызывается для создания объекта, в менее распространенном случае, когда контейнер вызовет статический фабричный метод класса для создания компонента.

#### 1.3.2.1 Instantiation with a Constructor

Когда вы создаете bean-компонент с помощью конструктора, все обычные классы могут использоваться и совместимы с Spring. То есть разрабатываемый класс не должен реализовывать какие-либо конкретные интерфейсы или быть закодирован определенным образом. Достаточно просто указать класс компонента. Однако в зависимости от того, какой тип IoC вы используете для этого конкретного компонента, вам может потребоваться конструктор по умолчанию (пустой).

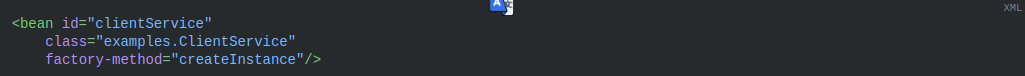
Пример XML:



#### 1.3.2.2 Instantiating with a Static Factory Method

При определении bean-компонента, который вы создаете с помощью статического фабричного метода, используйте атрибут class, чтобы указать класс, который содержит статический фабричный метод, и атрибут с именем Factory-method, чтобы указать имя самого фабричного метода. У вас должна быть возможность вызвать этот метод (с необязательными аргументами, как описано ниже) и вернуть живой объект, который впоследствии обрабатывается так, как если бы он был создан с помощью конструктора. Одним из применений такого определения компонента является вызов статических фабрик в устаревшем коде.

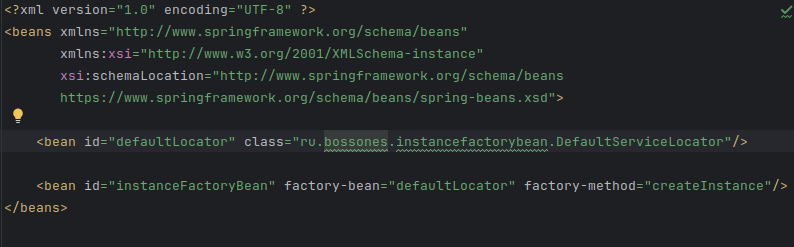
Пример XML:

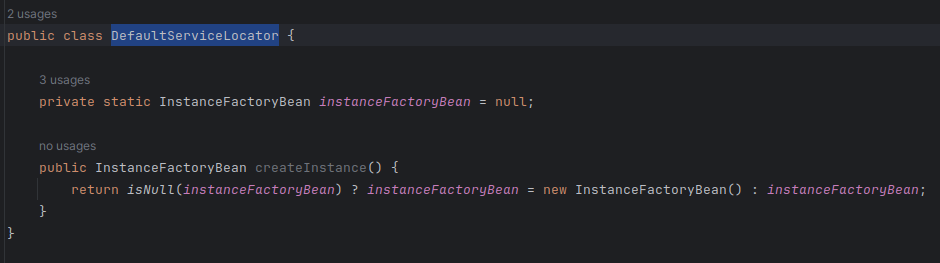


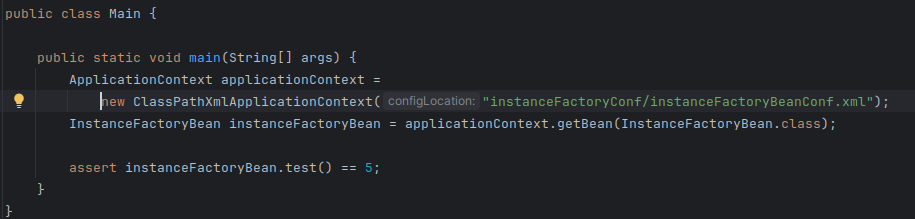
#### 1.3.2.3 Instantiation by Using an Instance Factory Method

Создание экземпляра через данный фабричный метод похож на статический фабричный метод, но здесь применяется нестатический метод экземпляра класса. Чтобы использовать данные механизм, не нужно задавать атрибут class в xml bean definition и в factory-bean атрибуте определяется имя бина в (текущем или родительском) контейнере который содержит инстансный метод который будет вызван чтобы создать объект.

