

# Spring - Копаем до самого ядра

Евгений Борисов  
[bsevgeny@gmail.com](mailto:bsevgeny@gmail.com)

# О чём пойдёт речь: (часть первая)

- Разные дизайн патерны
- Reflection – это круто
- Spring – основные концепции:
  - Inversion of control
  - Dependency injection
  - Spring Bean
  - BeanFactory
- Разные виды контекстов

# О чём пойдёт речь: (часть вторая)

- BeanPostProcessors
  - 4 уровня понимания
  - Написание BeanPostProcessora
  - Написание продвинутого BeanPostProcessora
- BeanFactoryPostProcessors
- ApplicationListener

# О чём пойдёт речь: (часть третья)

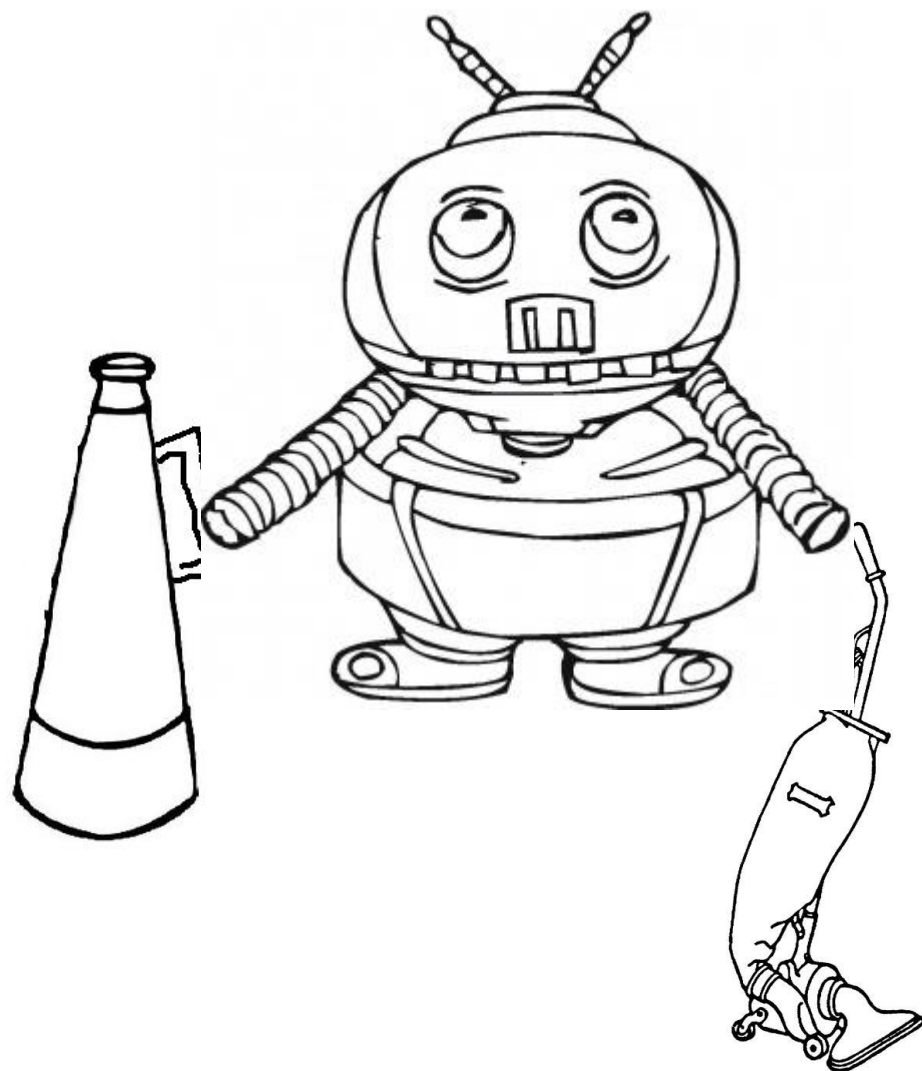
- Spring + Hibernate
- Spring + RMI
- Spring + Quartz

# Как ТЫ создаешь объекты?

- Я пользуюсь: **new**
- Я очень крут, и использую только reflections
- Мне их создают
- Зачем объекты? Есть статические методы!



А чем плохо пользоваться **new**?

