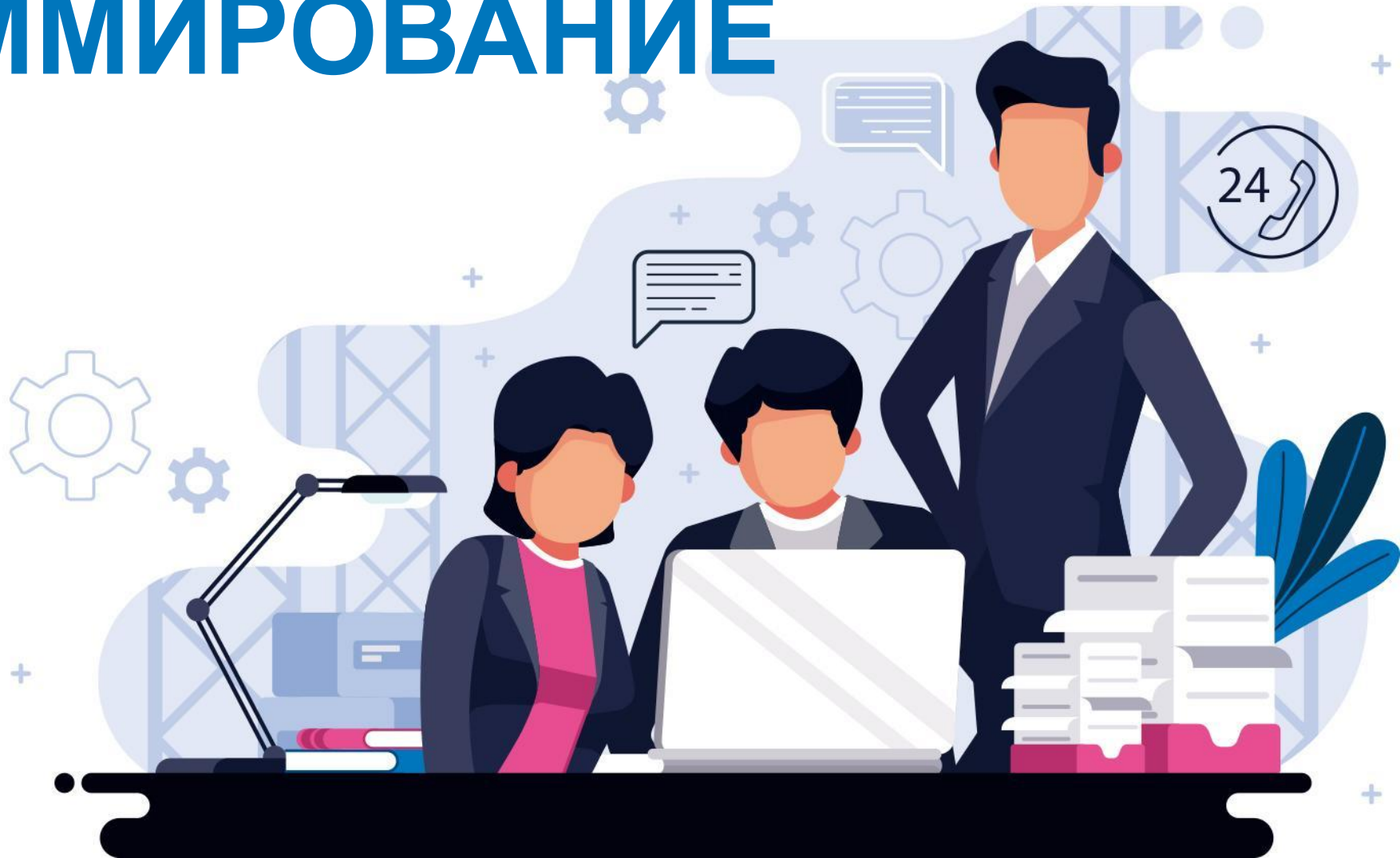
ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Лекция 5



План

- Join Point
- Advice @AfterReturning
- Advice @AfterThrowing
- Advice @After
- Advice @Around

Join Point

Join Point – это точка/момент в программе, когда следует применять Advice. Т.е. точка переплетения метода с бизнес-логикой и метода со служебным функционалом.

