

# Лекция №1

Inversion of Control  
&  
Dependency Injection

# Содержание курса

IoC и DI

Spring MVC + Hibernate + AOP

AOP

Spring REST

Hibernate

Spring Security

Spring MVC

Spring Boot

# Spring

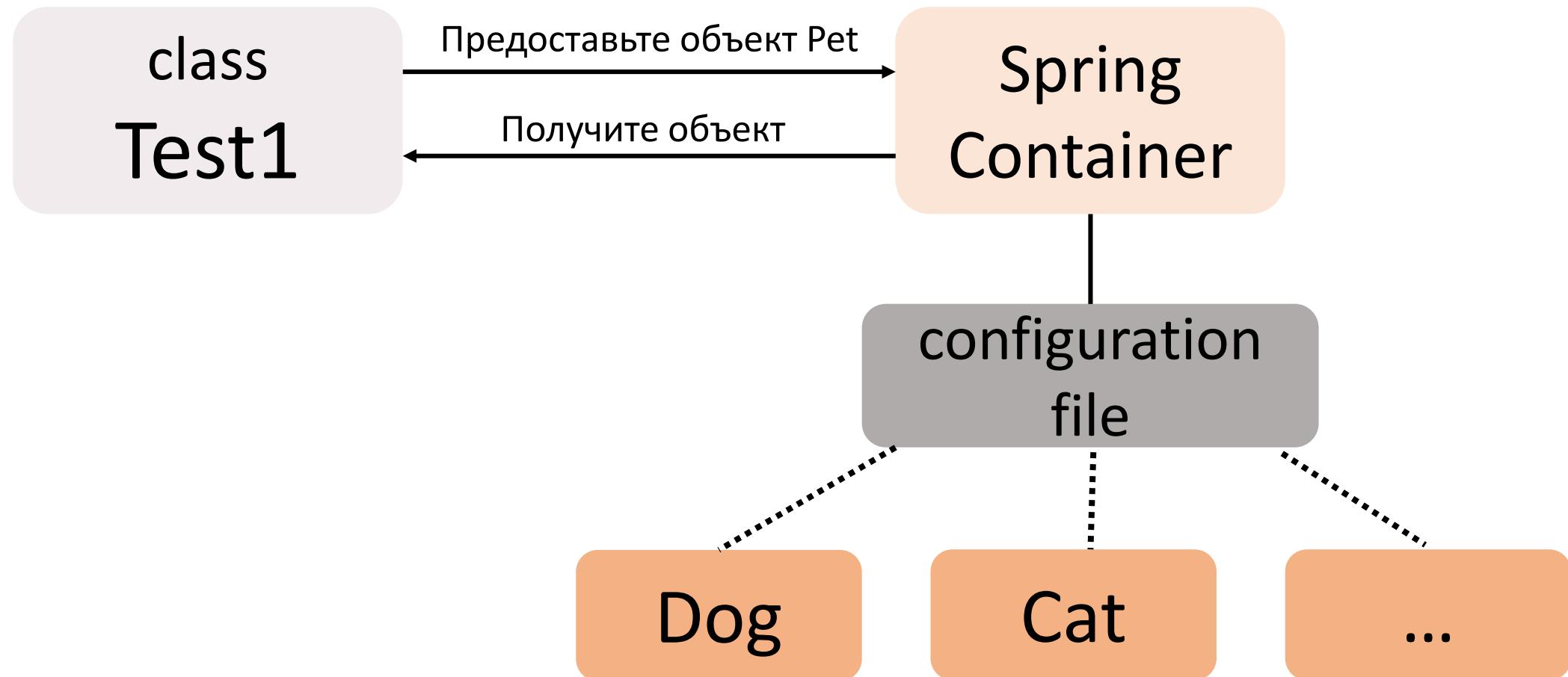
Spring – это фреймворк,  
предназначенный для более быстрого и  
простого построения Java приложений

Год создания: 2004

Последняя версия (на 16.12.2021) – **Spring 6**

Официальный сайт: [www.spring.io](http://www.spring.io)

# Inversion of Control



# Inversion of Control

Основные функции, которые выполняет  
Spring Container:

- IoC – инверсия управления  
Создание и управление объектами
- DI – Dependency Injection  
Внедрение зависимостей

IoC – аутсорсинг создания и управления  
объектами. Т.е. передача программистом прав  
на создание и управление объектами Spring-у.

