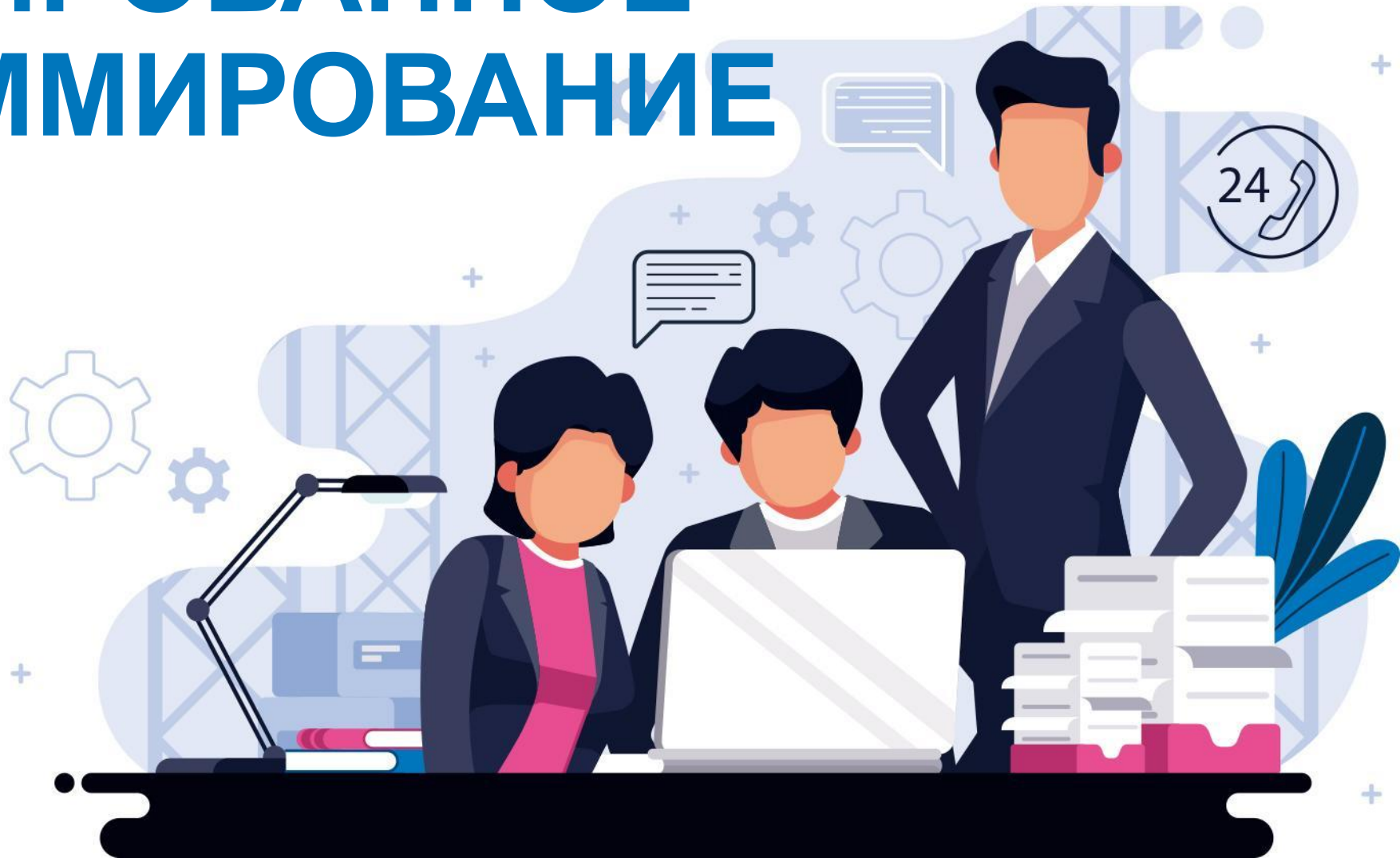
ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Лекция 4



План

- Pointcut
- Комбинирование Pointcut
- Порядок выполнения Aspect-ов

Pointcut

Pointcut – выражение, описывающее, где должен быть применен Advice.

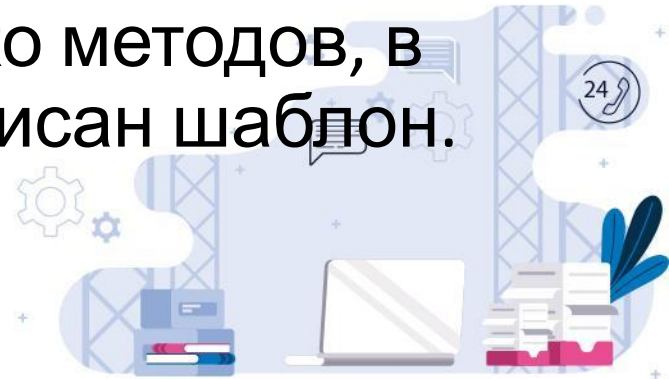
Spring AOP использует AspectJ Pointcut expression language, то есть определенные правила в написании выражений для создания Pointcut.

