Лабораторная работа 9

Инверсия управления и внедрение зависимостей

Цель работы: изучить способы реализации механизма внедрения зависимостей.

1. Теоретическая часть

Инверсия управления (IoC, Inversion of Control) — это паттерн проектирования, который определяет, что объекты должны зависеть от абстракций, а не от конкретных реализаций, и что объекты должны быть созданы и настроены вне зависимых классов.

Вместо того, чтобы явно создавать и управлять объектами, разработчик определяет зависимости и описывает, как они должны быть созданы и внедрены в приложение.

Внедрение зависимостей (DI, Dependency Injection) — представляет собой процесс предоставления зависимостей объекту внешним образом, вместо того, чтобы объект самостоятельно создавать или искать зависимости.

Существует несколько видов:

- внедрение через конструктор (Constuctor Injection)
- внедрение через свойства (Property Injection)
- внедрение через параметры метода (Method Injection)

2. Практическая часть

2.1. Разработка собственного DI-контейнера

Определим две аннотации.

Autowired будет использоваться для описания способа внедрения объекта.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.CONSTRUCTOR})
public @interface Autowired {
}
```

Component будет маркировать объект (бин), которым будет управлять контейнер

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Component {
    String value() default "";
}
```

Далее определим несколько простых классов, которые должны стать бинами и использоваться механизмом DI.

```
@Component("specialService")
public class MyService {
    private final MyRepository repository;

    @Autowired
    public MyService(MyRepository repository) {
        this.repository = repository;
```

```
        }

        public String process() {
            return "Processed: " + repository.getData();
        }
    }

    @Component
    public class MyRepository {
        public String getData() {
            return "Data from repository";
        }
    }

    @Component
    public class MyController {
        @Autowired
        private MyService service;

        public void execute() {
            System.out.println(service.process());
        }
    }
```

Далее реализуем сам контейнер на основе аннотаций (код приведен в приложении 1).

Далее посмотрим как можно использовать разрботанный контейнер

```
public static void main(String[] args) throws Exception {
   AnnotationContainer container = new AnnotationContainer();

   // Сканируем и регистрируем бины
   container.scanAndRegisterBeans("ru.kafpin");

   // Получаем бин и используем
   MyController controller =
               container.getBeanByType(MyController.class);
   controller.execute();

   // Или по имени
   MyService service = container.getBeanByName("specialService");
   System.out.println(service.process());
}
```

2.2. Использование Spring Context на основе XML-конфигураций

Для начала импортируем сам фреймворк

```
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>6.2.6</version>
</dependency>
```

Рассмотрим на простом примере. Определим интерфейс для Студента.

```
public interface Student {
    String getName();
    String getKnowledge();
}
```

Определим двух студентов. Для простоты имена будем задавать сразу в коде.

```
public class JavaStudent implements Student{
    private String name = "Misha";
    @Override
    public String getName() {
        return name;
    }

    @Override
    public String getKnowledge() {
        return "Java coder";
    }
}

public class PythonStudent implements Student{
    private String name = "Masha";
    @Override
    public String getName() {
        return name;
    }

    @Override
    public String getKnowledge() {
        return "Python";
    }
}
```

Далее создадим класс Department. Здесь определим возможность внедрения зависимостей через конструктор и через метод.

```
public class Department {
    private Student student;

    public Department(Student student) {
        this.student = student;
```

```
    }

    public void setStudent(Student student) {
        this.student = student;
    }

    public void displayInfo(){
        System.out.println("Student: " + student.getName());
        System.out.println("Can coding on: " + student.getKnowledge());
    }
}
```

Сам механизм регистрации бинов и их внедрения опишем в файле appContext.xml, который должен располагаться в папке resources.

```
<?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
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="javaStudent" class="beans.JavaStudent"/>
    <bean id="pythonStudent" class="beans.PythonStudent"/>

    <bean id="department" class="Department">
        <constructor-arg ref="javaStudent"/>
    </bean>

</beans>
```

Здесь описано внедрение через параметр конструктора. В таком случае конструктор по умолчанию не нужен.