# Inversion of Control

Способы конфигурации Spring Container:

- XML file (устаревший способ)
- Annotations + XML file (современный способ)
- Java code (современный способ)

# Inversion of Control

Конфигурация XML файла:

```
<bean id = "myPet"
      class = "ioc.Cat">
</bean>
```

- id – идентификатор бина
- class – полное имя класса

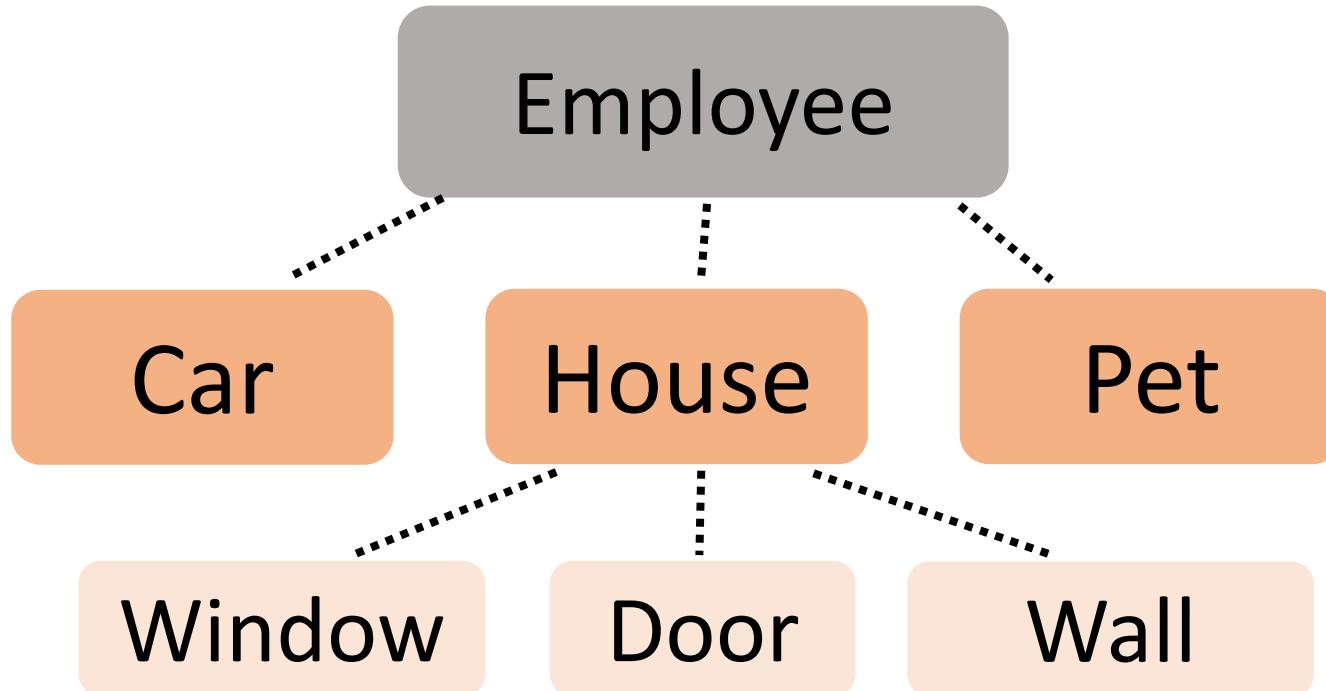
# Inversion of Control

Spring Bean (или просто bean) – это объект, который создаётся и управляется Spring Container

ApplicationContext представляет собой Spring Container. Поэтому для получения бина из Spring Container нам нужно создать ApplicationContext.

```
ClassPathXmlApplicationContext context =  
    new ClassPathXmlApplicationContext( configLocation: "applicationContext.xml" );  
  
Pet pet = context.getBean( name: "myPet", Pet.class );
```