Join Point

Для рассмотрения примера добавим несколько параметров и методов в класс Book, а в UniLibrary модифицируем вывод.

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

    @Value("Джордж Оруэлл")
    private String author;

    @Value("1949")
    private int year;

    public String getName() { return name; }

    public String getAuthor() { return author; }

    public int getYear() { return year; }
}
```

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

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

Join Point

Метод addBook теперь будет принимать параметр.

```
public void addBook(String personName, Book book) {
    System.out.println("Мы добавляем книгу в UniLibrary");
    System.out.println("--------------------------------");
}
```

В MyPointcuts поменяем get на add, чтобы работать с add-методами, в частности addBook, а также добавим вывод разделительных линий в Advice.

```
public class MyPointcuts {
    @Pointcut("execution(* add*(..))")
    public void allAddMethods(){}
}
```

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

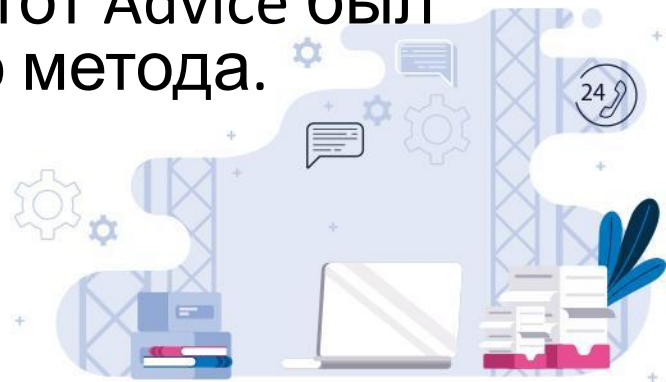
Join Point

Для чего же нужен **Join Point**?

Прописав **Join Point** в параметре метода **Advice**, мы получаем доступ к информации о сигнатуре и параметрах метода с бизнес-логикой.

Наверняка у многих при написании метода связанного с логированием, возникал вопрос, как же в лог мы будем писать информацию о самом методе, благодаря которому этот Advice был вызван, как мы будем использовать параметры этого метода.

Для этого мы используем **Join Point**.



Join Point

Рассмотрим на примере. Вызовем метод addBook в Test1.

```
UniLibrary uniLibrary = context.getBean( name: "uniLibrary", UniLibrary.class);
Book book = context.getBean( name: "book", Book.class);
uniLibrary.addBook( personName: "Admin", book);
```

В LoggingAspect рассмотрим, что мы можем получить из

```
@Before("com.donnu.demo.aop.aspects.MyPointcuts.allAddMethods()")
public void beforeAddLoggingAdvice(JoinPoint joinPoint) {

    MethodSignature methodSignature = (MethodSignature) joinPoint.getSignature();
    // Выводим сигнатуру
    System.out.println("methodSignature = " + methodSignature);
    // Выводим сам метод
    System.out.println("method = " + methodSignature.getMethod());
    // Выводим возвращаемый тип
    System.out.println("return type = " + methodSignature.getReturnType());
    // Выводим имя метода
    System.out.println("method name = " + methodSignature.getName());

    System.out.println("beforeAddLoggingAdvice: попытка получить книгу/журнал");
    System.out.println("--------------------------------");
}
```

Join Point

Вывод:

```
methodSignature = void com.donnu.demo.aop.UniLibrary.addBook(String,Book)
method = public void com.donnu.demo.aop.UniLibrary.addBook(java.lang.String,com.donnu.demo.aop.Book)
return type = void
method name = addBook
beforeAddLoggingAdvice: попытка получить книгу/журнал
--------------------------------
beforeAddSecurityAdvice: проверка прав на получение книги/журнала
--------------------------------
beforeAddLoggingAdvice: ловим/обрабатываем ошибки
--------------------------------
Мы добавляем книгу в UniLibrary
--------------------------------
```

Join Point

Вывод для addMagazine

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

        UniLibrary uniLibrary = context.getBean( name: "uniLibrary", UniLibrary.class);
        Book book = context.getBean( name: "book", Book.class);
        uniLibrary.addBook( personName: "Admin", book);
        uniLibrary.addMagazine();

        context.close();
    }
}
```

```
-------------------------------
methodSignature = void com.donnu.demo.aop.UniLibrary.addMagazine()
method = public void com.donnu.demo.aop.UniLibrary.addMagazine()
return type = void
method name = addMagazine
beforeAddLoggingAdvice: попытка получить книгу/журнал
-------------------------------
```

Join Point

Рассмотрим работу с параметрами. Поскольку у addMagazine нет параметров, он нам не интересен, мы хотим получить параметры метода addBook. Это можно сделать простым

```
MethodSignature methodSignature = (MethodSignature) joinPoint.getSignature();
// Выводим сигнатуру
System.out.println("methodSignature = " + methodSignature);
// Выводим сам метод
System.out.println("method = " + methodSignature.getMethod());
// Выводим возвращаемый тип
System.out.println("return type = " + methodSignature.getReturnType());
// Выводим имя метода
System.out.println("method name = " + methodSignature.getName());

if (methodSignature.getName().equals("addBook")) {
    // работа с методом addBook
}
```

Join Point

Для работы с параметрами мы можем применить метод **getArgs()**, который вернет нам **Object[]**

```
if (methodSignature.getName().equals("addBook")) {
    Object[] arguments = joinPoint.getArgs();

    for (Object obj:arguments) {

        if (obj instanceof Book) {
            Book myBook = (Book) obj;

            System.out.println("Информация о книге:\nназвание: " + myBook.getName() +
                               "\nавтор: " + myBook.getAuthor() +
                               "\nгод выпуска: " + myBook.getYear());
        } else if (obj instanceof String) {
            System.out.println("Книгу принес " + obj);
        }

    }
}
```

```
Книгу принес Admin
Информация о книге:
название: 1984
автор: Джордж Оруэлл
год выпуска: 1949
```

Join Point

Полный вывод:

```
methodSignature = void com.donnu.demo.aop.UniLibrary.addBook(String,Book)
method = public void com.donnu.demo.aop.UniLibrary.addBook(java.lang.String,com.donnu.demo.aop.Book)
return type = void
method name = addBook
Книгу принес Admin
Информация о книге:
название: 1984
автор: Джордж Оруэлл
год выпуска: 1949
beforeAddLoggingAdvice: попытка получить книгу/журнал
```

Advice типы

Типы Advice:

- **Before** – выполняется до метода с основной логикой
- **After returning** – выполняется только после того, как метод корректно отработал
- **After throwing** – выполняется после метода с основной логикой, если было выброшено исключение
- **After / After finally** – выполняется после метода с основной логикой (в любом случае)
- **Around** – выполняется и до, и после

Advice @AfterReturning

@AfterReturning Advice выполняется в том случае, если при работе метода с бизнес-логикой не возникло исключений.

Также методы с основной логикой можно назвать target-методами.

Advice @AfterReturning

Создадим класс Student, добавим геттеры, сеттеры и переопределим метод toString.

```java
public class Student {
    private String nameSurname;
    private int course;
    private double averageGrade;

    public Student(String nameSurname, int course, double averageGrade) {
        this.nameSurname = nameSurname;
        this.course = course;
        this.averageGrade = averageGrade;
    }

    public String getNameSurname() { return nameSurname; }

    public void setNameSurname(String nameSurname) { this.nameSurname = nameSurname; }

    public int getCourse() { return course; }

    public void setCourse(int course) { this.course = course; }

    public double getAverageGrade() { return averageGrade; }

    public void setAverageGrade(double averageGrade) { this.averageGrade = averageGrade; }

    @Override
    public String toString() {
        return "Student{" +
                "nameSurname='" + nameSurname + '\'' +
                ", course=" + course +
                ", averageGrade=" + averageGrade +
                '}';
    }
}
```

Advice @AfterReturning

Создадим класс University и определим его как компонент.

