

Введение в Spring (часть 2)



ПРЕПОДАВАТЕЛЬ



Юрий Костяной

Java/Kotlin backend-разработчик

- 3+ года опыта в коммерческой разработке
- 2+ года опыта в преподавании
- Проекты по интеграции сторонних платформ, CRM
- Проблемно-ориентированный подход в преподавании



ВАЖНО:

- Камера должна быть включена на протяжении всего занятия.
- Если у Вас возник вопрос в процессе занятия, пожалуйста, поднимите руку и дождитесь, пока преподаватель закончит мысль и спросит Вас, также можно задать вопрос в чате или когда преподаватель скажет, что начался блок вопросов.
- Организационные вопросы по обучению решаются с кураторами, а не на тематических занятиях.
- Вести себя уважительно и этично по отношению к остальным участникам занятия.
- Во время занятия будут интерактивные задания, будьте готовы включить камеру или демонстрацию экрана по просьбе преподавателя.

ПЛАН ЗАНЯТИЯ

1. Повторение
2. Основной блок
3. Вопросы по основному блоку
4. Домашняя работа



TEL-RAN
by Starta Institute

1

ПОВТОРЕНИЕ ИЗУЧЕННОГО

Повторение

Класс Demention хранит данные измерений габаритов груза. Класс Weight хранит данные массы груза. Класс Load хранит всю информацию о грузе.

Класс Scales получает id груза и возвращает Weight. Класс TapeMeasure получает id груза и возвращает Demention. Класс LoadService получает входные данные о грузе, присваивает грузу id, с помощью Scales и TapeMeasure получает данные о массе и габаритах груза, затем возвращает объект Load.

Какие из указанных классов нужно сделать бинами?



Повторение

Исправьте ошибку в коде

```
<bean id="firstBean" class="org.example.pojo.FirstBean"/>
```

```
public class FirstBean {  
    private String name;  
  
    public FirstBean(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

Повторение

Исправьте ошибку в коде

```
<bean id="firstBean" class="org.example.pojo.FirstBean">  
  <constructor-arg value="Mr. First"/>  
</bean>
```

В конструктор класса *FirstBean* необходимо передать строку. Этот аргумент должен быть указан в конфигурации контекста.

```
public class FirstBean {  
    private String name;  
  
    public FirstBean(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```


Повторение

Исправьте ошибку в коде

```
<bean id="secondBean" class="org.example.pojo.SecondBean">  
  <constructor-arg value="Mr. Second"/>  
  <constructor-arg ref="businessLogic"/>  
</bean>
```

```
public class SecondBean {  
  private String name;  
  
  private final BusinessLogic businessLogic;  
  
  public SecondBean(String name, BusinessLogic businessLogic) {  
    this.name = name;  
    this.businessLogic = businessLogic;  
  }  
}
```

Повторение

Исправьте ошибку в коде

```
<bean id="secondBean" class="org.example.pojo.SecondBean">  
  <constructor-arg value="Mr. Second"/>  
  <constructor-arg ref="businessLogic"/>  
</bean>
```

```
public class SecondBean {  
  private String name;  
  
  private final BusinessLogic businessLogic;  
  
  public SecondBean(String name, BusinessLogic businessLogic) {  
    this.name = name;  
    this.businessLogic = businessLogic;  
  }  
}
```

Нет описания бина *businessLogic*.

Повторение

Исправьте ошибку в коде

```
<bean id="businessLogic" class="org.example.pojo.BusinessLogicImpl"/>
<bean id="secondBean" class="org.example.pojo.SecondBean">
  <constructor-arg value="Mr. Second"/>
  <constructor-arg ref="businessLogic"/>
</bean>
```

```
public static void main(String[] args) {
    ClassPathXmlApplicationContext context =
        new ClassPathXmlApplicationContext("applicationContext.xml");
    BusinessLogicImpl logic = context.getBean("businessLogic", BusinessLogicImpl.class);
    context.close();

    SecondBean sb = new SecondBean("Mr. Second", new BusinessLogicImpl());
    System.out.println(sb.getName());
    sb.getBusinessLogic().doBusinessLogic();
}
```

Повторение

Исправьте ошибку в коде

```
<bean id="businessLogic" class="org.example.pojo.BusinessLogicImpl"/>
<bean id="secondBean" class="org.example.pojo.SecondBean">
  <constructor-arg value="Mr. Second"/>
  <constructor-arg ref="businessLogic"/>
</bean>
```

```
public static void main(String[] args) {
    ClassPathXmlApplicationContext context =
        new ClassPathXmlApplicationContext("applicationContext.xml");
    SecondBean sb = context.getBean("businessLogic", SecondBean.class);
    System.out.println(sb.getName());
    sb.getBusinessLogic().doBusinessLogic();
    context.close();
}
```

Нет необходимости создавать SecondBean и инжектировать в него BusinessLogic, т.к. Spring делает это самостоятельно.