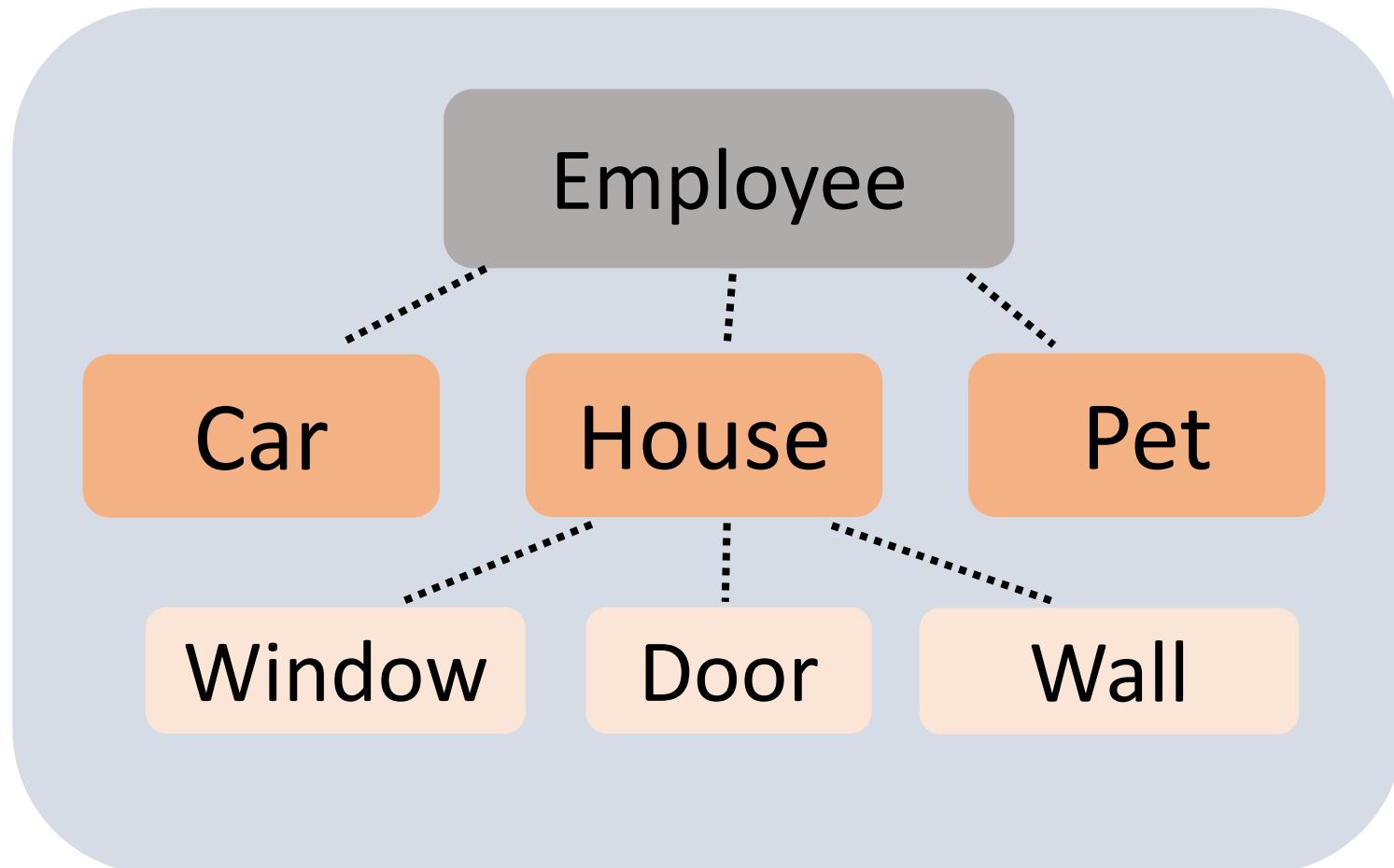
# Dependency Injection



```
Employee emp = new Employee();
Car car = new Car();           emp.setCar(car);
House house = new House();
Door door = new Door();       house.setDoor(door);
```

# Dependency Injection

Spring Container



# Dependency Injection

DI – аутсорсинг добавления/внедрения зависимостей. DI делает объекты нашего приложения слабо зависимыми друг от друга.

Способы внедрения зависимостей:

- С помощью конструктора
- С помощью сеттеров
- Autowiring

# Dependency Injection

DI с помощью конструктора:

```
<bean id = "myPet"
      class = "ioc.Cat">
</bean>
```

За кулисами:

```
Cat myPet = new Cat();
```

```
<bean id = "myPerson"
      class="ioc.Person">
    <constructor-arg ref="myPet"/>
</bean>
```

```
Person myPerson = new Person(myPet);
```

- constructor-arg – аргумент конструктора
- ref – ссылка на bean id

# Dependency Injection

DI с помощью сеттера:

```
<bean id = "myPet"
      class = "ioc.Cat">
</bean>
```

```
<bean id = "myPerson"
      class="ioc.Person">
    <property name="pet" ref="myPet"/>
</bean>
```

За кулисами:

```
Cat myPet = new Cat();
```

```
Person myPerson = new Person();
myPerson.setPet(myPet);
```

Первая буква в слове «pet» становится заглавной и в начало слова добавляется «set». После чего вызывается получившийся метод.