Pointcut

Для описания Pointcut используется шаблон. Выделены **обязательные** элементы.

execution(modifiers-pattern? **return-type-pattern** declaring-type-pattern? **method-name-pattern(parameters-pattern)** throws-pattern? **)**

Под шаблон может подходить один или несколько методов, в зависимости от того, насколько детально был описан шаблон.



Pointcut

Рассмотрим пример с предыдущей лекции.

```
@Component
@Aspect
public class LoggingAspect {
    @Before("execution(public void getBook())")
    public void beforeGetBookAdvice() { System.out.println("beforeGetBookAdvice: попытка получить книгу"); }
}
```

По шаблону мы указали все, кроме declaring-type-pattern и throws-pattern. Отсутствие declaring-type-pattern означает, что под шаблон подойдет метод getBook() абсолютно любого класса.

Pointcut

Создадим абстрактный класс

```
public abstract class AbstractLibrary {
    abstract public void getBook();
}
```

от которого будет наследоваться класс Library

```
@Component("libraryBean")
public class Library extends AbstractLibrary{
    @Override
    public void getBook() { System.out.println("Мы берем книгу"); }
}
```

Pointcut

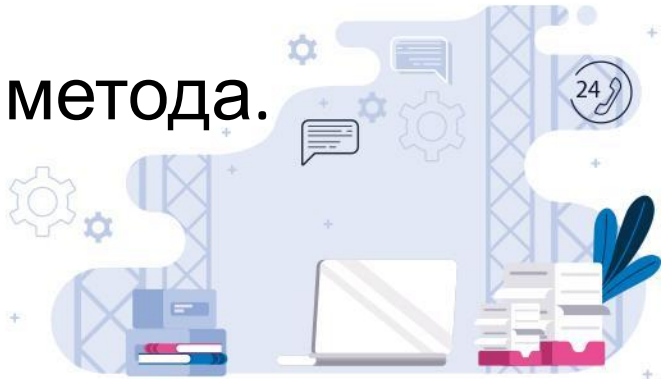
Переименуем класс, чтобы его название конкретизировало его природу. Например, в университетскую библиотеку.

```
@Component
public class UniLibrary extends AbstractLibrary{
    @Override
    public void getBook() { System.out.println("Мы берем книгу из UniLibrary"); }
}
```

Создадим класс школьная библиотека.

```
@Component
public class SchoolLibrary extends AbstractLibrary{
    @Override
    public void getBook() { System.out.println("Мы берем книгу из SchoolLibrary"); }
}
```

Как вы можете видеть, под шаблон подходит оба метода.



Pointcut

Вызовем метод в классе Test1

```
public class Test1 {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(MyConfig.class);

        UniLibrary uniLibrary = context.getBean( name: "uniLibrary", UniLibrary.class);
        uniLibrary.getBook();

        SchoolLibrary schoolLibrary = context.getBean( name: "schoolLibrary", SchoolLibrary.class);
        schoolLibrary.getBook();

        context.close();
    }
}
```

Pointcut

Вывод:

```
21:57:13.235 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'schoolLibrary'
21:57:13.283 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'uniLibrary'
21:57:13.298 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'loggingAspect'
beforeGetBookAdvice: попытка получить книгу
Мы берем книгу из UniLibrary
beforeGetBookAdvice: попытка получить книгу
Мы берем книгу из SchoolLibrary
21:57:13.352 [main] DEBUG org.springframework.context.annotation.AnnotationConfigApplicationContext - Closing org.springframework.context.annotation.Annotat

Process finished with exit code 0
```

Это произошло, поскольку оба метода подходят под Pointcut.

Pointcut

Модифицируем Pointcut, чтобы метод вызвался только для UniLibrary. Для этого нам необходимо указать declaring-type-pattern, а именно полное имя класса UniLibrary.

```
@Component
@Aspect
public class LoggingAspect {
    @Before("execution(public void com.donnu.demo.aop.UniLibrary.getBook())")
    public void beforeGetBookAdvice() { System.out.println("beforeGetBookAdvice: попытка получить книгу"); }
}
```

Вывод:

```
22:03:42.812 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'schoolLibrary'
22:03:42.812 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'uniLibrary'
22:03:42.865 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'loggingAspect'
beforeGetBookAdvice: попытка получить книгу
Мы берем книгу из UniLibrary
Мы берем книгу из SchoolLibrary
```

Pointcut

execution(public void getBook()) – соответствует методу без параметров, где бы он ни находился, с модификатором доступа **public**, возвращаемым типом **void** и названием **getBook()**

execution(public void com.donnu.demo.aop.UniLibrary.getBook()) – соответствует методу без параметров, из класса **UniLibrary**, с модификатором доступа **public**, возвращаемым типом **void** и названием **getBook()**

Pointcut

execution(public void get*()) – соответствует методу без параметров, где бы он ни находился, с модификатором доступа **public**, возвращаемым типом **void** и названием, начинающимся на **get**.

```
@Component
@Aspect
public class LoggingAspect {
    @Before("execution(public void get*())")
    public void beforeGetBookAdvice() { System.out.println("beforeGetBookAdvice: попытка получить книгу"); }
}
```

Pointcut

Добавим метод getMagazine в UniLibrary.

```
@Component
public class UniLibrary extends AbstractLibrary{
    @Override
    public void getBook() { System.out.println("Мы берем книгу из UniLibrary"); }

    public void getMagazine() { System.out.println("Мы берем журнал из UniLibrary"); }
}
```