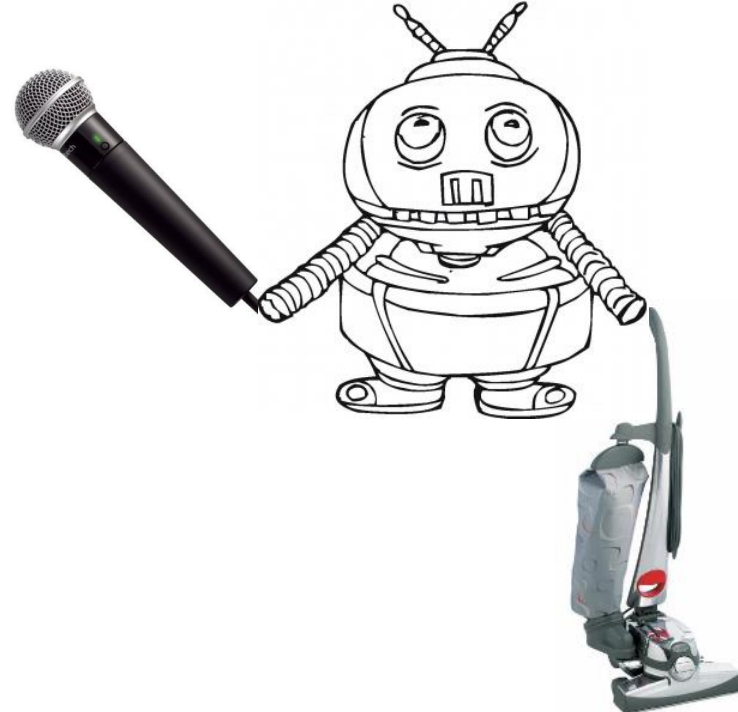


```
public class IRobot {  
    private VacuumCleaner vacuumCleaner = new VacuumCleaner();  
    private Speaker speaker = new Speaker();  
  
    public void cleanDustInTheRoom(RoomFrame room) {  
        speaker.sayJobStarted();  
        vacuumCleaner.suck();  
        room.makeRoomBrighter();  
        speaker.sayJobFinished();  
    }  
}
```

# В чём тут проблема?

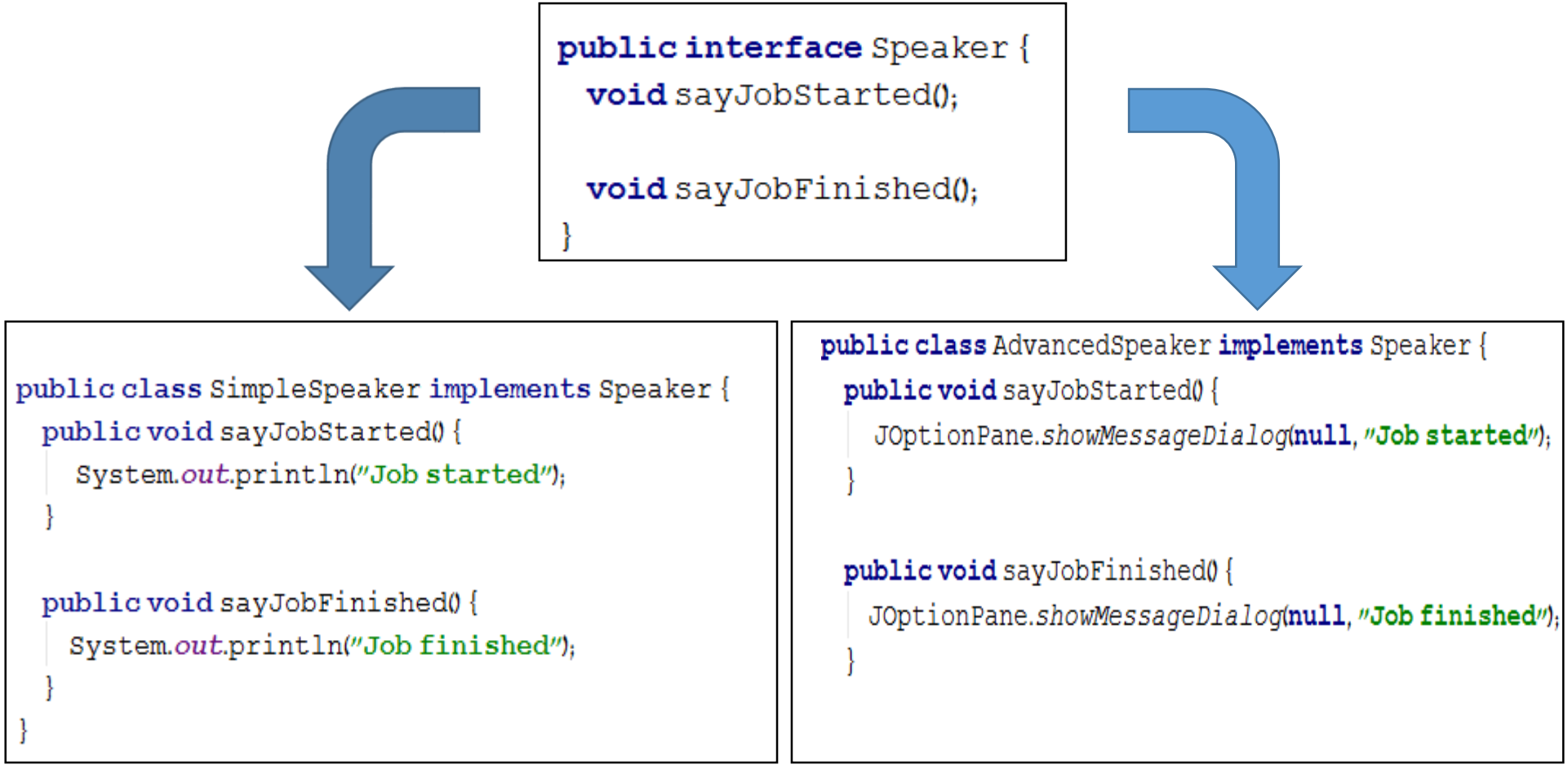
```
private VacuumCleaner vacuumCleaner = new VacuumCleaner();  
private Speaker speaker = new Speaker();
```

- А если надо поменять имплементацию, надо код вскрывать, да?