Примеры XML:







## 1.4 Dependencies

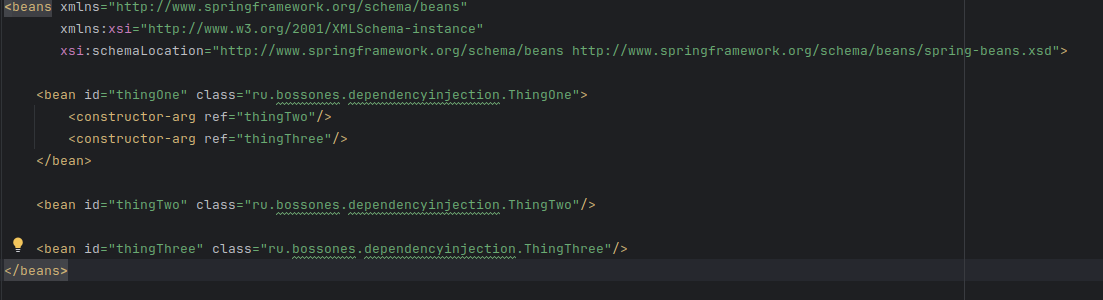
### 1.4.1 Dependency Injection

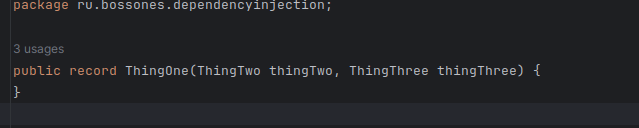
Dependency Injection это процесс, посредством которого объекты определяют их зависимости только через аргументы конструктора, аргументы фабричного метода или свойствами, которые устанаваливаются экземпляру объекта после того как он сконструирован или возвращен из фабричного метода. Затем контейнер внедряет эти зависимости при создании бина. Этот процесс по сути является инверсивным (отсюда и название — Inversion of Control) самому компоненту, управляющему созданием своих зависимостей самостоятельно, используя прямое создание классов или шаблон Service Locator.

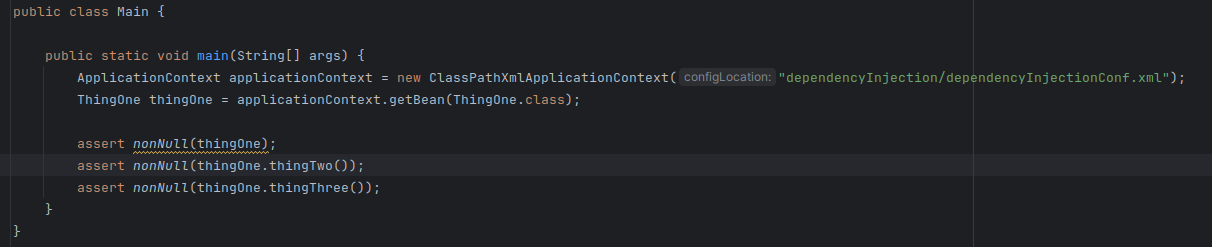
#### 1.4.1.1 Constructor Argument Resolution

Разрешение аргументов конструктора имеет место в использовании типов аргументов. Если нет потенциальной двусмысленности в аргументах конструктора bean definition, порядок аргументов конструктора, который определен в bean definition, является порядком в котором эти аргументы поставляются соответствующему конструктору при создании экземпляра компонента.