Повторение

Исправьте ошибку в коде

```
<bean id="recipeSteps" class="org.example.cooking.RecipeSteps" scope="singleton"/>
<bean id="tastyDish" class="org.example.cooking.TastyDish" scope="prototype">
  <constructor-arg ref="recipeSteps"/>
</bean>
```

```
package org.example.cooking;
```

```
public class RecipeSteps {
    private final Queue<String> steps;
    public RecipeSteps() {
        steps = new ArrayDeque<>(List.of("Wash", "Dry", "Cut", "Fry", "Serve"));
    }
    public String getNextStep() {
        return steps.poll();
    }
}
```

```
package org.example.cooking;
```

```
public class TastyDish {
    public TastyDish(RecipeSteps steps) {
        String step = steps.getNextStep();
        while (step != null) {
            System.out.println(step);
            step = steps.getNextStep();
        }
        System.out.println("Dish is ready\n");
    }
}
```

```
public static void main(String[] args) {
    ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
    TastyDish dish1 = context.getBean("tastyDish", TastyDish.class);
    TastyDish dish2 = context.getBean("tastyDish", TastyDish.class);
    context.close();
}
```

Повторение

Исправьте ошибку в коде

Бин recipeSteps в каждом tastyDish должен быть своим, поэтому его scope должен быть prototype



```
<bean id="recipeSteps" class="org.example.cooking.RecipeSteps" scope="prototype"/>
<bean id="tastyDish" class="org.example.cooking.TastyDish" scope="prototype">
  <constructor-arg ref="recipeSteps"/>
</bean>
```

```
package org.example.cooking;
```

```
public class RecipeSteps {
    private final Queue<String> steps;
    public RecipeSteps() {
        steps = new ArrayDeque<>(List.of("Wash", "Dry", "Cut", "Fry", "Serve"));
    }
    public String getNextStep() {
        return steps.poll();
    }
}
```

```
package org.example.cooking;
```

```
public class TastyDish {
    public TastyDish(RecipeSteps steps) {
        String step = steps.getNextStep();
        while (step != null) {
            System.out.println(step);
            step = steps.getNextStep();
        }
        System.out.println("Dish is ready\n");
    }
}
```

```
public static void main(String[] args) {
    ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
    TastyDish dish1 = context.getBean("tastyDish", TastyDish.class);
    TastyDish dish2 = context.getBean("tastyDish", TastyDish.class);
    context.close();
}
```

Повторение

В чём прикол мема?



2

ОСНОВНОЙ БЛОК

Введение

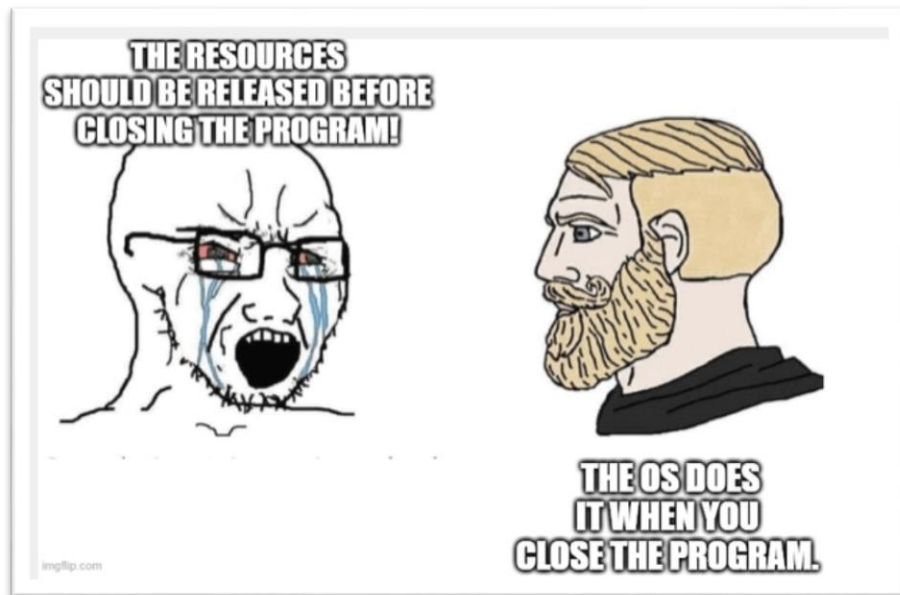
- Невероятная жизнь
- Фабричный значит качественный
- Тонкая настройка
- Сложный выбор