# Интерфэйс лучше, правда же?

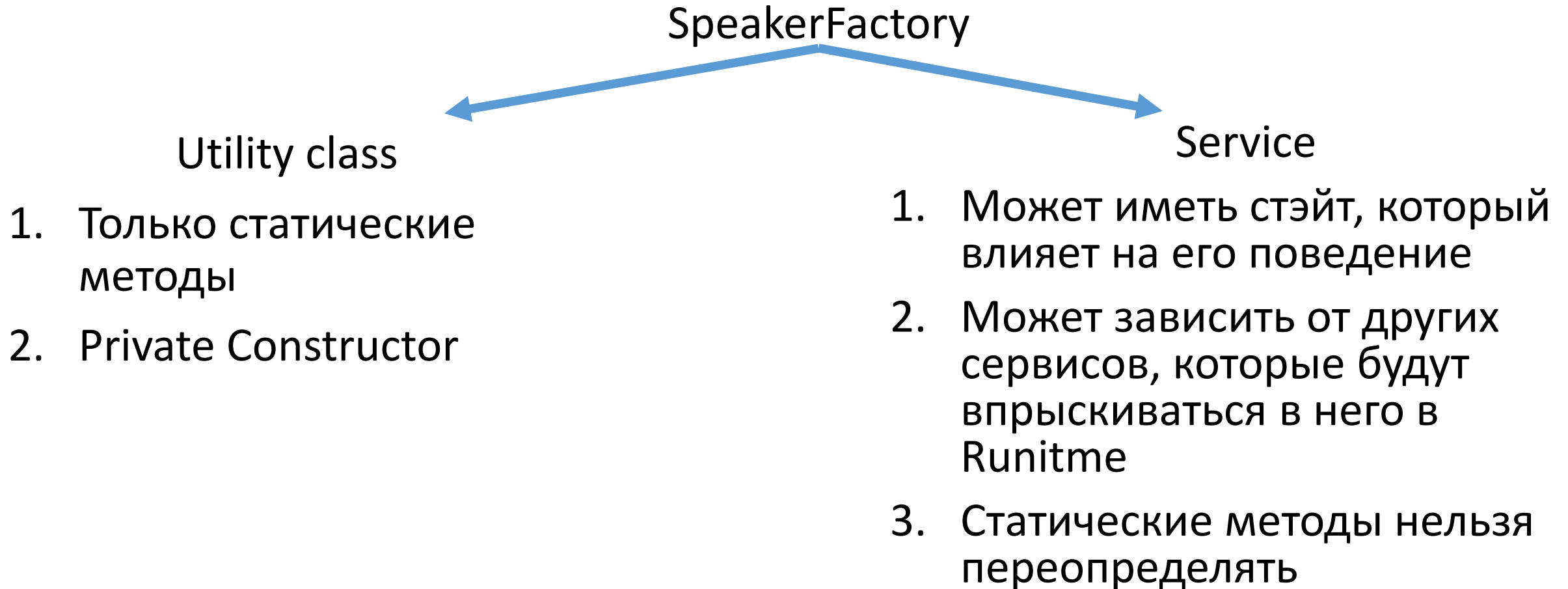


```
public interface Speaker {  
    void sayJobStarted();  
  
    void sayJobFinished();  
}
```

```
public class SimpleSpeaker implements Speaker {  
    public void sayJobStarted() {  
        System.out.println("Job started");  
    }  
  
    public void sayJobFinished() {  
        System.out.println("Job finished");  
    }  
}
```

```
public class AdvancedSpeaker implements Speaker {  
    public void sayJobStarted() {  
        JOptionPane.showMessageDialog(null, "Job started");  
    }  
  
    public void sayJobFinished() {  
        JOptionPane.showMessageDialog(null, "Job finished");  
    }  
}
```

# А кто будет решать, какая имплементация?



А ты можешь написать сингальтон?

- Шесть фаз понимая  
сингальтона:



Фаза первая:  
«Студент»



Фаза вторая:  
«Стажёр»

А что с  
мультитредингом?

```
public class SpeakerFactory {  
  
    public static SpeakerFactory speakerFactory;  
  
    private SpeakerFactory() {  
    }  
  
    public static SpeakerFactory getInstance() {  
        if (speakerFactory == null) {  
            //create new factory and initialize all it's stuff  
            speakerFactory = new SpeakerFactory();  
        }  
        return speakerFactory;  
    }  
  
    public Speaker createSpeaker(){  
        //do all business logic  
        return new SimpleSpeaker();  
    }  
}
```

# Фаза третья: «Junior Software Engineer»

А что с  
перформенсом?