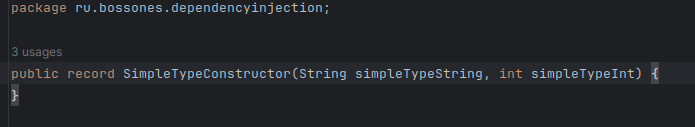
Например в XML Bean Definition:

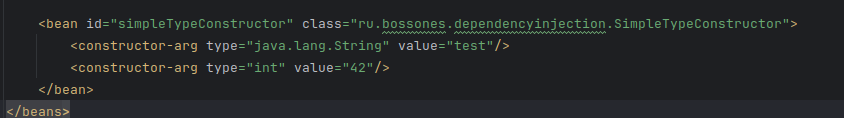




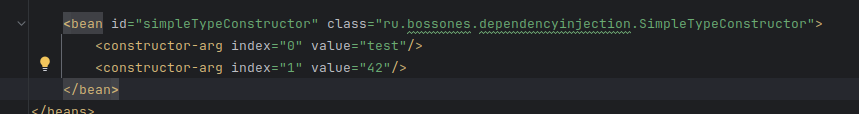


Когда идет ссылка на другой бин, то тип его известен и совпадение может произойти, как в кейсе выше. Но когда применяются примитивные типы в конструкторе, то Spring может не понять какой тип нужно использовать. Поэтому нужно ему помочь в этом.





Так же можно задавать индекс аргументов конструктора в bean definition.

 При использовании одинаковых примитивных типов в конструкторе, использование индекса в bean definition помогает разрешить двусмысленность.

Так же можно использовать имя аргументов в конструкторе, чтобы разрешать двусмысленность.

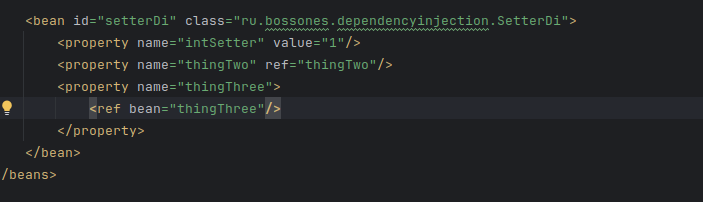
#### 1.4.1.2 Setter-based DI

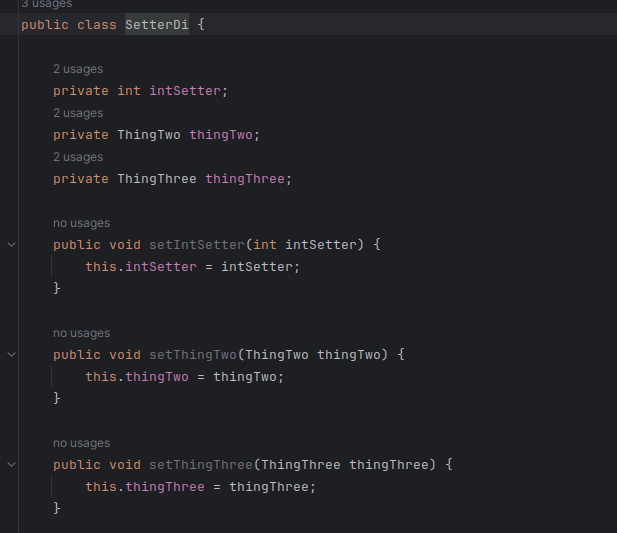
DI базированный на сет методах выполняется контейнером вызывая сеттер методы на бине после выполнения no-args конструктора или no-args статическтого фабричного метода для создания экземпляра вашего бина.

Может использоваться в том случае, если зависимости опицональны, то есть равны null.