Проблема

Если объект какого-то класса должен работать с потоками ввода-вывод (файлом, сетью, базой данных и т.д.), то в обычной программе мы контролируем его работу, чтобы занимать и освобождать такие ресурсы. Но при работе со *Spring* мы не контролируем процесс создания и уничтожения бинов.

Как выполнить работу с ресурсами в Spring?



Невероятная жизнь Жизненный цикл бина

Для того, чтобы выполнять некоторые действия до создания бина и перед его полным уничтожением (после закрытия контекста) было введено понятие **жизненный цикл бина**.

Coffee beans lifecycle



Невероятная жизнь

Жизненный цикл бина

Жизненный цикл бина в Spring-приложении можно описать следующей схемой





Невероятная жизнь Жизненный цикл бина

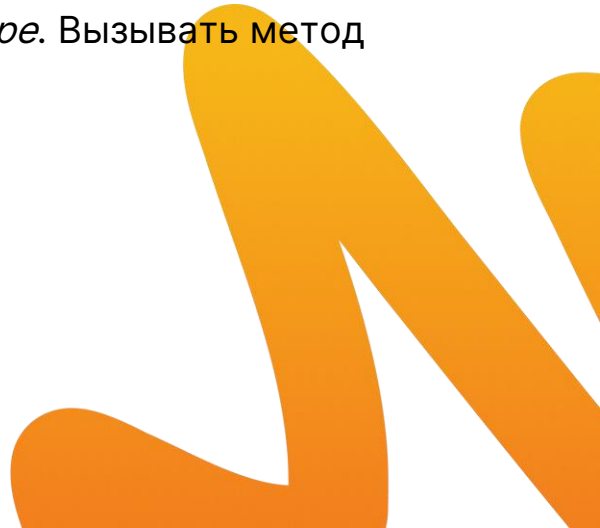
Жизненный цикл бина в Spring-приложении можно описать следующей схемой



Init и destroy методы

- могут иметь любой модификатор доступа;
- тип возвращаемого значения может быть любой, но обычно используется void, т.к. не возможности получить возвращаемое значение;
- название может быть любым;

-  методы не должны принимать на вход какие-либо аргументы;
-  Spring не вызывает destroy-метод у бинов со *scope = prototype*. Вызывать метод должен сам разработчик.



Невероятная жизнь

Пример жизненного цикла бина



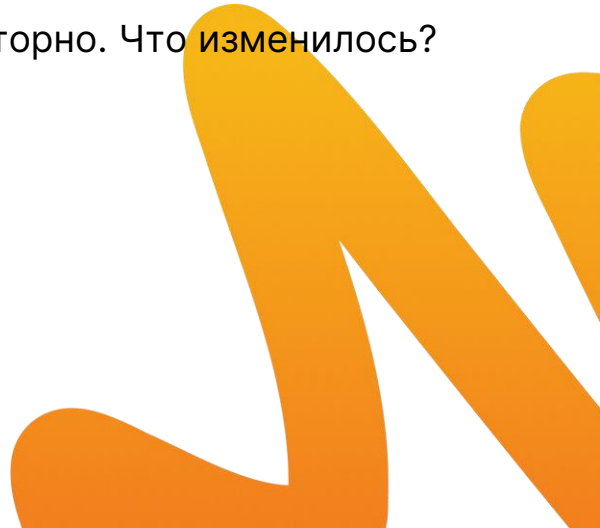
SpringXmlBeanLifecycleExample.zip



Задание

Создайте Spring-приложение, которое использует бины класса ImmediateThread. Перед созданием бина ImmediateThread создаёт поток из бина типа Runnable. На стадии создания бина ImmediateThread запускает поток. На стадии уничтожения бина поток помечается для прерывания. Сопроводите все этапы соответствующими сообщениями в консоль.