Pointcut

Добавим вызов метода getMagazine в Test1.

```
public class Test1 {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(MyConfig.class);

        UniLibrary uniLibrary = context.getBean( name: "uniLibrary", UniLibrary.class);
        uniLibrary.getBook();
        uniLibrary.getMagazine();

        SchoolLibrary schoolLibrary = context.getBean( name: "schoolLibrary", SchoolLibrary.class);
        schoolLibrary.getBook();

        context.close();
    }
}
```

Pointcut

Как мы можем видеть аспект был вызван трижды. Для getBook и getMagazine из UniLibrary и getBook из SchoolLibrary. Все эти методы подошли под шаблон.

```
22:15:07.701 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'schoolLibrary'
22:15:07.734 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'uniLibrary'
22:15:07.744 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'loggingAspect'
beforeGetBookAdvice: попытка получить книгу
Мы берем книгу из UniLibrary
beforeGetBookAdvice: попытка получить книгу
Мы берем журнал из UniLibrary
beforeGetBookAdvice: попытка получить книгу
Мы берем книгу из SchoolLibrary
22:15:07.830 [main] DEBUG org.springframework.context.annotation.AnnotationConfigApplicationContext - Closing org.springframework.context.annotation.Annotati

Process finished with exit code 0
```

Pointcut

Приведем пример работы с **return-type-pattern.** Добавим еще один Advice:

```
@Component
@Aspect
public class LoggingAspect {
    @Before("execution(public void get*())")
    public void beforeGetBookAdvice() { System.out.println("beforeGetBookAdvice: попытка получить книгу"); }

    @Before("execution(public void returnBook())")
    public void beforeReturnBookAdvice() { System.out.println("beforeReturnBookAdvice: попытка вернуть книгу"); }
}
```

и метод в UniLibrary:

```
public void returnBook() { System.out.println("Мы возвращаем книгу в UniLibrary"); }
```

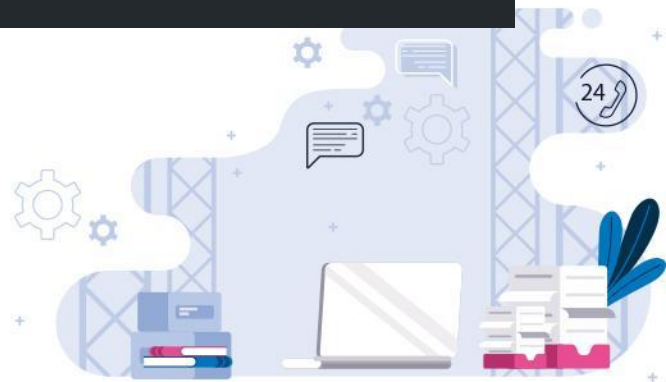
Pointcut

В классе Test1 вызовем метод returnBook:

```
UniLibrary uniLibrary = context.getBean( name: "uniLibrary", UniLibrary.class);
uniLibrary.returnBook();
```

Вывод указывает на то, что Advice работает:

```
22:29:14.506 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'schoolLibrary'
22:29:14.542 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'uniLibrary'
22:29:14.558 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'loggingAspect'
beforeReturnBookAdvice: попытка вернуть книгу
Мы возвращаем книгу в UniLibrary
```



Pointcut

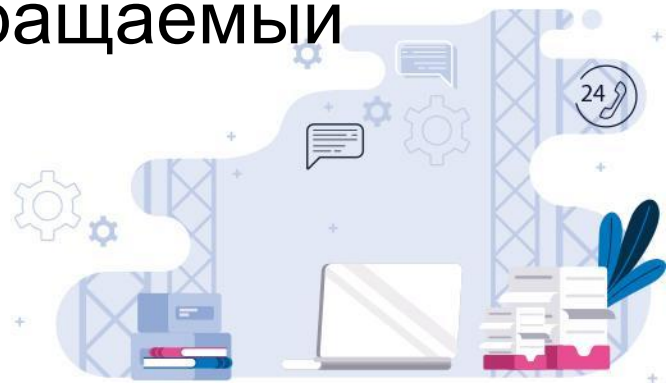
Теперь, если мы изменим возвращаемый тип, например вот

```
public String returnBook() {
    System.out.println("Мы возвращаем книгу в UniLibrary");
    return "OK";
}
```

Метод больше не будет подходить под шаблон, Advice не будет вызван. Если же мы не хотим зависеть от возвращаемого типа, а он является обязательным параметром шаблона, мы можем изменить возвращаемый

```
@Before("execution(public * returnBook())")
public void beforeReturnBookAdvice() { System.out.println("beforeReturnBookAdvice: попытка вернуть книгу"); }
```

```
beforeReturnBookAdvice: попытка вернуть книгу
Мы возвращаем книгу в UniLibrary
```



Pointcut

Если мы хотим аналогичным образом поступить с модификатором доступа, мы можем просто его убрать, оставив только *

```
@Before("execution(* returnBook())")
public void beforeReturnBookAdvice() { System.out.println("beforeReturnBookAdvice: попытка вернуть книгу"); }
```