В основном приложении загружаем конфигурацию, получаем бин и можем его использовать:

```
public static void main(String[] args) {
    ClassPathXmlApplicationContext context =
                new ClassPathXmlApplicationContext("appContext.xml");
    Department department =
                context.getBean("department", Department.class);
    department.displayInfo();
    context.close();
    }
```

Далее попробуем вместо конструктора использовать внедрение через сеттер. Для этого в файле конфигурации изменим способ внедрения. Для работы этого метода в классе должен быть конструктор по умолчанию.

```
<property name="student" ref="pythonStudent"/>
```

2.3.Использование Spring Context на основе java-конфигураций

Аннотации удобны для простого создания объектов. Если необходимо реализовать дополнительную логику, описывающую процесс создания бина, то это можно сделать в отдельном файле. Вместо xml это можно сделать в java-классе.

Продолжим изучение на основе тех же примеров. В данном случае вместо описания имени студента внутри класса, предоставим возможность задать его извне. Для этого нам понадобится конструктор с параметром.

```
public class JavaStudent implements Student{
    private String name;

    public JavaStudent(String name) {
        this.name = name;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public String getKnowledge() {
        return "Java";
    }
}
```

Аналогично с классом PythonStudent.

Далее создадим отдельный класс AppConfig. Данный класс необходимо пометить аннотацией @Configuration. Также понадобится дополнительная аннотация @ComponentScan(basePackages = "ru.kafpin"), которая укажет путь для автосканирования бинов.

```
@Configuration
@ComponentScan(basePackages = "ru.kafpin")
public class AppConfig {
    @Bean
    public Student firstStudent() {
        return new JavaStudent("Misha");
    }

    @Bean
    public Student secondStudent() {
        return new PythonStudent("Masha");
    }
}
```

В этой конфигурации опишем два создаваемых бина. В данном случае регистрируем два бина с одинаковым классом, но разными идентификаторами.

Изменим класс Department. Во-первых, его нужно пометить как компонент. Во-вторых, описать внедрение зависимых бинов.

```
@Component
public class Department {
    private Student student;

    @Autowired
    public void setStudent(@Qualifier("firstStudent") Student student)
{
        this.student = student;
    }

}
```

В основном приложении загрузим конфигурацию из java-файла. Далее аналогично предыдущему примеру получим бин, проверим его работу.

```
public static void main(String[] args) {
    ApplicationContext context =
           new AnnotationConfigApplicationContext(AppConfig.class);

    Department department = context.getBean(Department.class);
    department.displayInfo();

    }
```

Также дополнительно получим сами бины из контейнера и проверим их работу:

```
Student student1 =
                context.getBean("javaStudent", JavaStudent.class);
Student student2 =
                context.getBean("pythonStudent", PythonStudent.class);
System.out.println(student1);
System.out.println(student2);
```

2.4.Использование Spring Context на основе аннотаций

Аналогично предыдущему способу будем создавать и внедрять бины. Отличие в том, что описание самих бинов будет не в отдельном файле, а на основе аннотаций.

Для того, чтобы фреймворк произвел сканирование и поиск нужных классов, необходимо в основном классе приложении включить этот механизм (вместо конфига).

```
@ComponentScan(basePackages = "ru.kafpin")
public class Program {
    public static void main(String[] args) {
        ApplicationContext context =
                new AnnotationConfigApplicationContext(Program.class);
```

Чтобы фреймоврк смог понять, что класс является компонентом, нужно этот класс пометить соответствующей аннотацией:

```
@Component("javaStudent")
public class JavaStudent implements Student{
    private String name = "Misha";
...
}

@Component
public class PythonStudent implements Student{
    private String name = "Masha";
...
}
```

Имя студента в данном случае снова задано в самом классе.