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

import java.util.ArrayList;
import java.util.List;

@Component
public class University {
    private List<Student> students = new ArrayList<>();

    public void addStudents() {
        Student st1 = new Student( nameSurname: "Ivan Ivanov",  course: 4,  averageGrade: 85.6);
        Student st2 = new Student( nameSurname: "Petr Petrov",  course: 4,  averageGrade: 65.7);
        Student st3 = new Student( nameSurname: "Ada Lovelace",  course: 4,  averageGrade: 99.9);

        students.add(st1);
        students.add(st2);
        students.add(st3);
    }

    public List<Student> getStudents() {
        System.out.println("getStudents:");
        System.out.println(students);
        return students;
    }
}
```

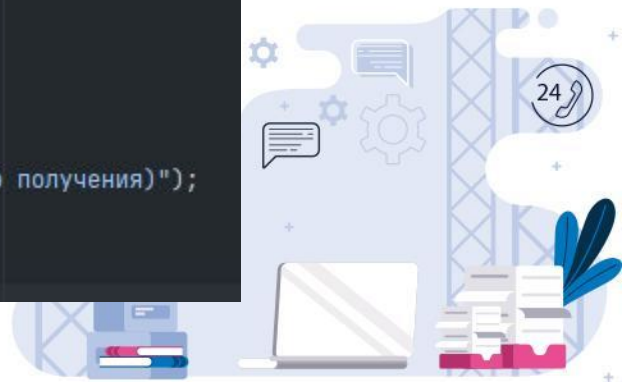
Advice @AfterReturning

Создадим Advice который отработает до метода и который отработает после успешного завершения.

```
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class UniversityLoggingAspect {
    @Before("execution(* getStudents())")
    public void beforeGetStudentsLoggingAdvice() {
        System.out.println("beforeGetStudentsLoggingAdvice: логируем получение списка студентов (перед получением)");
    }

    @AfterReturning("execution(* getStudents())")
    public void afterReturningGetStudentsLoggingAdvice() {
        System.out.println("afterReturningGetStudentsLoggingAdvice: логируем получение списка студентов (после успешного получения)");
    }
}
```



Advice @AfterReturning

Для проверки работоспособности создаем класс Test2

```java
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import java.util.List;

public class Test2 {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(MyConfig.class);

        University university = context.getBean( name: "university", University.class);
        university.addStudents();

        List<Student> students = university.getStudents();
        System.out.println(students);

        context.close();
    }
}
```

Advice @AfterReturning

Вывод:

```
beforeGetStudentsLoggingAdvice: логируем получение списка студентов (перед получением)
getStudents:
[Student{nameSurname='Ivan Ivanov', course=4, averageGrade=85.6}, Student{nameSurname='Petr Petrov', course=4, averageGrade=65.7}, Student{nameSurname='Ada Lovelace', course=4, averageGrade=99.9}]
afterReturningGetStudentsLoggingAdvice: логируем получение списка студентов (после успешного получения)
[Student{nameSurname='Ivan Ivanov', course=4, averageGrade=85.6}, Student{nameSurname='Petr Petrov', course=4, averageGrade=65.7}, Student{nameSurname='Ada Lovelace', course=4, averageGrade=99.9}]
```

Advice @AfterReturning

Важный момент!

Поскольку @AfterReturning отрабатывает после метода, в нем мы можем перехватить результат работы метода, залогировать, и даже **модифицировать** его.

Advice @AfterReturning

Для этого необходимо указать дополнительный параметр returning. Имя переданное в returning должно совпадать с именем параметра переданного в метод.