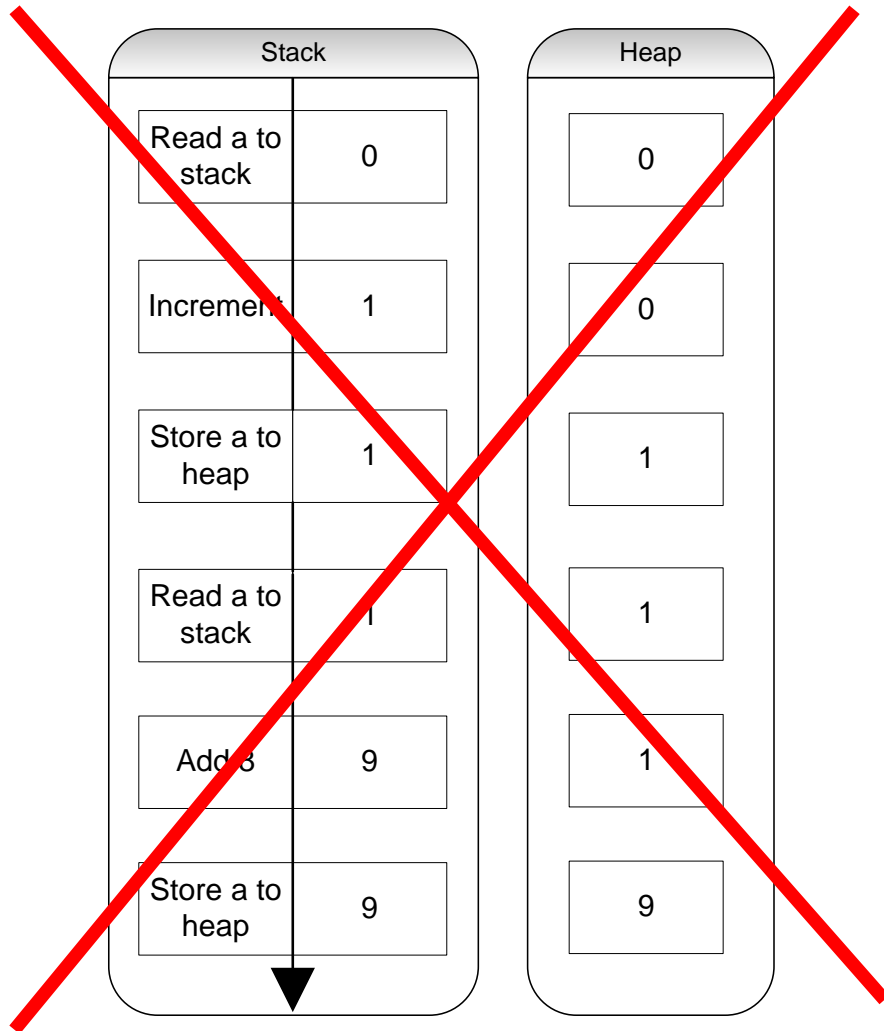
```
public class SpeakerFactory {  
  
    public static SpeakerFactory speakerFactory;  
  
    private SpeakerFactory() {  
    }  
  
    public synchronized static SpeakerFactory getInstance() {  
        if (speakerFactory == null) {  
            //create new factory and initialize all it's stuff  
            speakerFactory = new SpeakerFactory();  
        }  
        return speakerFactory;  
    }  
  
    public Speaker createSpeaker(){  
        //do all business logic  
        return new SimpleSpeaker();  
    }  
}
```

Фаза четвертая:  
«Senior Software  
Engineer»

А что на счёт  
джава  
оптимизаций?

```
public class SpeakerFactory {  
  
    public static SpeakerFactory speakerFactory;  
  
    private SpeakerFactory() {  
    }  
  
    public static SpeakerFactory getInstance() {  
        if (speakerFactory == null) {  
            synchronized (SpeakerFactory.class) {  
                if (speakerFactory == null) {  
                    //all needed stuff for initialization  
                    speakerFactory = new SpeakerFactory();  
                }  
            }  
        }  
    }  
}
```

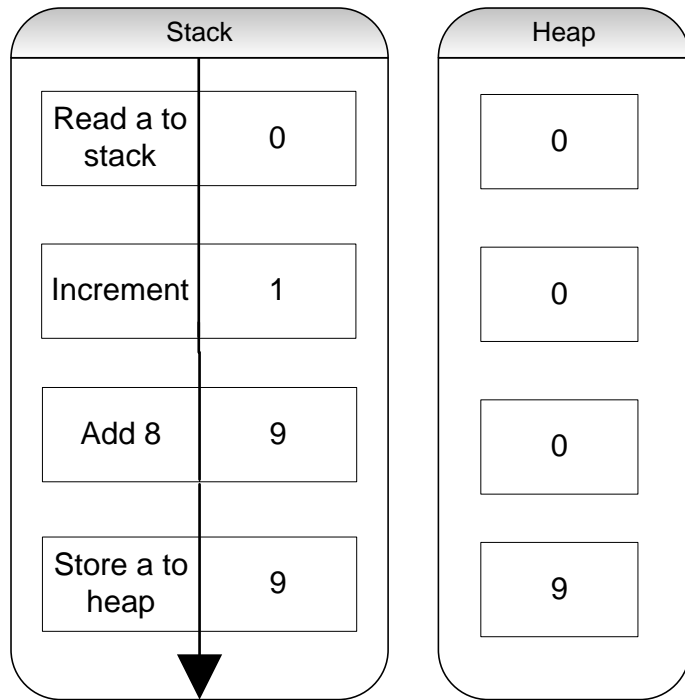
# Out-Of-Order Execution



```
1 public class OutOfOrder {  
2  
3     private int a;  
4  
5     public void foo() {  
6         a++;  
7         a+=8;  
8     }  
9  
10 }
```