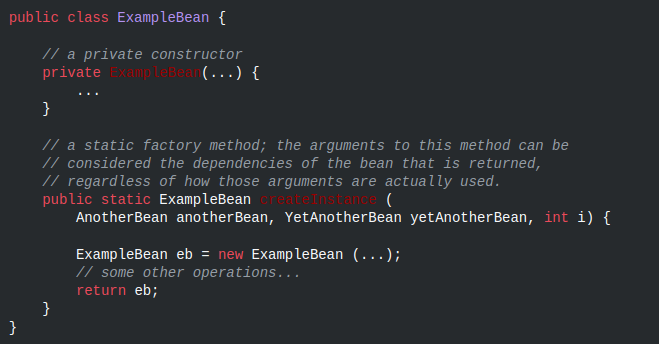
Так же такой метод DI может быть полезен при разрешении циклических зависимостей, т. к. сеттер DI используется уже после того, как бин текущий и бин зависимостей будут созданы. Соответственно зависимость сможет ввнедриться в нужный бин через сеттер метод.

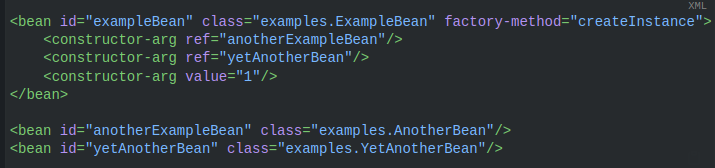
Пример XML:





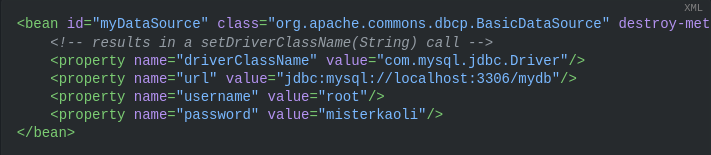
Аналогично Constructor DI можно сделать фабричный метод с несколькими аргументами, которые будут заинжекчены в конструктор в том же порядке:

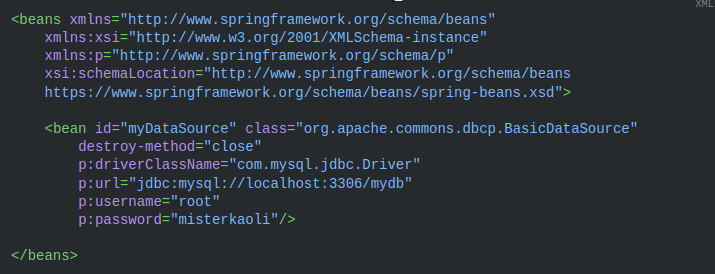




### 1.4.2 Dependencies and Configuration in details.

Для установки значений через сеттеры можно использовать p-namespace, который используется как property элемент. К примеру:





Объявление получается очень кратким, но опечатки обнаруживаются только в рантайме, а не во время разработки. Поэтому лучше использовать IDE.

Так же дополнительно можно конфигурировать java.util.Properties которые всегда можно взять в приложении.



#### 1.4.2.1 Idref элемент.

Данный элемент позволяет очень легко избежать ошибок при передаче идентификатора бина как зависимость для другого через сеттер или через конструктор.

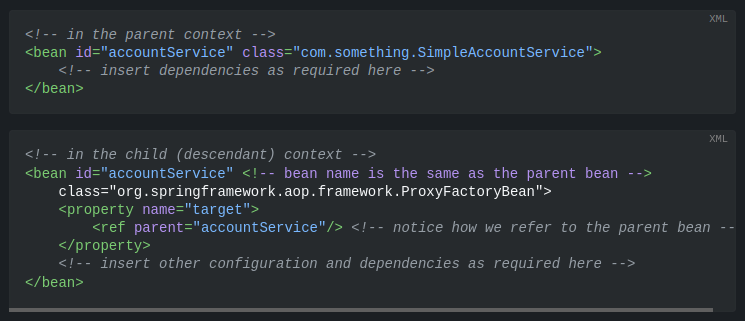
У меня не работет, хз почему(((

#### 1.4.2.2 References to Other Beans (Collaborators)

Общей формой для ссылки на другой бин является использование элемента <ref bean=<<someBean>>> где указывается либо идентификатор бина либо его другое любое наименование в этом контейнере или в любом родительском контейнере.