# Dependency Injection

Внедрение строк и других значений:

```
<bean id = "myPerson"
      class="ioc.Person">
    <property name="surname" value="Tregulov"/>
    <property name="age" value="33"/>
</bean>
```

За кулисами:

```
Person myPerson = new Person();
myPerson.setSurname("Tregulov");
myPerson.setAge(33);
```

- value – значение, которое мы хотим присвоить.

# Dependency Injection

Внедрение строк и других значений из properties файла:

```
person.surname = Tregulov  
person.age = 33
```

```
<context:property-placeholder location="classpath:myApp.properties"/>
```

```
<bean id = "myPerson"  
      class="ioc.Person">  
    <property name="surname" value="${person.surname}" />  
    <property name="age" value="${person.age}" />  
</bean>
```

За кулисами:

```
Person myPerson = new Person();  
myPerson.setSurname("Tregulov");  
myPerson.setAge(33);
```

# IoC & DI

IoC – аутсорсинг создания и управления объектами. Т.е. передача программистом прав на создание и управление объектами Spring-у.

DI – аутсорсинг добавления/внедрения зависимостей. DI делает объекты нашего приложения слабо зависимыми друг от друга.

Большое количество программистов используют эти термины как взаимозаменяемые.

# Bean scope

Scope (область видимости) определяет:

- жизненный цикл бина
- возможное количество создаваемых бинов

Разновидности bean scope:

**singletone**

**prototype**

**request**

**session**

**global-session**

# Bean scope

singleton – дефолтный scope.

- такой бин создаётся сразу после прочтения Spring Container-ом конфиг файла.
- является общим для всех, кто запросит его у Spring Container-а.
- подходит для stateless объектов.

```
<bean id="myPet"
      class="ioc.Dog"
      scope="singleton">
</bean>
```

```
Dog myDog = context.getBean( name: "myPet", Dog.class);
```

```
Dog yourDog = context.getBean( name: "myPet", Dog.class);
```

Spring Container

Dog

# Bean scope

## prototype

- такой бин создаётся только после обращения к Spring Container-у с помощью метода getBean.
- для каждого такого обращения создаётся новый бин в Spring Container-е.
- подходит для stateful объектов.

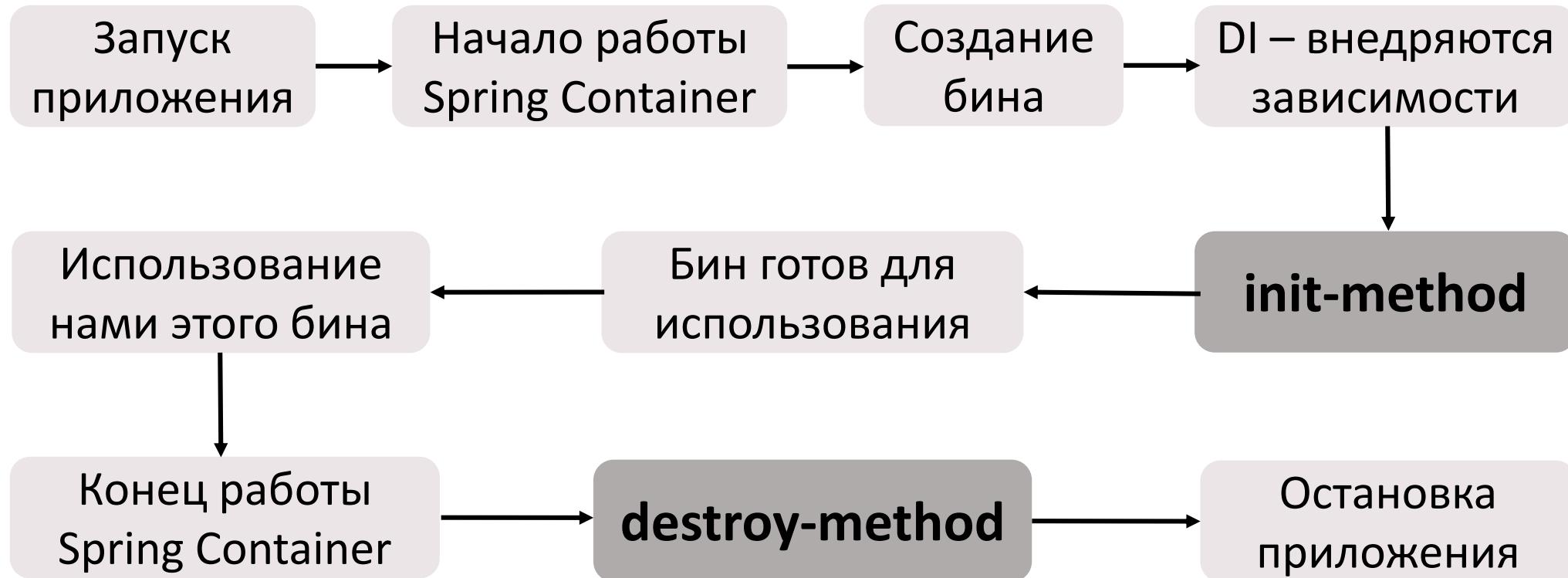
```
<bean id="myPet"
      class="ioc.Dog"
      scope="prototype">
</bean>
Dog myDog = context.getBean( name: "myPet", Dog.class );
Dog yourDog = context.getBean( name: "myPet", Dog.class );
```