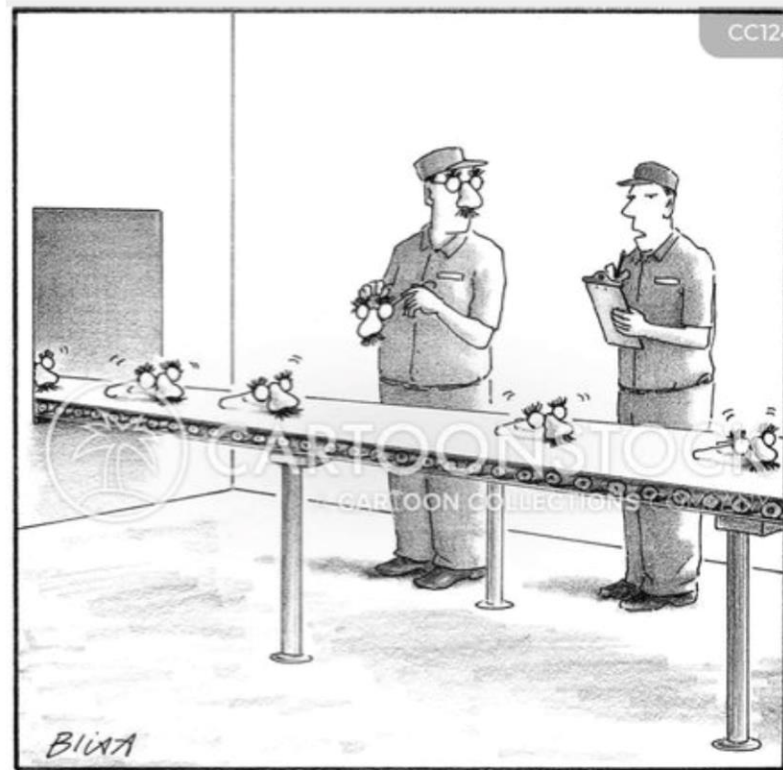
Создайте бин ImmediateThread с областью видимости singleton и выполните программу. Затем измените scope на prototype и выполните программу повторно. Что изменилось?



Проблема

Многие классы по разным причинам переносят функционал создания объектов из конструкторов в отдельный метод. Например, объект *java.time.LocalDate* можно создать только с помощью метода *LocalDate.of()*. Аналогично *java.nio.charset.Charset* имеет метод *forName()*. Все *singleton*-классы используют *getInstance()* для получения экземпляра.

Как Spring справляется с такими методами?



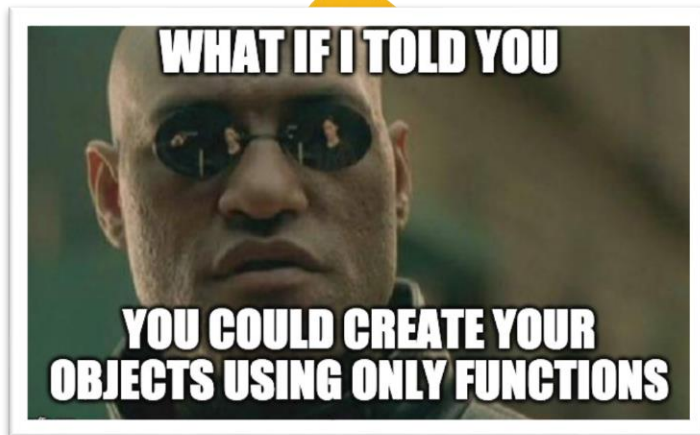
"Yeah, that one's funny, too. Next."

Фабричный значит качественный

Паттерн Фабричный метод

Фабричный метод – паттерн проектирования, который переносит создание объекта из конструктора в отдельный статический метод класса или суперкласса. Это позволяет решить несколько задач:

1. Лучше контролировать процесс создания задачи и при необходимости возвращать не новую сущность, а ранее созданную или пустое значение (null, Optional.empty()).
2. При определении порождающего метода в суперклассе наследники могут переопределять тип возвращаемого значения.



Фабричный значит качественный

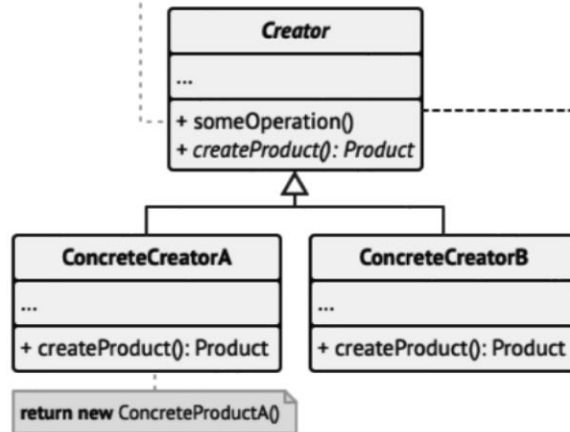
Паттерн Фабричный метод

3 Создатель объявляет фабричный метод, который должен возвращать новые объекты продуктов. Важно, чтобы тип результата совпадал с общим интерфейсом продуктов.