Как вы помните modifiers-pattern не является обязательным элементом шаблона

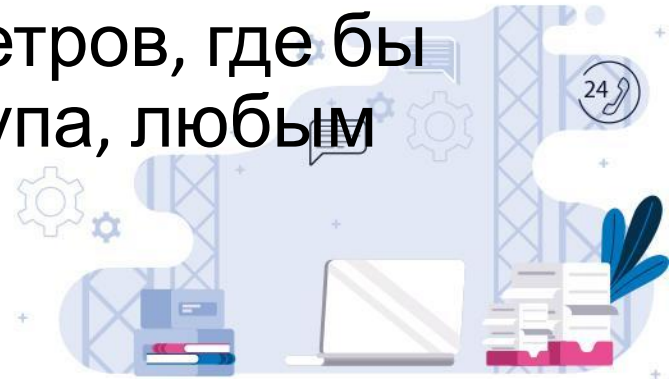
```
22:39:33.536 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'schoolLibrary'
22:39:33.583 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'uniLibrary'
22:39:33.598 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'loggingAspect'
beforeReturnBookAdvice: попытка вернуть книгу
Мы возвращаем книгу в UniLibrary
```

Pointcut

Таким образом:

execution(* returnBook()) - соответствует методу без параметров, где бы он ни находился, с любым модификатором доступа, любым возвращаемым типом и названием **returnBook()**

execution(* *()) - соответствует методу без параметров, где бы он ни находился, с любым модификатором доступа, любым возвращаемым типом и любым называнием



Pointcut

Рассмотрим параметры метода при написании Pointcut. Для этого вернем класс UniLibrary в первоначальное состояние.

```
@Component
public class UniLibrary {
    public void getBook() { System.out.println("Мы берем книгу из UniLibrary"); }

    public void getMagazine() { System.out.println("Мы берем журнал из UniLibrary"); }

    public String returnBook() {...}
}
```

Это необходимо для более простой работы с добавлением параметров.

Pointcut

Добавим параметр в метод getBook

```
public void getBook(String bookName) {
    System.out.println("Мы берем книгу из UniLibrary: " + bookName);
}
```

В Test1 добавим параметр при вызове метода

```
UniLibrary uniLibrary = context.getBean( name: "uniLibrary", UniLibrary.class);
uniLibrary.getBook( bookName: "1984");
```

Убедимся, что при этом наш Advice больше не срабатывает

```
23:19:14.124 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'schoolLibrary'
23:19:14.140 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'uniLibrary'
23:19:14.155 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'loggingAspect'
Мы берем книгу из UniLibrary: 1984
23:19:14.225 [main] DEBUG org.springframework.context.annotation.AnnotationConfigApplicationContext - Closing org.springframework.context.annotation.Annotat
```

Pointcut

Добавим параметр в Advice. Обратите внимание, что указывается только тип параметра, но не его название.

```
@Component
@Aspect
public class LoggingAspect {
    @Before("execution(public void getBook(String))")
    public void beforeGetBookAdvice() { System.out.println("beforeGetBookAdvice: попытка получить книгу"); }
}
```

Вывод:

```
23:21:44.884 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'schoolLibrary'
23:21:44.884 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'uniLibrary'
23:21:44.938 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'loggingAspect'
beforeGetBookAdvice: попытка получить книгу
Мы берем книгу из UniLibrary: 1984
```

Pointcut

execution(public void getBook(String)) - соответствует методу с параметром **String**, где бы он ни находился, с модификатором доступа **public**, возвращаемым типом **void** и названием **getBook()**

Pointcut

Допустим, что мы хотим, чтобы под наш шаблон подходил любой метод, имеющий только 1 параметр. Для этого добавляем параметр в getMagazine:

```
public void getMagazine(int a) { System.out.println("Мы берем журнал из UniLibrary: " + a + " шт"); }
```

А сам шаблон модифицируем следующим образом:

```
@Component
@Aspect
public class LoggingAspect {
    @Before("execution(public void *(*))")
    public void beforeGetBookAdvice() { System.out.println("beforeGetBookAdvice: попытка получить книгу"); }
}
```

Pointcut

Вызовем метод в Test1:

```
public class Test1 {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(MyConfig.class);

        UniLibrary uniLibrary = context.getBean( name: "uniLibrary", UniLibrary.class);
        uniLibrary.getBook( bookName: "1984");
        uniLibrary.getMagazine( a: 8);

        context.close();
    }
}
```

Вывод:

```
beforeGetBookAdvice: попытка получить книгу
Мы берем книгу из UniLibrary: 1984
beforeGetBookAdvice: попытка получить книгу
Мы берем журнал из UniLibrary: 8 шт
```

Pointcut

Если же мы хотим, чтобы под описанный шаблон подходил метод с любым количеством параметров, необходимо *

```
@Component
@Aspect
public class LoggingAspect {
    @Before("execution(public void *(..))")
    public void beforeGetBookAdvice() { System.out.println("beforeGetBookAdvice: попытка получить книгу"); }
}
```

Это будет работать даже в том случае, если параметров 0

```
public void getMagazine() { System.out.println("Мы берем журнал из UniLibrary"); }
```

```
beforeGetBookAdvice: попытка получить книгу
Мы берем книгу из UniLibrary: 1984
beforeGetBookAdvice: попытка получить книгу
Мы берем журнал из UniLibrary
```