Spring Container

Dog

Dog

# Жизненный цикл бина



Чаще всего **init-method** используется для открытия или настройки каких-либо ресурсов, например баз данных, стримов и т.д. **destroy-method** чаще всего используется для их закрытия.

# Методы init и destroy

```
public void init() {
    System.out.println("Class Dog: init method");
}

public void destroy() {
    System.out.println("Class Dog: destroy method");
}
```

```
<bean id="myPet"
      class="ioc.Dog"
      init-method="init"
      destroy-method="destroy">
</bean>
```

# Методы `init` и `destroy`

У данных методов access modifier может быть любым

У данных методов return type может быть любым. Но из-за того, что возвращаемое значение мы никак не можем использовать, чаще всего return type – это void.

Называться данные методы могут как угодно.

В данных методах не должно быть параметров.

# Методы `init` и `destroy`

Если у бина scope = prototype, то:

- `init-method` будет вызываться для каждого новосозданного бина.
- для этого бина `destroy-method` вызываться не будет
- программисту необходимо самостоятельно писать код для закрытия/освобождения ресурсов, которые были использованы в бине

# Конфигурация с помощью аннотаций

Аннотации – это специальные комментарии/метки/метаданные, которые нужны для передачи определённой информации.

Конфигурация с помощью аннотаций более короткий и быстрый способ, чем конфигурация с помощью XML файла.

Процесс состоит из 2-х этапов:

1. сканирование классов и поиск аннотации  
`@Component`
2. Создание (регистрация) бина в Spring Container-е

# Конфигурация с помощью аннотаций

```
<context:component-scan base-package="spring_introduction"/>
```

```
import org.springframework.stereotype.Component;

@Component("catBean")
public class Cat implements Pet {
    public Cat() {
        System.out.println("Cat bean is created");
    }
    @Override
    public void say() { System.out.println("Meow-Meow"); }
}
```

```
Cat myCat = context.getBean(name:"catBean", Cat.class);
```

# Конфигурация с помощью аннотаций

Если к аннотации `@Component` не прописать bean id, то бину будет назначен дефолтный id.

Дефолтный bean id получается из имени класса, заменяя его первую заглавную букву на прописную.

```
@Component → cat
```

```
class Cat {
```

```
@Component → favoriteSong
```

```
class FavoriteSong{
```

```
@Component → SQLTest
```

```
class SQLTest{
```

# @Autowired

Для внедрения зависимостей с помощью аннотаций используется аннотация `@Autowired`

Типы autowiring-а или где мы можем использовать данный DI:

- Конструктор
- Сеттер
- Поле

# @Autowired

Процесс внедрения зависимостей при использовании @Autowired такой:

1. Сканирование пакета, поиск классов с аннотацией @Component
2. При наличии аннотации @Autowired начинается поиск подходящего по типу бина

Далее ситуация развивается по одному из сценариев:

- Если находится 1 подходящий бин, происходит внедрение зависимости;
- Если подходящих по типу бинов нет, то выбрасывается исключение;
- Если подходящих по типу бинов больше одного, тоже выбрасывается исключение.