Зачастую фабричный метод объявляют абстрактным, чтобы заставить все подклассы реализовать его по-своему. Но он может возвращать и некий стандартный продукт.

Несмотря на название, важно понимать, что создание продуктов **не является** единственной функцией создателя. Обычно он содержит и другой полезный код работы с продуктом. Аналогия: большая софтверная компания может иметь центр подготовки программистов, но основная задача компании — создавать программные продукты, а не готовить программистов.

```
Product p = createProduct()  
p.doStuff()
```



4 Конкретные создатели по-своему реализуют фабричный метод, производя те или иные конкретные продукты.

Фабричный метод не обязан всё время создавать новые объекты. Его можно переписать так, чтобы возвращать существующие объекты из какого-то хранилища или кэша.

1 Продукт определяет общий интерфейс объектов, которые может произвести создатель и его подклассы.

2 Конкретные продукты содержат код различных продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий.

Фабричный значит качественный

Фабричный метод в Spring

Для использования фабричного метода в Spring достаточно в конфигурации у бина указать имя метода с тегом *factory-method*.

```
<bean  
    id="localDateNow"  
    class="java.time.LocalDate"  
    factory-method="now"  
>
```

```
public static void main(String[] args) {  
    ClassPathXmlApplicationContext context =  
        new ClassPathXmlApplicationContext("applicationContext.xml");  
  
    LocalDate logger = context.getBean("localDateNow", LocalDate.class);  
    System.out.println(logger); // выведет сегодняшнюю дату  
    context.close();  
}
```

Задание

Создайте класс Plant с фабричным методом produceNewPlant. Создайте бин Plant.

Проверьте, позволяет ли использование фабричного метода получать новые экземпляры бинов, если не указан scope singleton.



Проблема

Конфигурировать бины через xml-файл не очень удобно, потому что xml многословный, да и прописывать каждый бин в конфигурации довольно утомительно.

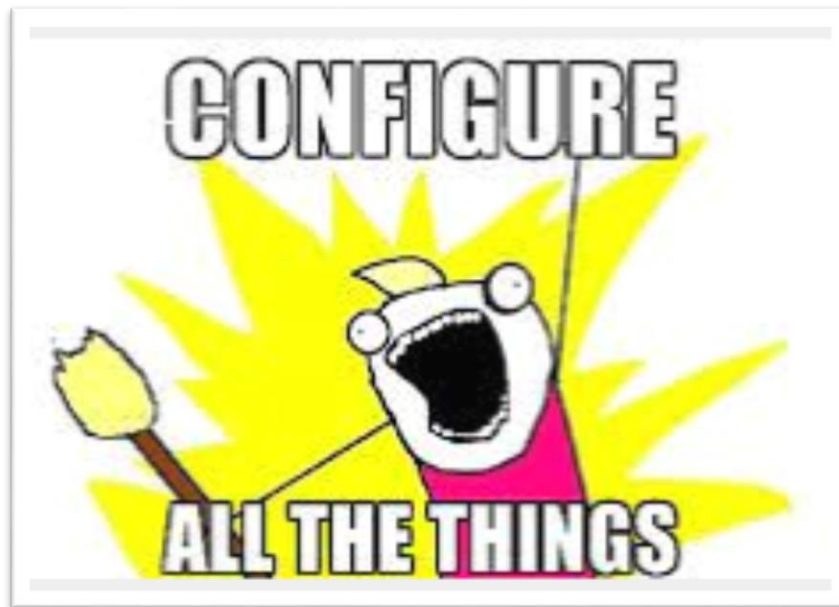
Может ли Spring и это автоматизировать?