# Out-Of-Order Execution



```
1 public class OutOfOrder {  
2  
3     private int a;  
4  
5     public void foo() {  
6         a++;  
7         a+=8;  
8     }  
9  
10 }
```

Фаза пятая:  
«Lead Software  
Engineer»

```
public class SpeakerFactory {
```

```
    public volatile static SpeakerFactory speakerFactory;
```

```
    private SpeakerFactory() {  
    }
```

```
    public static SpeakerFactory getInstance() {  
        if (speakerFactory == null) {  
            synchronized (SpeakerFactory.class) {  
                if (speakerFactory == null) {  
                    //all needed stuff for initialization  
                    speakerFactory = new SpeakerFactory();  
                }  
            }  
        }  
        return speakerFactory;  
    }  
}
```

# Почему не надо писать сингальтон

- Уверенность в завтрашнем дне
- Пишите вашу бизнес логику, а не изобретайте колесо
- А как вы будете тестировать?

# Шестая Фаза «Архитектор»

- Не надо писать сингальтоны, для этого есть спринг.



# Что нам даёт Spring?

- Много дизайн патернов из коробки
  - Strategy, Factory, Singleton, Proxy
- Dependency injection
- Можно тестировать без сервера приложений
- Интеграция с огромным количеством технологий
  - Hibernate, Quartz, JMS, RMI, WEB...
- AOP
- Спринг матёрый

# Немного истории

- В то время, когда программисты всё делали вручную и проклинали EJB 2 и J2EE 1.3
- В те далекие годы, когда EJB ещё писались при помощи 100 500 интерфейсов и 100 500 XML
- Когда ещё в джаве не было аннотаций
- Короче в 2002 году

# Немного истории

- Первый релиз спринга в 2002 году
- Spring 1.0 вышел в марте 2004 года
- В конце 2005 спринг становится лидерующей J2EE платформой
- В августе 2009 Vmware покупает спринг