# @Autowired

```
@Component("catBean")
public class Cat implements Pet {
```

## Constructor injection

```
@Component("personBean")
public class Person {
    private Pet pet;

    @Autowired
    public Person(Pet pet) {
        this.pet = pet;
    }
}
```

## Setter injection

```
@Autowired
public void setPet(Pet pet) {
    this.pet = pet;
}
```

## Field injection

```
@Autowired
private Pet pet;
```

## Any method injection

```
@Autowired
public void anyMethodName(Pet pet) {
    this.pet = pet;
}
```

# @Qualifier



Если при использовании `@Autowired` подходящих по типу бинов больше одного, то выбрасывается исключение. Предотвратить выброс данного исключения можно конкретно указав, какой бин должен быть внедрён. Для этого и используют аннотацию `@Qualifier`.

# @Qualifier



Field

```

@Autowired
@Qualifier("dog")
private Pet pet;

```

Setter

```

@Autowired
@Qualifier("dog")
public void setPet(Pet pet) {
    this.pet = pet;
}

```

Constructor

```

@Autowired
public Person(@Qualifier("dog") Pet pet) {
    this.pet = pet;
}

```

# @Value

Для внедрения строк и других значений можно использовать аннотацию @Value.

В этом случае в сеттерах нет необходимости, как это было при конфигурации с помощью XML файла.

Hardcoded вариант

```
@Value("Tregulov")
private String surname;
@Value("33")
private int age;
```

Вариант с properties файлом

```
<context:component-scan base-package="spring_introduction"/>
<context:property-placeholder location="classpath:myApp.properties"/>

@Value("${person.surname}")
private String surname;
@Value("${person.age}")
private int age;
```

# Bean scope

`singletone` – дефолтный scope.

- такой бин создаётся сразу после прочтения Spring Container-ом конфиг файла.
- является общим для всех, кто запросит его у Spring Container-a.
- подходит для stateless объектов.

`prototype`

- такой бин создаётся только после обращения к Spring Container-у с помощью метода `getBean`.
- для каждого такого обращения создаётся новый бин в Spring Container-e.
- подходит для stateful объектов.

```
@Component  
@Scope ("singleton")  
public class Dog implements Pet {
```

```
@Component  
@Scope ("prototype")  
public class Dog implements Pet {
```

# Методы `init` и `destroy`

У данных методов access modifier может быть любым

У данных методов return type может быть любым. Но из-за того, что возвращаемое значение мы никак не можем использовать, чаще всего return type – это void.

Называться данные методы могут как угодно.

В данных методах не должно быть параметров.

# Методы `init` и `destroy`

Если у бина scope = prototype, то:

- `init-method` будет вызываться для каждого новосозданного бина.
- для этого бина `destroy-method` вызываться не будет
- программисту необходимо самостоятельно писать код для закрытия/освобождения ресурсов, которые были использованы в бине

# @PostConstruct и @PreDestroy

```
@PostConstruct  
public void init() {  
  
    System.out.println("Class Dog: init method");  
}  
  
@PreDestroy  
public void destroy() {  
  
    System.out.println("Class Dog: destroy method");  
}
```

# Конфигурация Spring Container-а с помощью Java кода. Способ 1

```
@Configuration  
@ComponentScan("spring_introduction")  
public class MyConfig {  
}
```

Аннотация `@Configuration` означает, что данный класс является конфигурацией.

С помощью аннотации `@ComponentScan` мы показываем, какой пакет нужно сканировать на наличие бинов и разных аннотаций.

```
AnnotationConfigApplicationContext context =  
    new AnnotationConfigApplicationContext(MyConfig.class);
```

При использовании конфигурации с помощью Java кода, Spring Container будет представлен классом `AnnotationConfigApplicationContext`

# Конфигурация Spring Container-а с помощью Java кода. Способ 2

```
@Configuration  
public class MyConfig {  
    @Bean  
    public Pet cat() {  
        return new Cat();  
    }  
  
    @Bean  
    public Person myPerson() {  
        return new Person(cat());  
    }  
}
```

- Данный способ не использует сканирование пакета и поиск бинов. Здесь бины описываются в конфиг классе.
- Данный способ не использует аннотацию @Autowired. Здесь зависимости прописываются вручную.
- Название метода – это bean id.
- Аннотация @Bean перехватывает все обращения к бину и регулирует его создание.

# Аннотация @PropertySource

```
@Value("${person.surname}")
private String surname;
@Value("${person.age}")
private int age;
```

```
@Configuration
@PropertySource("classpath:myApp.properties")
public class MyConfig {
```

Аннотация @PropertySource указывает на property файл откуда мы можем использовать значения для полей