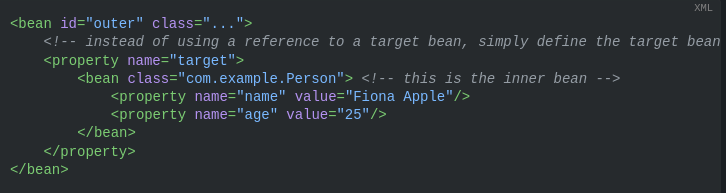
Если определять через parent атрибут, то он создает ссылку на бин, который находится в родительском контейнере текущего контейнера. Значение этого атрибута должно быть идентификатором бина или любым именем этого бина в родительском контейнере текущего контейнера. Необходимо использовать такую ссылку на бин, если у присутствует иерархия контейнеров и необходимо обернуть существующий бин в родительском контейнере с прокси, который имеет такое же имя как родительский бин.

Пример:



#### 1.4.2.3 Внутренние бины

Используя элемент <bean внутри <property/> или <constructor-arg/> элементах можно определить внутренний бин как показано на примере:

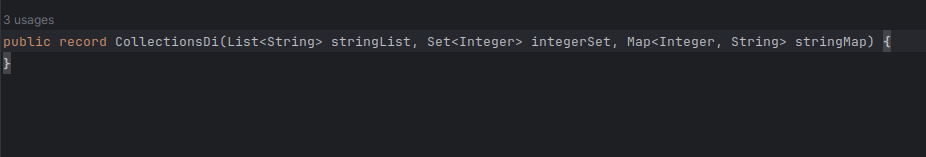


Внутреннее определение бина не требует идентификатора или имени. Если он использован, то контейнер не использует значение как идентификатор. Контейнер так же игнорирует scope при создании, поскольку внутренние бины всегда анонимны и создаются вместе с внешним бином. Нет возможности получить доступ к внутреннему бину независимо от внешнего или инжектировать его в другие бины.

#### 1.4.2.4 Collections

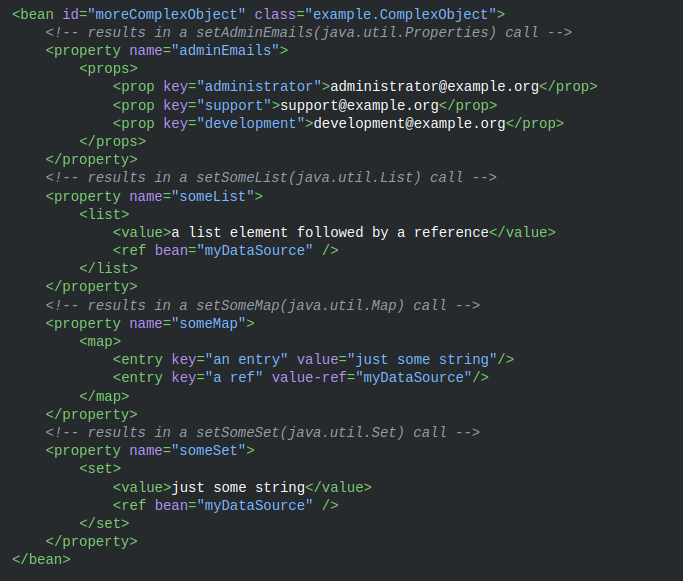
Так же можно задавать элементы коллекций для использования их как аргументы конструктора или как setter DI.

К примеру:



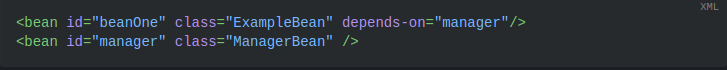


Вместо обычных примитивных значений это могут быть списки бинов или сеты бинов или мапы бинов. Одним словом, практически всё что угодно.