Pointcut

execution(public void getBook(String)) - соответствует методу с параметром **String**, где бы он ни находился, с модификатором доступа **public**, возвращаемым типом **void** и названием **getBook()**

execution(public void getBook(*)) - соответствует методу с любым одним параметром, где бы он ни находился, с модификатором доступа **public**, возвращаемым типом **void** и названием **getBook()**

execution(public void getBook(..)) - соответствует методу с любым количеством любого типа параметров, где бы он ни находился, с модификатором доступа **public**, возвращаемым типом **void** и названием **getBook()**

Pointcut

Рассмотрим ситуацию, когда параметр имеет тип созданного нами класса. Создадим класс Book:

```java
package com.donnu.demo.aop;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class Book {
    @Value("1984")
    private String name;

    public String getName() {
        return name;
    }
}
```

Pointcut

В UniLibrary меняем параметр метода getBook на Book.

```
@Component
public class UniLibrary {
    public void getBook(Book book) { System.out.println("Мы берем книгу из UniLibrary: " + book.getName()); }

    public void getMagazine() { System.out.println("Мы берем журнал из UniLibrary"); }

    public String returnBook() {...}
}
```

В Test1 получаем объект класса Book и передаем в метод getBook

```
Book book = context.getBean( name: "book", Book.class);

UniLibrary uniLibrary = context.getBean( name: "uniLibrary", UniLibrary.class);
uniLibrary.getBook(book);
uniLibrary.getMagazine();
```

Pointcut

Но, если в параметре Advice мы напишем просто Book будет ошибка.

```
@Component
@Aspect
public class LoggingAspect {
    @Before("execution(public void getBook(Book))")
    public void beforeGetBookAdvice() { System.                    Advice: попытка получить книгу"); }
}
```

Cannot resolve symbol 'Book'

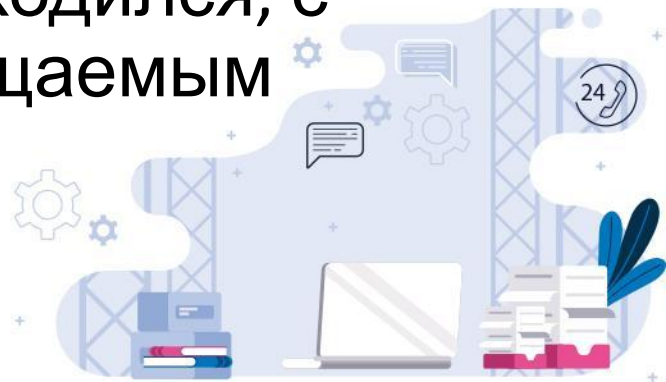
Нашему Pointcut непонятно, о каком типе идет речь. Необходимо указать **полное** имя класса.

```
@Component
@Aspect
public class LoggingAspect {
    @Before("execution(public void getBook(com.donnu.demo.aop.Book))")
    public void beforeGetBookAdvice() { System.out.println("beforeGetBookAdvice: попытка получить книгу"); }
}
```

Pointcut

execution(public void getBook(com.donnu.demo.aop.Book, ..)) - соответствует методу с первым параметром **Book**, и любым количеством других параметров, даже 0, где бы он ни находился, с модификатором доступа **public**, возвращаемым типом **void** и названием **getBook()**

execution(* *(..)) - соответствует методу с любым количеством других параметров любого типа, где бы он ни находился, с любым модификатором доступа, любым возвращаемым типом и любым названием



Pointcut

Вернем UniLibrary и другие элементы в исходное состояние

```java
@Component
public class UniLibrary {
    public void getBook() { System.out.println("Мы берем книгу из UniLibrary"); }

    public void getMagazine() { System.out.println("Мы берем журнал из UniLibrary"); }

    public String returnBook() {...}
}

public class Test1 {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(MyConfig.class);

        UniLibrary uniLibrary = context.getBean( name: "uniLibrary", UniLibrary.class);
        uniLibrary.getBook();
        uniLibrary.getMagazine();

        context.close();
    }
}
```

```java
@Component
@Aspect
public class LoggingAspect {
    @Before("execution(* get*())")
    public void beforeGetBookAdvice() { System.out.println("beforeGetBookAdvice: попытка получить книгу/журнал"); }
}
```

Pointcut

Изменим Aspect, чтобы реализовать в нем и Advice для проверки прав.

```java
@Component
@Aspect
public class LoggingAndSecurityAspect {
    @Before("execution(* get*())")
    public void beforeGetLoggingAdvice() {
        System.out.println("beforeGetLoggingAdvice: попытка получить книгу/журнал");
    }

    @Before("execution(* get*())")
    public void beforeGetSecurityAdvice() {
        System.out.println("beforeGetSecurityAdvice: проверка прав на получение книги/журнала");
    }
}
```

Pointcut

Проверим работу Advice.

```
public class Test1 {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(MyConfig.class);

        UniLibrary uniLibrary = context.getBean( name: "uniLibrary", UniLibrary.class);
        uniLibrary.getBook();
        uniLibrary.getMagazine();

        context.close();
    }
}
```

Логика работы beforeGetSecurityAdvice будет аналогична beforeGetLoggingAdvice:

```
beforeGetLoggingAdvice: попытка получить книгу/журнал
beforeGetSecurityAdvice: проверка прав на получение книги/журнала
Мы берем книгу из UniLibrary
beforeGetLoggingAdvice: попытка получить книгу/журнал
beforeGetSecurityAdvice: проверка прав на получение книги/журнала
Мы берем журнал из UniLibrary
```

Pointcut

Для того, чтобы не пользоваться копированием, когда для нескольких Advice подходит один и тот же Pointcut, есть возможность объявить Pointcut, а потом использовать его несколько раз.