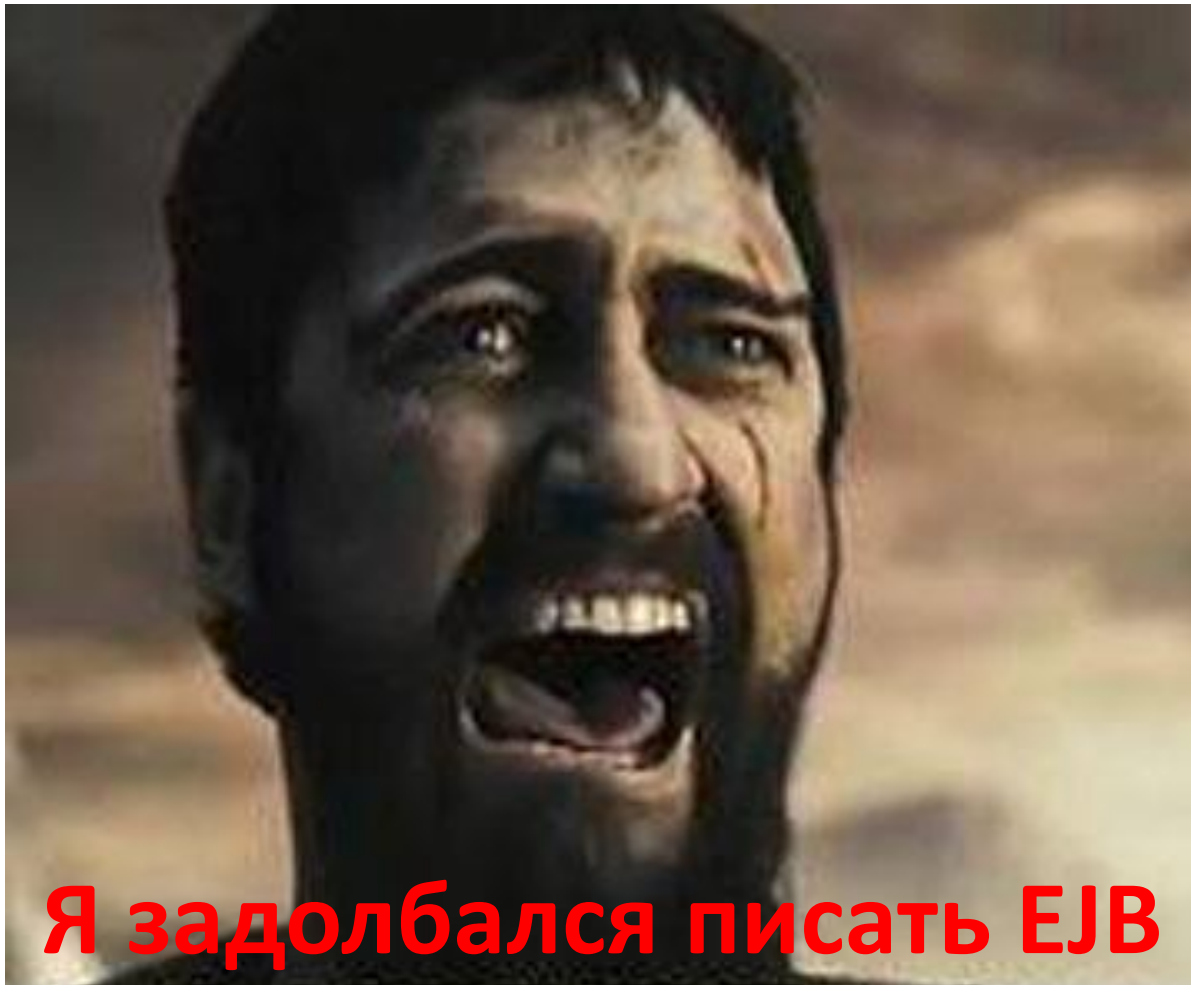
# Что сегодня

- Spring 3.2.5
- Spring 4.0 – выходит 2.12.2013 (Через 3 дня)
- 15 октября вышел Spring 4.0 RC1



# Философия спринга

- Или о чём думал Род Джонсон в 2002 году



**Я задолбался писать EJB**

# Inversion of control



# Dependency injection





J2EE должен быть более дружелюбным



**ДОЛОЙ CHECKED  
EXCEPTIONS**



# Еще философия спринга

- Работать надо против интерфейсов, имплементация меняется
- В джаве есть много конвенций
- Объектно ориентирование программирование – наше всё!
- Тесты – это очень важно

# А можно примеры IoC?

- Applet, Midlet
- Servlets instantiation & life-cycle.
- EJBs life-cycle.
- Event-driven applications.
- **Dependency Injection.**

# Dependency Injection

- Помогает решать проблему coupling-a
- Coupling – если модуль А связан с В то, А не может быть использован без В

```
public class IRobot {  
    private VacuumCleaner vacuumCleaner = new VacuumCleaner();  
    private Speaker speaker = new Speaker();  
  
    public void cleanDustInTheRoom(RoomFrame room) {  
        speaker.sayJobStarted();  
        vacuumCleaner.suck();  
        room.makeRoomBrighter();  
        speaker.sayJobFinished();  
    }  
}
```



# Configure Spring Context with 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">  
  
</beans>
```

Что такое Бин?



# Что такое Спринг Бин

- **SpringBeans** are [reusable software components](#) for [Java](#).
- Короче любой класс, главное, чтобы не статические методы 😊
  - Может быть сервисом
  - Может быть POJO
  - Может быть нашим классом
  - Может быть 3-d party library

# Определение бинов в xml-е

```
<!--Devices-->  
<bean id="speaker" class="devices.RussianSpeaker" scope="prototype"/>  
<bean id="vacuumCleaner" class="devices.VacuumCleaner" scope="prototype"/>
```