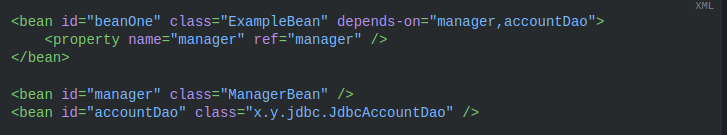


### 1.4.3 Using depends-on

Атрибут depends-on позволяет явно обозначить, что один или более бинов должны быть инициализированы до того, как бин, использующий этот элемент, будет инициализирован.



Для того, чтобы перечислить несколько бинов они перечисляются через запятые, пробелы и точки с запятой:



Атрибут depends-on может задавать как зависимость от времени инициализации, так и, в случае только бинов-одиночек, соответствующую зависимость от времени уничтожения. Зависимые бины, которые определяют depends-on отношения с данным бином, уничтожаются в первую очередь, вплоть до уничтожения самого данного бина. Таким образом, depend-on может также контролировать порядок завершения.

### 1.4.4 Lazy-initialized beans

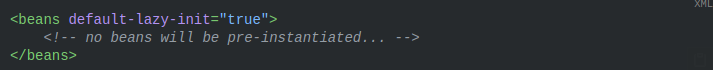
По умолчанию реализации ApplicationContext жадно создают и конфигурируют все синглтон бины как часть процесса инициализации. Как правило, такая предварительная реализация желательна, поскольку ошибки в конфигурации или окружающей среде обнаруживаются немедленно, а не часами или днями позже. Если же поведение не определено, можно самостоятельно предотвращать пре-инициализацию синглтон-бинов, если указать, что они lazy-initialized. Это свойство бина говорит контейнеру, чтобы создать экземпляр бина когда он впервые потребуется, а не при запуске.

В xml bean definition это свойство регулируется атрибутом lazy-init внутри <bean/> элемента:

 В примере выше бин lazy предварительно не создается, когда запускается ApplicationContext, в то время как бин not.lazy создается предварительно без всяких задержек.

Однако, если бин с отложенной инициализацией является зависимостью синглтон-бина, который не является lazy-бином, то ApplicationContext создает lazy-бин при запуске, поскольку он должен удовлетворять зависимостям бина-одиночки. Lazy-бин внедряется в синглтон-бин в другом месте, где неот отложенной инициализации.

Можно так же управлять lazy-инициализацией на уровне контейнера, используя атрибут default-lazy-init в элементе <beans/>



### 1.4.5 Autowiring Collaborators

Spring контейнер может автоматически связывать отношения между взаимодействующими бинами. Можно дать спрингу автоматически разрешить установление связни для бинов посредством инспектирования контента в ApplicationContext. Автосвязывание имеет следующие преимущества:

1. Существенно может уменьшать необходимость определять свойства (properties) или аргументы конструктора.

2. Автосвязывание может обновлять конфигурации по мере развития объектов. Например, если необходимо добавить зависимость в класс, то эта зависимость может быть установлена автоматически без необходимости модифицироовать конфигурацию. Таки образом, автосвязывание может быть особенно полезно при разработке, не исключая возможности переключения на явное связывание, когда база кода станет более стабильной.

Для xml bean definition можно применить атрибут autowire в <bean/> элементе. Имеется 4 режима автосвязывания.

|  |  |
| --- | --- |
| Режим | Разъяснения |
| no | Дефолт. Без автосвязывания. Ссылки на бины должны быть определены через ref элементы. Изменение дефолтных настроек не рекомендуется для больших приложений, т. к. определение связности зависимостей явно даёт больший контроль и четкость. |
| byName | Автосвязывание по имени свойства. Спринг будет смотреть на бин с таким же именем как у свойства, которое должно быть связано. Например, если у bean definition установлена автосвязность по имени и содержит свойство master (так, что у него есть сеттер setMaster()), спринг найдет bean definition с именем master и использует его для установления свойства. |
| byType | Позволяет свойству быть связанным, если один и только один бин с типом как у этого свойства существует в контейнере. Если существует более одного бина, будет выброшено исключение, которое говорит, что нельзя использовать автосвязывание по типу для этого бина. Если нет ни одного подходящего бина, то свойство не устанавливается. |
| constructor | Аналогично режиму byType но применяется для аргументов конструктора. Если нет ни одного подходящего бина, выбрасывается ошибка. |

Ограничения и недостатки автосвязывания:

1. Явно установленные зависимости в property и constructor-arg всегда переопределяют автосвязывание.