```
@AfterReturning(pointcut = "execution(* getStudents())",
                returning = "students")
public void afterReturningGetStudentsLoggingAdvice(List<Student> students) {

    Student firstStudent = students.get(0);
    String nameSurname = firstStudent.getNameSurname();
    nameSurname = "Mr. " + nameSurname;
    firstStudent.setNameSurname(nameSurname);

    double avgGrade = firstStudent.getAverageGrade();
    avgGrade = avgGrade+10;
    firstStudent.setAverageGrade(avgGrade);

    System.out.println("afterReturningGetStudentsLoggingAdvice: логируем получение списка студентов (после успешного получения)");
}
```

Advice @AfterReturning

Вывод после модификации:

```
beforeGetStudentsLoggingAdvice: логируем получение списка студентов (перед получением)
getStudents:
[Student{nameSurname='Ivan Ivanov', course=4, averageGrade=85.6}, Student{nameSurname='Petr Petrov', course=4, averageGrade=65.7}, Student{nameSurname='Ada Lovelace', course=4, averageGrade=99.9}]
afterReturningGetStudentsLoggingAdvice: логируем получение списка студентов (после успешного получения)
[Student{nameSurname='Mr. Ivan Ivanov', course=4, averageGrade=95.6}, Student{nameSurname='Petr Petrov', course=4, averageGrade=65.7}, Student{nameSurname='Ada Lovelace', course=4, averageGrade=99.9}]
```

Advice @AfterReturning

Вывод:

@AfterReturning Advice выполняется после метода с бизнес-логикой, если не было брошено исключений. Он выполняется **до присвоения** результата метода переменной. Поэтому с помощью @AfterReturning Advice можно изменить результат работы метода. Злоупотреблять этим не стоит.

Если есть необходимость, можно указать Join Point:

```
@AfterReturning(pointcut = "execution(* getStudents())",
                returning = "students")
public void afterReturningGetStudentsLoggingAdvice(JoinPoint joinPoint, List<Student> students) {
```

Advice @AfterThrowing

Advice **@AfterThrowing** выполняется после метода с бизнес-логикой, если в процессе работы методы было брошено исключение.

*если бросается исключение – сразу наступает окончание работы метода.

Advice @AfterThrowing

Изменим метод getStudents таким образом, чтобы в процессе работы было брошено исключение.

```
public List<Student> getStudents() {
    System.out.println("Начало работы метода getStudents");
    System.out.println(students.get(3));
    System.out.println("getStudents:");
    System.out.println(students);
    return students;
}
```

*в students всего 3 элементы с индексами 0, 1 и 2.

Advice @AfterThrowing

Создадим Advice afterThrowingGetStudentsLoggingAdvice

```
@AfterThrowing("execution(* getStudents())")
public void afterThrowingGetStudentsLoggingAdvice() {
    System.out.println("afterThrowingGetStudentsLoggingAdvice: логируем исключение");
}
```

И запускаем Test2 без изменений. Получаем ошибку:

```
beforeGetStudentsLoggingAdvice: логируем получение списка студентов (перед получением)
Начало работы метода getStudents
afterThrowingGetStudentsLoggingAdvice: логируем исключение
Exception in thread "main" java.lang.IndexOutOfBoundsException Create breakpoint : Index 3 out of bounds for length 3 <3 internal lines>
```

Advice @AfterThrowing

Обратите внимание, что advice отработал до того как исключение было выведено в консоль, соответственно до того, как оно попало в main-метод и выполнение программы прекратилось.

Чтобы это предотвратить мы можем использовать try:

```
try {
    List<Student> students = university.getStudents();
    System.out.println(students);
} catch (IndexOutOfBoundsException e) {
    System.out.println("IndexOutOfBoundsException: было поймано исключение  " + e);
} catch (Exception e) {
    System.out.println("Exception: было поймано исключение " + e);
}
```

Advice @AfterThrowing

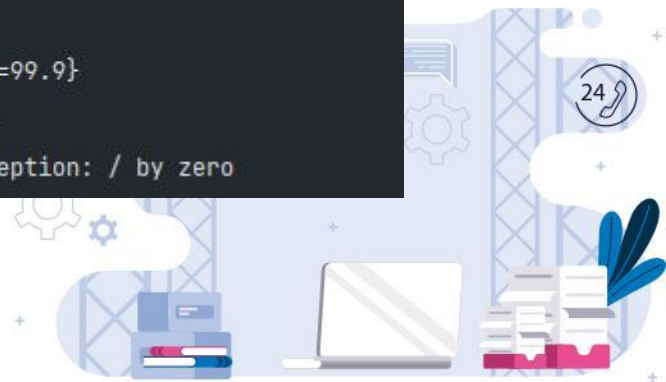
Таким образом мы можем поймать не только ожидаемое исключение:

```
beforeGetStudentsLoggingAdvice: логируем получение списка студентов (перед получением)
Начало работы метода getStudents
afterThrowingGetStudentsLoggingAdvice: логируем исключение
IndexOutOfBoundsException: было поймано исключение  java.lang.IndexOutOfBoundsException: Index 3 out of bounds for length 3
```

но и любое другое:

```
public List<Student> getStudents() {
    System.out.println("Начало работы метода getStudents");
    System.out.println(students.get(2));
    System.out.println(8/0);
    System.out.println("getStudents:");
    System.out.println(students);
    return students;
}
```

```
beforeGetStudentsLoggingAdvice: логируем получение списка студентов (перед получением)
Начало работы метода getStudents
Student{nameSurname='Ada Lovelace', course=4, averageGrade=99.9}
afterThrowingGetStudentsLoggingAdvice: логируем исключение
Exception: было поймано исключение java.lang.ArithmeticException: / by zero
```



Advice @AfterThrowing

Чтобы получить само исключение в advice необходимо:

```
@AfterThrowing(pointcut = "execution(* getStudents())",
               throwing = "exception")
public void afterThrowingGetStudentsLoggingAdvice(Throwable exception) {
    System.out.println("afterThrowingGetStudentsLoggingAdvice: логируем исключение " + exception);
}
```

Вывод:

```
beforeGetStudentsLoggingAdvice: логируем получение списка студентов (перед получением)
Начало работы метода getStudents
afterThrowingGetStudentsLoggingAdvice: логируем исключение java.lang.IndexOutOfBoundsException: Index 3 out of bounds for length 3
IndexOutOfBoundsException: было поймано исключение  java.lang.IndexOutOfBoundsException: Index 3 out of bounds for length 3
```

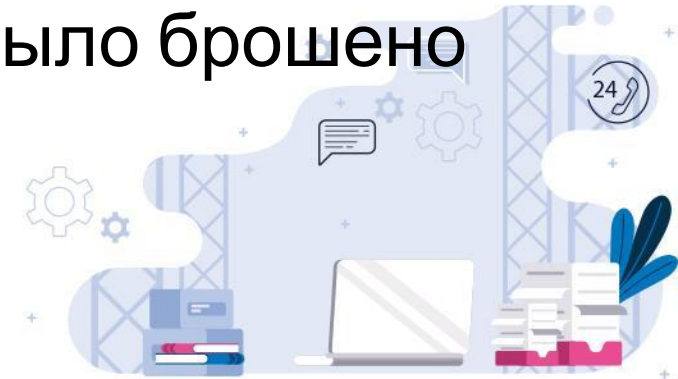
Advice @AfterThrowing

Важно!

Мы не можем остановить или обработать исключение в **Advice @AfterThrowing**, чтобы оно не попало в main.

Таким образом:

Advice @AfterThrowing не влияет на протекание программы при выбрасывании исключений. С помощью @AfterThrowing можно получить доступ к исключению, которое было брошено из метода с бизнес-логикой.



Advice @After

Advice @After – выполняется после метода с бизнес логикой, независимо от того, как метод отработал.

```
@After("execution(* getStudents())")
public void afterGetStudentsLoggingAdvice() {
    System.out.println("afterGetStudentsLoggingAdvice: логируем окончание работы метода");
}
```

Вывод:

```
Начало работы метода getStudents
afterGetStudentsLoggingAdvice: логируем окончание работы метода
IndexOutOfBoundsException: было поймано исключение  java.lang.IndexOutOfBoundsException: Index 3 out of bounds for length 3
```

Advice @After

С помощью Advice @After невозможно:

1. получить доступ к исключению, которое было брошено методом с бизнес-логикой;
2. получить доступ к возвращаемому результату.

В Advice @After можно использовать Join Point.

Advice @Around

Advice @Around выполняется до и после метода с бизнес логикой.

С помощью Advice @Around можно:

1. произвести действие до работы target-метода;
2. произвести действие после работы target-метода;
3. получить результат работы target-метода (изменить его);
4. предпринять действия, если target-метод выбросил исключение.

Advice @Around

Вернемся в класс UniLibrary и изменим метод returnBook

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

Создадим новый Aspect

```
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class NewLoggingAspect {
    @Around("execution(public String returnBook())")
    public void aroundReturnBookLoggingAdvice() {
        System.out.println("aroundReturnBookLoggingAdvice: в библиотеку возвращают книгу");
    }
}
```

Advice @Around

Создадим Test3 для вызова метода

```
public class Test3 {
    public static void main(String[] args) {
        System.out.println("Метод main starts");
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(MyConfig.class);

        UniLibrary uniLibrary = context.getBean( name: "uniLibrary", UniLibrary.class);
        String bookName = uniLibrary.returnBook();
        System.out.println("В библиотеку вернули книгу " + bookName);

        context.close();
        System.out.println("Метод main ends");
    }
}
```

Advice @Around

Выводим результат:

```
"C:\Program Files\BellSoft\LibericaJDK-11\bin\java.exe" ...
Метод main starts
```

```
aroundReturnBookLoggingAdvice: в библиотеку возвращают книгу
В библиотеку вернули книгу null
05:20:03.193 [main] DEBUG org.springframework.context.annotation.
Метод main ends
```

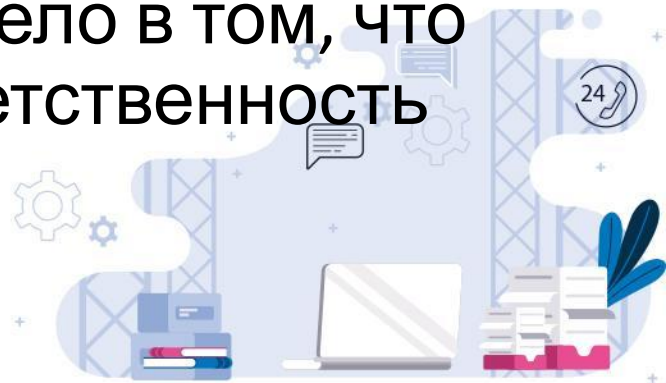
Почему вывелся null и sout в самом методе не сработал?

Advice @Around

Это связано с работой **Advice @Around**. Он работает до и после метода, но не таким образом, как изначально могло показаться. Можно подумать, что он просто дважды выполнит содержимое своего метода:

```
@Around("execution(public String returnBook())")
public void aroundReturnBookLoggingAdvice() {
    System.out.println("aroundReturnBookLoggingAdvice: в библиотеку возвращают книгу");
}
```

На самом деле **Advice @Around** работает не так. Обратите внимание на вывод, target-метод не отработал. Дело в том, что используя **Advice @Around** мы берем на себя ответственность **самим запускать target-метод**.



Advice @Around

В качестве параметра метод принимает ProceedingJoinPoint.
Это связь с target-методом.

```
@Around("execution(public String returnBook())")
public Object aroundReturnBookLoggingAdvice(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
    // Первая часть логики
    System.out.println("aroundReturnBookLoggingAdvice: в библиотеку пытаются вернуть книгу");
    // Вызов метода
    Object targetMethodResult = proceedingJoinPoint.proceed();
    // Вторая часть логики
    System.out.println("aroundReturnBookLoggingAdvice: в библиотеку успешно вернули книгу");
    // Возвращаем результат
    return targetMethodResult;
}
```

Вывод:

```
aroundReturnBookLoggingAdvice: в библиотеку пытаются вернуть книгу
Мы возвращаем книгу в UniLibrary
aroundReturnBookLoggingAdvice: в библиотеку успешно вернули книгу
В библиотеку вернули книгу Война и мир
```

Advice @Around

Стоит отметить, что с помощью ProceedingJoinPoint можно получить доступ к сигнатуре и параметрам метода, как и с помощью Join Point.

Advice @Around

Засечем время работы метода returnBook