Класс Department теперь также будет описываться с помощью аннотации Component. Внедрение нужного класса Student описывается аннотацией Autowired по имени бина:

```
@Component
public class Department {
    private Student student;

    @Autowired
    public void setStudent(@Qualifier("javaStudent") Student student) {
        this.student = student;
    }

    public void displayInfo(){
        System.out.println("Student: " + student.getName());
        System.out.println("Can coding on: " + student.getKnowledge());
    }

}
```

Задание на лабораторную работу:

1. Изучить работу класса `AnnotationContainer`. Добавить ко всем его методам JavaDoc.
2. Изучить реализацию DI с помощью Spring Context на основе xml-конфигурации. Добавить еще одного студента и внедрить его в класс Department.
3. Изучить реализацию DI с помощью Spring Context на основе java-конфигурации.
4. Изучить реализацию DI с помощью Spring Context на основе аннотаций. Изменить код так, чтобы имя студента можно было задать при внедрении бина в классе Department, взамен жестко заданным в самих классах студентов.

Содержание отчета по лабораторной работе

- Титульный лист
- Описание задания
- Код из пунктов 2.1-2.4

Приложение 1

```
public class AnnotationContainer {
    private final Map<String, Object> beans = new HashMap<>();

    public void scanAndRegisterBeans(String basePackage) throws Exception {
        // В реальной реализации здесь должен быть сканер
        // классов в указанном пакете
        // Для примера просто регистрируем классы вручную
        registerBean(MyRepository.class);
        registerBean(MyService.class);
        registerBean(MyController.class);
    }

    public void registerBean(Class<?> clazz) throws Exception {
        if (!clazz.isAnnotationPresent(Component.class)) {
            throw new IllegalArgumentException("Class must be annotated with
@Component");
        }

        String beanName = getBeanName(clazz);
        Object beanInstance = createBeanInstance(clazz);
        beans.put(beanName, beanInstance);
    }

    private String getBeanName(Class<?> clazz) {
        Component component = clazz.getAnnotation(Component.class);
        return component.value().isEmpty() ?
                clazz.getSimpleName() :
                component.value();
    }

    private Object createBeanInstance(Class<?> clazz) throws Exception {
        Constructor<?>[] constructors = clazz.getConstructors();

        for (Constructor<?> constructor : constructors) {
            if (constructor.isAnnotationPresent(Autowired.class)) {
                return createBeanWithAutowiredConstructor(constructor);
            }
        }

        Object instance = clazz.getDeclaredConstructor().newInstance();
        autowireFields(instance);
        return instance;
    }

    private Object createBeanWithAutowiredConstructor(Constructor<?>
constructor) throws Exception {
        Class<?>[] paramTypes = constructor.getParameterTypes();
        Object[] params = new Object[paramTypes.length];

        for (int i = 0; i < paramTypes.length; i++) {
            params[i] = getBeanByType(paramTypes[i]);
        }

        Object instance = constructor.newInstance(params);
        autowireFields(instance);
        return instance;
    }

    private void autowireFields(Object instance) throws Exception {
        for (Field field : instance.getClass().getDeclaredFields()) {
            if (field.isAnnotationPresent(Autowired.class)) {
```

```
                Object dependency = getBeanByType(field.getType());
                field.setAccessible(true);
                field.set(instance, dependency);
            }
        }
    }

    public <T> T getBeanByType(Class<T> type) {
        return beans.values().stream()
                .filter(type::isInstance)
                .map(type::cast)
                .findFirst()
                .orElseThrow(() -> new RuntimeException("Bean of type " +
type + " not found"));
    }

    public <T> T getBeanByName(String name) {
        //return (T) beans.get(name);
        if (!beans.containsKey(name)) {
            throw new NoSuchElementException("Bean with name '" + name + "'
not found");
        }
        return (T) beans.get(name);
    }
}
```