2. Автосвязывание менее понимаемо, в отличии от явного связывания. Хотя спринг с осторожностью пытается избежать угадывание в случае двусмысленности, которое может привести к непредвиденным результатам. Отношения между управляемыми объектами Спринга больше не документируется явно.

3. Информация о подключении может быть не доступна для инструментов, которые могут генерировать документацию из контейнера спринга.

Исключение бина из автосвязывания.

Можно попробовать исключать бины для автосвязывания. Для xml фоормата устанавливается атрибут autowire-candidate=false для элемента <bean/>. Контейнер сделает этот особенный bean definition не доступным для автосвязывания (включая автосвязывание через аннотацию @Autowiring).

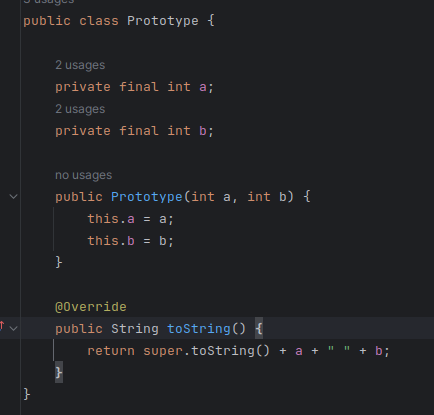
### 1.4.6 Method injection

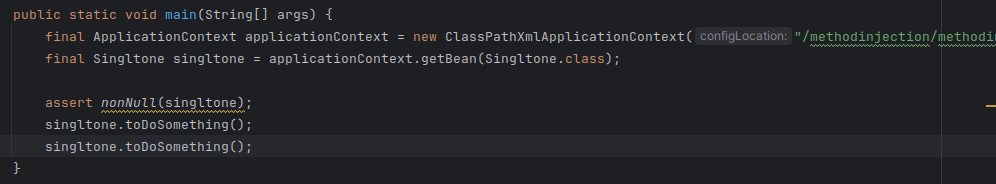
В большинстве сценариев подавляющее количество бинов являются синглтонами. Когда синглотн бин взаимодействет с другим синглтон бином или не синглтон бин взаимодействет с другим не синглтон бином, вы обычно управляете зависимостями путем определения одного бина как свойство другого бина. Проблемы возникают, когда жизненные циклы бинов разные. Прдположим, что синглтон бин А требует использовать не синглтон (prototype) бин Б, возможно в каждом вызове метода в бине А. Контейнер создаст синглтон бин А только один раз и, таким образом, получит воззможность установить свойства. Контейнер не может дать бину А новый экземпляр бина Б каждый раз, когда он нужен.

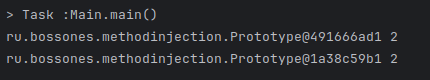
Решение заключается в том, чтобы отказаться от инверсии управления. Можно сделать бин осведомленным о контейнере благодаря реализации ApplicationContextAware интерфйеса, вызывая создание бина Б через контейнер каждый раз, когда он необходим.