Тонкая настройка

Конфигурация Spring

- С помощью xml-файла (старый способ, встречается в старых проектах);
- С помощью аннотаций и немного xml (современный, популярный подход)
- С помощью Java-класса (современный, менее популярный подход)

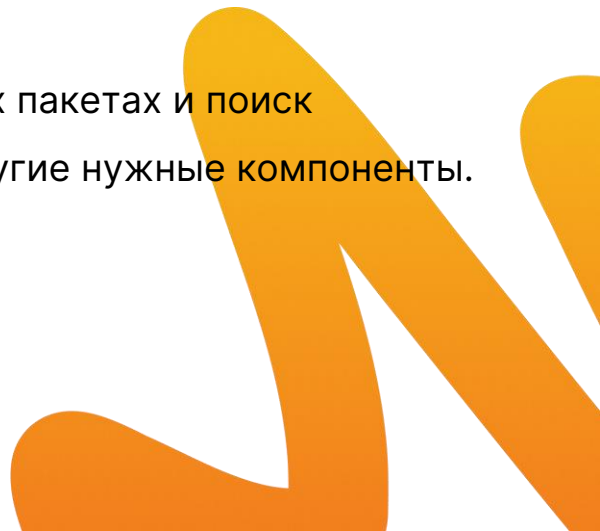


Тонкая настройка

Конфигурация с помощью аннотаций

Конфигурация с помощью аннотаций предполагает

- размещение метаданных в аннотациях классов и членов классов, а не в *xml*-файле, т.е. использование возможностей аспектно-ориентированного программирования (AOP);
- создание дополнительных классов конфигурации;
- применение контейнера *ApplicationContext*, т.к. контейнер *BeanFactory* поддерживает только *xml*-конфигурацию;
- автоматическое сканирование *Spring*ом классов в указанных пакетах и поиск аннотаций, указывающих на бины, файлы конфигураций и другие нужные компоненты.



Тонкая настройка **@Component**

Аннотацией *@Component* помечаются *POJO-классы*, объекты которых нужно превратить в бины. *Spring* будет сканировать классы на наличие этой аннотации. *@Component* нельзя указывать над интерфейсом, потому что нельзя создать объект из интерфейса.

Было

```
<bean id="firstComponent" class="org.example.pojo.FirstComponent"/>
```

Стало

```
@Component  
public class FirstComponent { }
```

Имя бина можно задать в параметрах аннотации. Если его не указать, то *Spring* укажет в качестве имени бина имя класса, написанное с маленькой буквы.

```
@Component("componentAnotherName")  
public class FirstComponent { }
```

Тонкая настройка

Пример использования @Component



SpringAnnotationComponentExample.zip



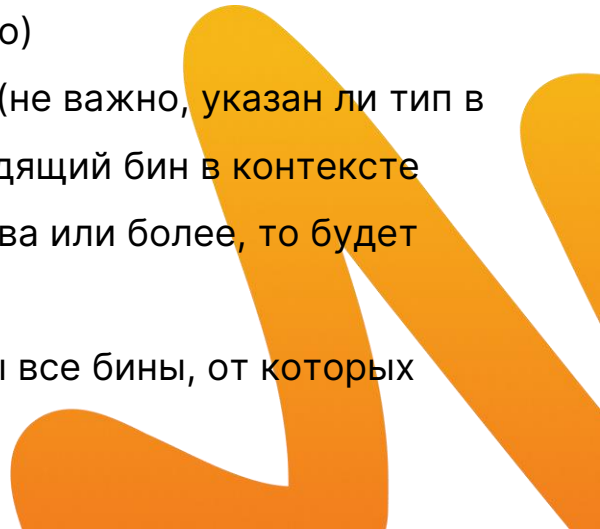
Тонкая настройка **@Autowired**

Автоматическое внедрение зависимостей можно выполнить с помощью аннотации *@Autowired*. Этой аннотацией помечаются либо:

- конструктор
- сеттеры
- поля (не рекомендуется инъекция на уровне полей, т.к. это приводит к применению Reflection API для приватных полей, что ломает инкапсуляцию)

Spring подбирает бин, подходящий для данного типа аргумента (не важно, указан ли тип в виде интерфейса или в виде класса) и внедряет его. Если подходящий бин в контексте отсутствует, то будет брошено исключение. Если бинов будет два или более, то будет брошено другое исключение.

Spring сам определяет порядок инициализации бинов так, чтобы все бины, от которых зависят другие, были созданы первыми.