```
<bean class="robots.IRobot" name="iRobot">  
  <property name="speaker" ref="speaker"/>  
  <property name="vacuumCleaner" ref="vacuumCleaner"/>  
</bean>
```

# Как дают название бину в xml-е

- Attribute ID
- Attribute name
- При помощи тага <alias> можно добавить ещё одно имя уже существующему бину
- `<alias name="fromName" alias="toName"/>`

# Важные вещи про базовые настройки бина

- Атрибут класс – получает полное имя класса (включая все пакэджи)
- По умолчанию scope = singleton (это не вам не guice)
- По умолчанию все сингальтоны создаются при поднятии контекста
- Это можно изменить в тэге <beans...> default-lazy=true
- Или для конкретного бина: lazy=true

# Как работает таг <property>?

- Property – подразумевает наличие сетора. (таковы конвенции)
- Зная названия property легко догадаться, как будет выглядеть setter: `setPropertyname(...)`
- При помощи reflection вызывается setter и в него передаётся то, что попросили

Стоп! А насколько хорошо вы знаете Reflection?



# Опрос: ваш уровень знания Reflection



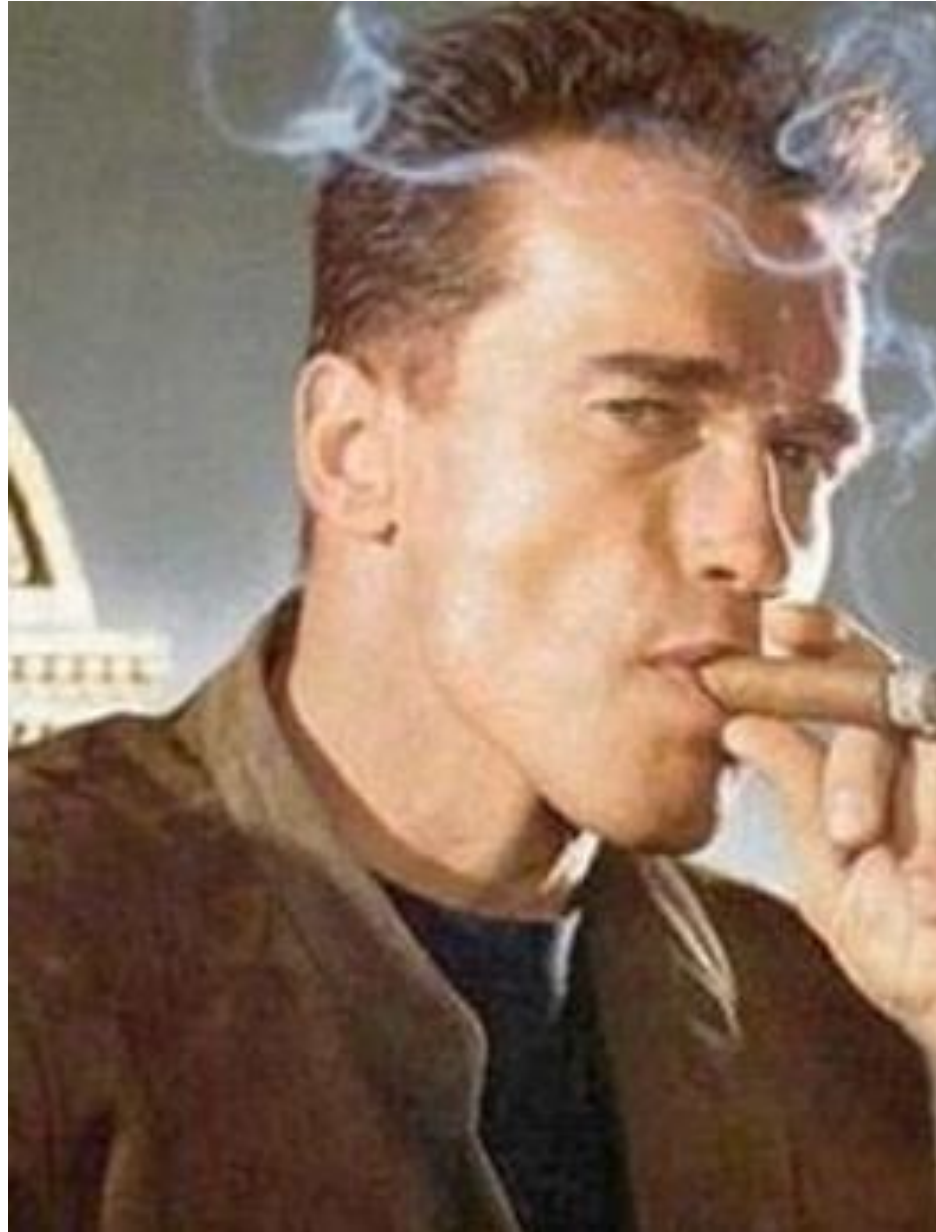
1. Я никогда не слышал, что такое reflection
2. Я знаю зачем он нужен
3. Я его использую
4. Я писал свои аннотации и считывал их в runtime
5. Я знаю разницу между RetentionPolicy.RUNTIME, RetentionPolicy.SOURCE и RetentionPolicy.CLASS



# Задание

- Написать свою аннотацию, скажем `RunThisMethod`
- У неё должен быть параметр `repeat`
- Написать `ObjectFactory` с методом `createObject`
- Метод получает тип класса, и возвращает объект данного типа, но если в этом классе, есть методы, аннотированные `@RunThisMethod` `ObjectFactory` должна из запустить соответственно значению параметра `repeat`

Вот теперь мы знаем Reflection



# Что можно впрыскивать в бины?

1. Другие бины при помощи атрибута **ref**
2. primitives and Strings. При помощи атрибута **value**
3. Collections.
  1. List
  2. Set
  3. Map

# Впрыскивание value в бин

```
<bean class="quoters.Person">  
    <property name="firstName" value="Jack"/>  
    <property name="age" value="35"/>  
</bean>
```

# Впрыскивание бина в бин

```
<!--Devices-->
```

```
<bean id="speaker" class="devices.RussianSpeaker" scope="prototype"/>
```

```
<bean id="vacuumCleaner" class="devices.VacuumCleaner" scope="prototype"/>
```

```
<bean class="robots.IRobot" name="iRobot">
```

```
  <property name="speaker" ref="speaker"/>
```

```
  <property name="vacuumCleaner" ref="vacuumCleaner"/>
```

```
</bean>
```



# Впрыскивание листа

```
<property name="messages">
  <list value-type="java.lang.String">
    <value>Shalom</value>
    <value>Privet</value>
    <value>Hi</value>
    <value>Guten Tak</value>
  </list>
</property>
```

# Как вытащить бин из контекста



# Создание и использование контекста