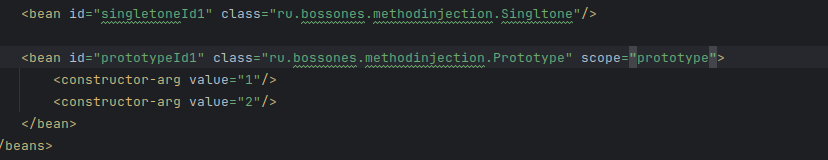
Пример:







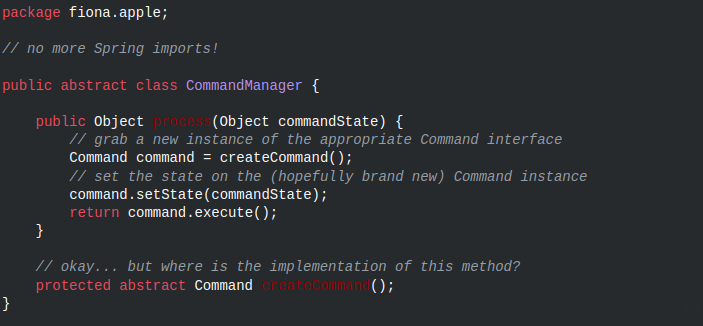




#### 1.4.6.1 Lookup Method Injection

Внедрение зависимости через метод поиска - это способность контейнера переопределять методы управляемых контейнером бинов и возвращать результат поиска для другого именованного бина в контейнере. Поиск обычно предусматривает бин-прототип, как в сценарии, описанном в предыдущем разделе. Фреймворк Spring реализует это внедрение зависимости через метод, используя генерацию байт-кода из библиотеки CGLIB для динамической генерации подкласса, который переопределяет метод.

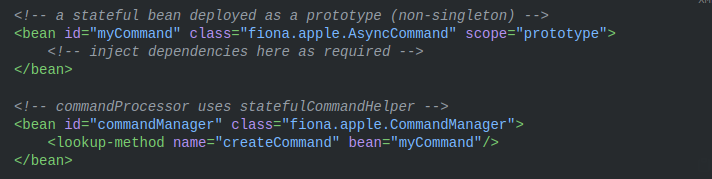
* Чтобы это динамическое построение подклассов работало, класс, который контейнер бина Spring разделяет на подклассы, не может быть final, и переопределяемый метод тоже не может быть final.
* Модульное тестирование (юнит-тестирование) класса, в котором присутствует abstract метод, требует, чтобы вы самостоятельно создали подкласс класса и обеспечили реализацию функции-заглушки abstract метода.
* Также необходимы конкретные методы для сканирования компонентов, для чего требуются конкретные классы.
* Еще одним ключевым ограничением является то, что методы поиска не работают с фабричными методами и, в частности, с методами @Bean в конфигурационных классах, поскольку в этом случае контейнер не отвечает за создание экземпляра и поэтому не может создать генерируемый во время выполнения подкласс на лету.



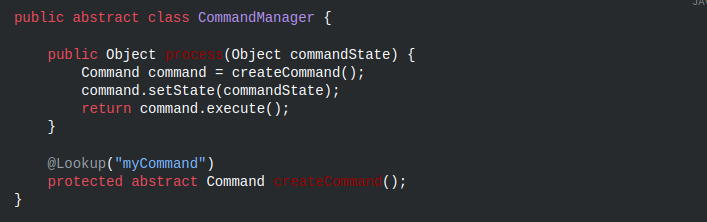
Метод поиска должен соответствовать следующей сигнатуре:



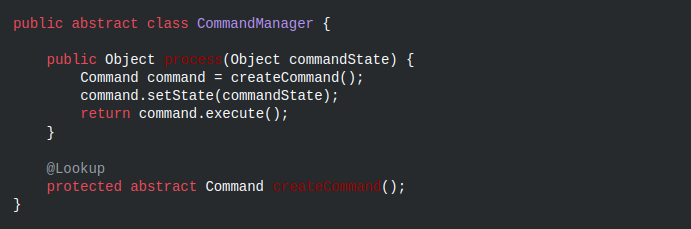
В xml bean definition будет так:



Можно также использовать аннотации Java с указанием конкретного бина:



А можно и без указания, положившись на тип бина, который должен быть подставлен в возвращаемый результат метода.



## 1.5 Bean Scopes