@Pointcut(“pointcut_expression”)
private void pointcut_reference(){}

Использование:

@Before(“**pointcut_reference()**”)
public void advice_name(){ /*code*/}

Pointcut

В нашем примере будет выглядеть следующим образом:

```java
@Component
@Aspect
public class LoggingAndSecurityAspect {
    @Pointcut("execution(* get*())")
    private void allGetMethods(){}

    @Before("allGetMethods()")
    public void beforeGetLoggingAdvice() {
        System.out.println("beforeGetLoggingAdvice: попытка получить книгу/журнал");
    }

    @Before("allGetMethods()")
    public void beforeGetSecurityAdvice() {
        System.out.println("beforeGetSecurityAdvice: проверка прав на получение книги/журнала");
    }
}
```

```
beforeGetLoggingAdvice: попытка получить книгу/журнал
beforeGetSecurityAdvice: проверка прав на получение книги/журнала
Мы берем книгу из UniLibrary
beforeGetLoggingAdvice: попытка получить книгу/журнал
beforeGetSecurityAdvice: проверка прав на получение книги/журнала
Мы берем журнал из UniLibrary
```

Pointcut

Если данный метод будет иметь модификатор доступа public, то мы сможем использовать его в других классах аспектах.

```
@Pointcut(“pointcut_expression”)
public void pointcut_reference(){}
```

Pointcut

Плюсы объявления **Pointcut**:

- Возможность использования одного **Pointcut** для множества Advice
- Возможность быстрого изменения **Pointcut** для множества Advice
- Возможность комбинирования **Pointcut**

Комбинирование Pointcut

Добавим в класс UniLibrary несколько методов

```java
@Component
public class UniLibrary {
    public void getBook() { System.out.println("Мы берем книгу из UniLibrary"); }

    public void getMagazine() { System.out.println("Мы берем журнал из UniLibrary"); }

    public void returnBook() { System.out.println("Мы возвращаем книгу в UniLibrary"); }

    public void returnMagazine() { System.out.println("Мы возвращаем журнал в UniLibrary"); }

    public void addBook() { System.out.println("Мы добавляем книгу в UniLibrary"); }

    public void addMagazine() { System.out.println("Мы добавляем журнал в UniLibrary"); }
}
```

Комбинирование Pointcut

Добавим в LoggingAndSecurityAspect добавим метод логирования получения книг

```
@Pointcut("execution(* com.donnu.demo.aop.UniLibrary.get*())")
private void allGetMethodsFromUniLibrary(){}

@Before("allGetMethodsFromUniLibrary()")
public void beforeGetLoggingAdvice(){
    System.out.println("beforeGetLoggingAdvice: writing Log#1");
}
```

Комбинирование Pointcut

Аналогично для return-методов. Но, что если у нас есть сквозная логика, которая должна выполняться в обоих случаях?

```java
@Component
@Aspect
public class LoggingAndSecurityAspect {

    @Pointcut("execution(* com.donnu.demo.aop.UniLibrary.get*())")
    private void allGetMethodsFromUniLibrary(){}

    @Pointcut("execution(* com.donnu.demo.aop.UniLibrary.return*())")
    private void allReturnMethodsFromUniLibrary(){}

    @Before("allGetMethodsFromUniLibrary()")
    public void beforeGetLoggingAdvice() { System.out.println("beforeGetLoggingAdvice: writing Log#1"); }

    @Before("allReturnMethodsFromUniLibrary()")
    public void beforeReturnLoggingAdvice() { System.out.println("beforeReturnLoggingAdvice: writing Log#2"); }

// ...
}
```

Комбинирование Pointcut

Комбинирование Pointcut-ов – это их объединение с помощью логических операторов **&& || !**

Комбинирование Pointcut

```java
@Component
@Aspect
public class LoggingAndSecurityAspect {

    @Pointcut("execution(* com.donnu.demo.aop.UniLibrary.get*())")
    private void allGetMethodsFromUniLibrary(){}

    @Pointcut("execution(* com.donnu.demo.aop.UniLibrary.return*())")
    private void allReturnMethodsFromUniLibrary(){}

    @Pointcut("allGetMethodsFromUniLibrary() || allReturnMethodsFromUniLibrary()")
    private void allGetAndReturnMethodsFromUniLibrary(){}

    @Before("allGetMethodsFromUniLibrary()")
    public void beforeGetLoggingAdvice() { System.out.println("beforeGetLoggingAdvice: writing Log#1"); }

    @Before("allReturnMethodsFromUniLibrary()")
    public void beforeReturnLoggingAdvice() { System.out.println("beforeReturnLoggingAdvice: writing Log#2"); }

    @Before("allGetAndReturnMethodsFromUniLibrary()")
    public void beforeGetAndReturnLoggingAdvice(){
        System.out.println("beforeGetAndReturnLoggingAdvice: writing Log#3");
    }

// ...
}
```