```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(xmlName);  
IRobot iRobot = (IRobot) context.getBean("iRobot");  
iRobot.cleanDustInTheRoom(room);
```

- Это называется LookUp
- LookUp можно делать по имени бина, по названию его класса, и по названию его интерфейса
- Его используют для тестирования
- Или для интеграции с легаси фреймворками, аля struts 1
- А как же в продакшоне?



# Какие бывают аппликации?

- Event Driven



1. Определить все бины
2. Впрыснуть все зависимости
3. Поднять контекст

- Scheduled



1. Определить все бины
2. Впрыснуть все зависимости
3. Если это аппликация аля мэйн -> сконфигурировать init method у главного бина.
4. Если это scheduled -> то настроить джоб кварца

# Задание

- Написать interface `quoter` с методом `sayQuote` и 2 имплементации
  - `ShakeSpearQuoter` (с property message) and `TerminatorQuoter` (messages)
- По вызову метода `sayQuote` они печатаю все меседжи
- Прописать их как бины в контексте и впрыснуть в xml-е им месседжи
- Протестировать
  - Создать контекст и сделать лукап
  - Попробовать сделать лукап не только по имени, но и по интерфейсу.
  - Объяснить exception

# Как настроить init method у бина?

- Если мы говорим про контекст xml то есть атрибут init-method
- Можно прописать его для конкретного бина в том же теге где он определяется
- Можно воспользоваться атрибутом default-init-methods для всех бинов, прописав этот метод в теге beans

Задание - Впрыснем Терминатора и Шекспира в третий бин



# Задание

- Написать интерфэйс с имплементацией TalkingRobot у которого будет List<Quoter> и (прописать метод talk, чтобы он делегировал на все методы quote из всех Quoter-ов)
- Определить этот бин в xml-е и впрыснуть ему оба quoter-a
- Определить метод talk, как init-method
- В тесте только создавать контекст
- Изменить score на prototype
- Почему тест перестал печатать?

# Constructor Injection

- Таг у бина: <constructor-arg>
- Имеет атрибуты: type, index (начинается с 0), name и value

```
<bean class="heroes.Wizard">  
    <constructor-arg index="1" value="Серый"/>  
    <constructor-arg index="0" value="Гендальф"/>  
</bean>
```

```
<constructor-arg value="12" type="int"></constructor-arg>
```

- А сейчас я покажу фокус

# Задание

- Создать бин класса `String`, внести в него value (“trust me”)
- Инжектнуть его в лист цитат терминатора

# А как можно еще запускать init-method

- Init Method может определить тот, кто настраивает контекст
- А что, если это хочет сделать разработчик класса?
  1. Можно имплементировать интерфейс InitializingBean, и прописать в нём метод: afterPropertySet
- Но это было очень давно, когда ещё не было аннотаций
  2. Можно поставить @PostConstruct над нужным методом



# У меня вопрос

1. А что будет если несколько `@PostConstruct` поставить?
2. А на хрена это вообще надо?  
Ведь есть же конструктор, туда и надо фигачить инициализирующую логику



# У меня загадка... Точнее две

```
public class Parent {  
  
    public Parent() {  
        printPi();  
    }  
  
    public void printPi(){  
        System.out.println("Pi");  
    }  
}
```

```
public static void main(String[] args) {  
    Son son = new Son();  
}
```

```
public class Son extends Parent {  
    private double pi = Math.PI;  
  
    public Son() {  
        printPi();  
    }  
  
    @Override  
    public void printPi() {  
        System.out.println(pi);  
    }  
}
```

?

Answer: 0.0  
3.141592653589793

# Расположите в правильной последовательности

- @PostConstruct
- Spring setter (annotation) injection
- Son Initializer
- Parent Initializer
- Son inline
- Parent Inline
- Son Constructor
- Parent Constructor

# Правильная последовательность

- Parent Inline
- Parent Initializer
- Parent constructor
- Son Inline
- Son Initializer
- Son constructor
- Spring setter (annotation) injection
- @PostConstruct Или init methods

Зачем это нужно знать???

Разве только для интервью!!!



# Практическое применение ЭТИХ ЗНАНИЙ

```
public class BestService {  
    public BestService(){  
        chuckNorrisMethod();  
    }  
}
```

```
public void chuckNorrisMethod() {  
    out.println("Save the world");  
}
```

```
public static void main(String[] args) {  
    new BetterThanBestService();  
}
```

```
public class BetterThanBestService extends BestService {  
    private List cache = new ArrayList();  
}
```

@Override

```
public void chuckNorrisMethod() {  
    cache.add(1);  
}
```

?

1. Will not compile.
2. Runtime exception.
3. You can't override Chuck Norris methods.
4. Everything is ok.

# А теперь реальное применение:

```
public abstract class AbstractFtlDataFiller implements FtlDataFiller {  
    @Inject  
    private Environment environment;  
    @Inject  
    private VOFactory factory;  
    private String ftlPath;  
    @PostConstruct  
    private void findFtlPath() {  
        // some logic using factory and environment  
    }  
}
```

---

Вроде всё нормально да?