Тонкая настройка

Пример использования @Autowired



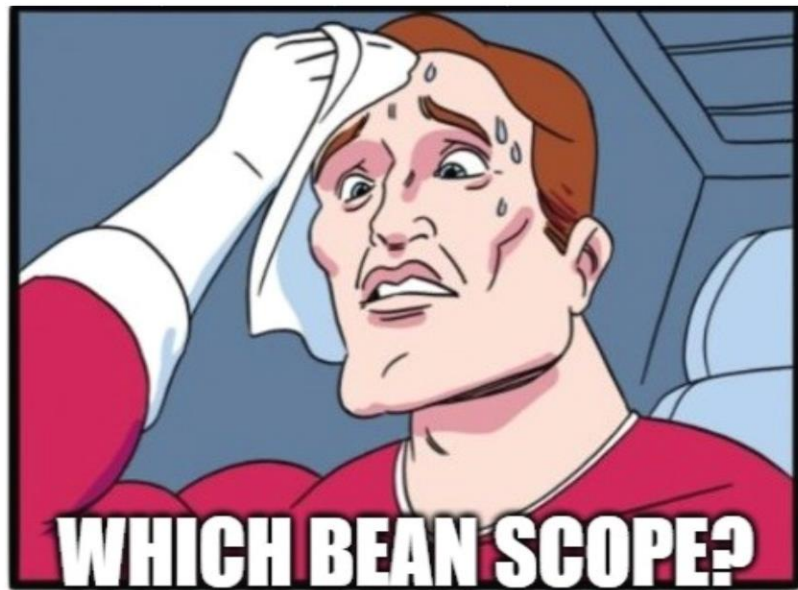
SpringAnnotationAutowiredExample.zip



Тонкая настройка

@Scope

Для указания области видимости бина применяется аннотация *@Scope*. По умолчанию её можно не указывать и, как и в xml-конфигурации, scope бина будет singleton. Можно задать его явно *@Scope("singleton")* или указать другой. Например, *@Scope("prototype")*.



Тонкая настройка

Пример использования @Scope



SpringAnnotationScopeExample.zip



Задание

Создайте POJO-класс Костюм, включающий поля Брюки, Пиджак, Рубашка, Галстук. Создайте интерфейсы для каждого элемента костюма, создайте по одной имплементации. Укажите автоматическое сканирование пакета и создание бинов. В main получите пару костюмов для Ваших друзей – Винсента Веги и Джулса Уиннфилда.

Временно закомментируйте аннотации на классе, имплементирующем интерфейс Галстук. Какое исключение было получено?

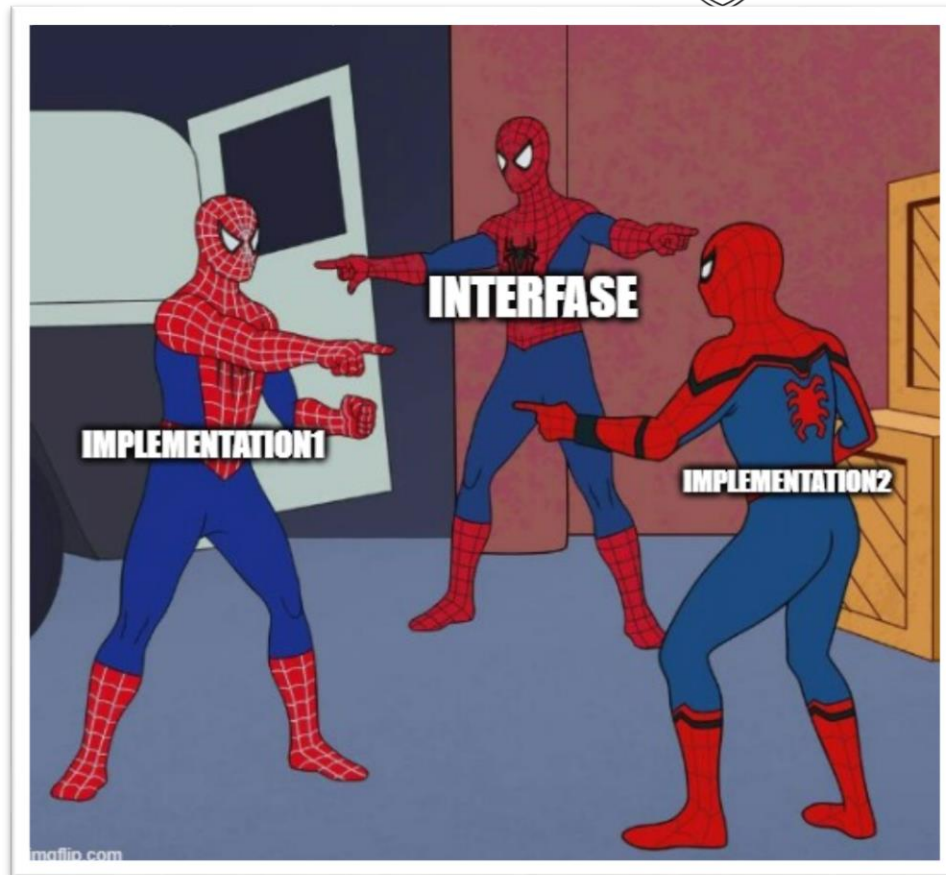
Раскомментируйте аннотации. Создайте вторую имплементацию интерфейса Галстук. Какое исключение было получено?