Комбинирование Pointcut

Запустим Test1

```
public class Test1 {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(MyConfig.class);

        UniLibrary uniLibrary = context.getBean( name: "uniLibrary", UniLibrary.class);
        uniLibrary.getBook();
        uniLibrary.getMagazine();
        uniLibrary.returnBook();
        uniLibrary.returnMagazine();

        context.close();
    }
}
```

```
beforeGetAndReturnLoggingAdvice: writing Log#3
beforeGetLoggingAdvice: writing Log#1
Мы берем книгу из UniLibrary
beforeGetAndReturnLoggingAdvice: writing Log#3
beforeGetLoggingAdvice: writing Log#1
Мы берем журнал из UniLibrary
beforeGetAndReturnLoggingAdvice: writing Log#3
beforeReturnLoggingAdvice: writing Log#2
Мы возвращаем книгу в UniLibrary
beforeGetAndReturnLoggingAdvice: writing Log#3
beforeReturnLoggingAdvice: writing Log#2
Мы возвращаем журнал в UniLibrary
```

Комбинирование Pointcut

Рассмотрим ситуацию, когда мы хотим вызвать Advice для всех методов кроме одного.

```java
@Component
@Aspect
public class LoggingAndSecurityAspect {

    @Pointcut("execution(* com.donnu.demo.aop.UniLibrary.*())")
    private void allMethodsFromUniLibrary(){}

    @Pointcut("execution(* com.donnu.demo.aop.UniLibrary.returnMagazine())")
    private void returnMagazineFromUniLibrary(){}

    @Pointcut("allMethodsFromUniLibrary() && !returnMagazineFromUniLibrary()")
    private void allMethodsExceptReturnMagazineFromUniLibrary(){}

    @Before("allMethodsExceptReturnMagazineFromUniLibrary()")
    public void beforeAllMethodsExceptReturnMagazineAdvice(){
        System.out.println("beforeAllMethodsExceptReturnMagazineAdvice: writing Log#10");
    }

// ...
}
```

Комбинирование Pointcut

Вызываем Test1

```java
public class Test1 {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(MyConfig.class);

        UniLibrary uniLibrary = context.getBean( name: "uniLibrary", UniLibrary.class);
        uniLibrary.getBook();
        uniLibrary.getMagazine();
        uniLibrary.returnBook();
        uniLibrary.returnMagazine();
        uniLibrary.addBook();
        uniLibrary.addMagazine();

        context.close();
    }
}
```

```
beforeAllMethodsExceptReturnMagazineAdvice: writing Log#10
Мы берем книгу из UniLibrary
beforeAllMethodsExceptReturnMagazineAdvice: writing Log#10
Мы берем журнал из UniLibrary
beforeAllMethodsExceptReturnMagazineAdvice: writing Log#10
Мы возвращаем книгу в UniLibrary
Мы возвращаем журнал в UniLibrary
beforeAllMethodsExceptReturnMagazineAdvice: writing Log#10
Мы добавляем книгу в UniLibrary
beforeAllMethodsExceptReturnMagazineAdvice: writing Log#10
Мы добавляем журнал в UniLibrary
```

Порядок выполнения Aspect-ов

Рассмотрим ранее написанный пример.

```java
@Component
@Aspect
public class LoggingAndSecurityAspect {

// ...

    @Pointcut("execution(* get*())")
    private void allGetMethods(){}

    @Before("allGetMethods()")
    public void beforeGetLoggingAdvice() {
        System.out.println("beforeGetLoggingAdvice: попытка получить книгу/журнал");
    }

    @Before("allGetMethods()")
    public void beforeGetSecurityAdvice() {
        System.out.println("beforeGetSecurityAdvice: проверка прав на получение книги/журнала");
    }
}
```

Порядок выполнения Aspect-ов

В Test1 вызовем методы с get.

```java
public class Test1 {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(MyConfig.class);

        UniLibrary uniLibrary = context.getBean( name: "uniLibrary", UniLibrary.class);
        uniLibrary.getBook();
        uniLibrary.getMagazine();

        context.close();
    }
}
```

Вывод:

```
beforeGetLoggingAdvice: попытка получить книгу/журнал
beforeGetSecurityAdvice: проверка прав на получение книги/журнала
Мы берем книгу из UniLibrary
beforeGetLoggingAdvice: попытка получить книгу/журнал
beforeGetSecurityAdvice: проверка прав на получение книги/журнала
Мы берем журнал из UniLibrary
```

Порядок выполнения Aspect-ов

Каким образом мы можем контролировать порядок выполнения?

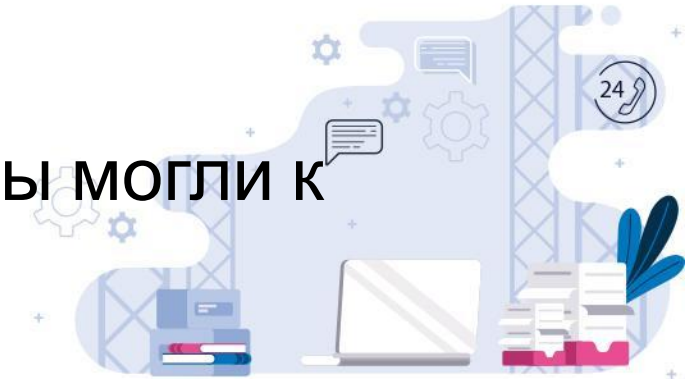
Для этого нам потребуется вынести методы в разные аспекты. Но начнем с того, что вынесем Pointcut в отдельный класс.

```java
package com.donnu.demo.aop.aspects;

import org.aspectj.lang.annotation.Pointcut;

public class MyPointcuts {
    @Pointcut("execution(* get*())")
    public void allGetMethods(){}
}
```

Установим модификатор доступа public, чтобы мы могли к нему обратиться из другого класса.



Порядок выполнения Aspect-ов

Теперь создадим два аспекта. Обратите внимание, что для того, чтобы получить Pointcut необходимо указать полное имя

```java
@Component
@Aspect
public class SecurityAspect {
    @Before("com.donnu.demo.aop.aspects.MyPointcuts.allGetMethods()")
    public void beforeGetSecurityAdvice() {
        System.out.println("beforeGetSecurityAdvice: проверка прав на получение книги/журнала");
    }
}
```

```java
@Component
@Aspect
public class LoggingAspect {
// ...

    @Before("com.donnu.demo.aop.aspects.MyPointcuts.allGetMethods()")
    public void beforeGetLoggingAdvice() {
        System.out.println("beforeGetLoggingAdvice: попытка получить книгу/журнал");
    }
}
```

Порядок выполнения Aspect-ов

Создадим еще один аспект.

```
@Component
@Aspect
public class ExceptionHandlingAspect {
    @Before("com.donnu.demo.aop.aspects.MyPointcuts.allGetMethods()")
    public void beforeGetExceptionHandlingAdvice() {
        System.out.println("beforeGetLoggingAdvice: ловим/обрабатываем ошибки");
    }
}
```

Теперь у нас три аспект-класса и три Advice направленных на get-метод.

Порядок выполнения Aspect-ов

Теперь мы можем указать им порядок, с помощью аннотации @Order

```java
@Component
@Aspect
@Order(1)
public class LoggingAspect {
// ...

    @Before("com.donnu.demo.aop.aspects.MyPointcuts.allGetMethods()")
    public void beforeGetLoggingAdvice() {
        System.out.println("beforeGetLoggingAdvice: попытка получить книгу/журнал");
    }
}
```

```java
@Component
@Aspect
@Order(2)
public class SecurityAspect {
    @Before("com.donnu.demo.aop.aspects.MyPointcuts.allGetMethods()")
    public void beforeGetSecurityAdvice() {
        System.out.println("beforeGetSecurityAdvice: проверка прав на получение книги/журнала");
    }
}
```

```java
@Component
@Aspect
@Order(3)
public class ExceptionHandlingAspect {
    @Before("com.donnu.demo.aop.aspects.MyPointcuts.allGetMethods()")
    public void beforeGetExceptionHandlingAdvice() {
        System.out.println("beforeGetLoggingAdvice: ловим/обрабатываем ошибки");
    }
}
```

Порядок выполнения Aspect-ов

Вывод до аннотации @Order:

```
beforeGetLoggingAdvice: ловим/обрабатываем ошибки
beforeGetLoggingAdvice: попытка получить книгу/журнал
beforeGetSecurityAdvice: проверка прав на получение книги/журнала
Мы берем книгу из UniLibrary
beforeGetLoggingAdvice: ловим/обрабатываем ошибки
beforeGetLoggingAdvice: попытка получить книгу/журнал
beforeGetSecurityAdvice: проверка прав на получение книги/журнала
Мы берем журнал из UniLibrary
```

Вывод после:

```
beforeGetLoggingAdvice: попытка получить книгу/журнал
beforeGetSecurityAdvice: проверка прав на получение книги/журнала
beforeGetLoggingAdvice: ловим/обрабатываем ошибки
Мы берем книгу из UniLibrary
beforeGetLoggingAdvice: попытка получить книгу/журнал
beforeGetSecurityAdvice: проверка прав на получение книги/журнала
beforeGetLoggingAdvice: ловим/обрабатываем ошибки
Мы берем журнал из UniLibrary
```

Порядок выполнения Aspect-ов

Если при вызове одного метода с бизнес-логикой срабатывает несколько Advice, то нет никакой гарантии, что они выполнятся в желаемом порядке.

Для соблюдении порядка такие Advice необходимо распределить по отдельным, упорядоченным аспектам.

Аннотация @Order(1) упорядочивает аспекты. Чем меньше число, тем выше приоритет. Число должно быть целым, отрицательное допустимо.

Порядок выполнения Aspect-ов

Иногда вы заранее не знаете, сколько будет аспектов. Тогда Order делают 10, 20, 30 или 100, 200, 300. Чтобы в случае добавления аспектов не менять Order в уже созданных.

Если для двух аспектов написать одинаковое значение – результат будет непредсказуем.