Проблема

Часто в приложении мы имеем множество реализаций одного и того же интерфейса. Это даёт гибкость применения одного и того же кода в похожих ситуациях, не привязываясь к конкретной реализации (например, см. паттерн [Стратегия](#)).

Каким образом мы можем делать внедрение нужного бина в автоматическом режиме?

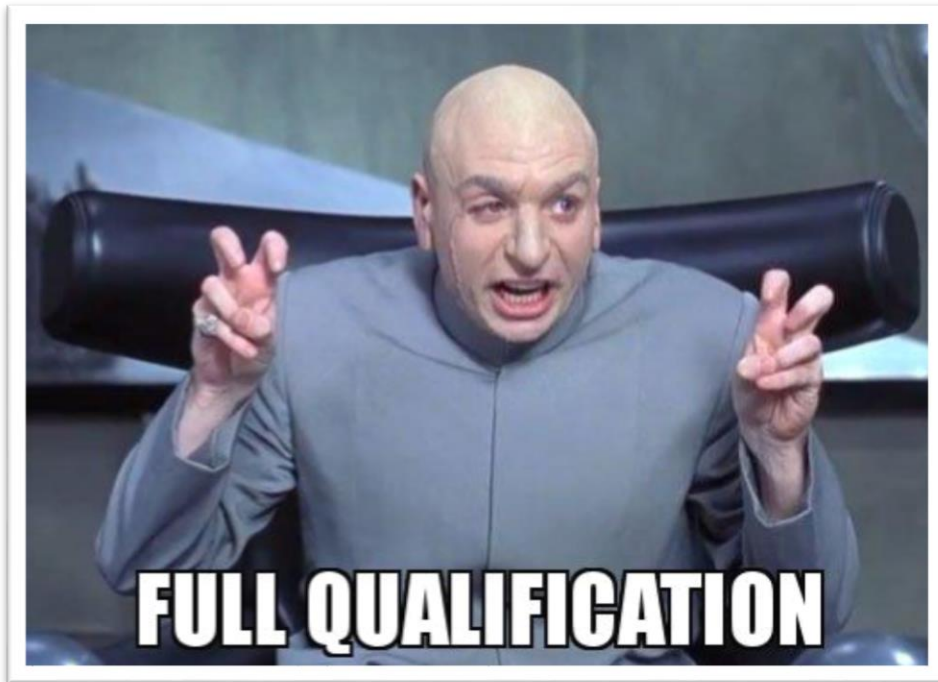


Сложный выбор

@Qualifier

@Qualifier("beanName") – это «уточнитель», бин с каким именем должен быть внедрён, если в контексте присутствует несколько подходящих бинов.

@Qualifier("beanName") или указать другой. *@Qualifier* используется совместно с *@Autowired*.



Сложный выбор

@Qualifier

При использовании *@Qualifier* на конструкторе. Аннотация переходит с конструктора в аргументы.

```
@Autowired
public MediaPlayer(@Qualifier("rockMusic") Music music1,
                  @Qualifier("classicalMusic") Music music2) {
    this.music1 = music1;
    this.music2 = music2;
}
```

Задание

Дополните задание с классом Костюм так, чтобы в костюме всегда внедрялась первая имплементация интерфейса Галстук.



3

Домашнее задание

Домашнее задание

1 Ваш класс Car зависит от интерфейса Engine. Создайте несколько реализаций интерфейса Engine (например, GasEngine и ElectricEngine). Аннотируйте их с помощью @Component с названием бинов, чтобы явно указать, какой двигатель должен быть использован в каждом случае. Затем внедрите зависимость двигателя в класс Car с помощью @Autowired и @Qualifier, чтобы можно было выбрать тип двигателя во время компиляции.

2 Создайте класс MessageGenerator, который генерирует уникальные сообщения с временной меткой. Каждый раз, когда бин запрашивается из контекста, создаётся новый экземпляр MessageGenerator. Внедрите этот бин в классы EmailSender и SmsSender.

Полезные ссылки

- Подробнее о паттернах проектирования <https://refactoring.guru/ru/design-patterns/catalog>

ЗАКЛЮЧЕНИЕ