```
@Around("execution(public String returnBook())")
public Object aroundReturnBookLoggingAdvice(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {

    System.out.println("aroundReturnBookLoggingAdvice: в библиотеку пытаются вернуть книгу");

    long begin = System.currentTimeMillis();
    Object targetMethodResult = proceedingJoinPoint.proceed();
    long end = System.currentTimeMillis();

    System.out.println("aroundReturnBookLoggingAdvice: в библиотеку успешно вернули книгу");
    System.out.println("aroundReturnBookLoggingAdvice: метод returnBook отработал за " + (end - begin) + " миллисекунды");
    return targetMethodResult;
}
```

Вывод:

```
aroundReturnBookLoggingAdvice: в библиотеку пытаются вернуть книгу
Мы возвращаем книгу в UniLibrary
aroundReturnBookLoggingAdvice: в библиотеку успешно вернули книгу
aroundReturnBookLoggingAdvice: метод returnBook отработал за 24 миллисекунды
В библиотеку вернули книгу Война и мир
```

Advice @Around

Можно изменить результат работы метода, как и было указано выше

```java
@Around("execution(public String returnBook())")
public Object aroundReturnBookLoggingAdvice(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {

    System.out.println("aroundReturnBookLoggingAdvice: в библиотеку пытаются вернуть книгу");

    long begin = System.currentTimeMillis();
    Object targetMethodResult = proceedingJoinPoint.proceed();
    targetMethodResult = "название скрыто";
    long end = System.currentTimeMillis();

    System.out.println("aroundReturnBookLoggingAdvice: в библиотеку успешно вернули книгу");
    System.out.println("aroundReturnBookLoggingAdvice: метод returnBook отработал за " + (end - begin) + " миллисекунды");
    return targetMethodResult;
}
```

```
aroundReturnBookLoggingAdvice: в библиотеку пытаются вернуть книгу
Мы возвращаем книгу в UniLibrary
aroundReturnBookLoggingAdvice: в библиотеку успешно вернули книгу
aroundReturnBookLoggingAdvice: метод returnBook отработал за 15 миллисекунды
В библиотеку вернули книгу название скрыто
```

Advice @Around

Используя **Advice @Around** можно предпринять следующие действия, если было брошено исключение:

1. Ничего не делать

```
public String returnBook() {
    int a = 10/0;
    System.out.println("Мы возвращаем книгу в UniLibrary");
    return "Война и мир";
}
```

```
@Around("execution(public String returnBook())")
public Object aroundReturnBookLoggingAdvice(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {

    System.out.println("aroundReturnBookLoggingAdvice: в библиотеку пытаются вернуть книгу");

    Object targetMethodResult = proceedingJoinPoint.proceed();

    System.out.println("aroundReturnBookLoggingAdvice: в библиотеку успешно вернули книгу");
    return targetMethodResult;
}
```

```
aroundReturnBookLoggingAdvice: в библиотеку пытаются вернуть книгу
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
```

Advice @Around

2. Обработать исключение

```java
@Around("execution(public String returnBook())")
public Object aroundReturnBookLoggingAdvice(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {

    System.out.println("aroundReturnBookLoggingAdvice: в библиотеку пытаются вернуть книгу");

    Object targetMethodResult = null;

    try {
        targetMethodResult = proceedingJoinPoint.proceed();
    } catch (Exception e) {
        System.out.println("aroundReturnBookLoggingAdvice: исключение " + e);
        targetMethodResult = "Неизвестная книга";
    }

    System.out.println("aroundReturnBookLoggingAdvice: в библиотеку успешно вернули книгу");
    return targetMethodResult;
}
```

```
aroundReturnBookLoggingAdvice: в библиотеку пытаются вернуть книгу
aroundReturnBookLoggingAdvice: исключение java.lang.ArithmeticException: / by zero
aroundReturnBookLoggingAdvice: в библиотеку успешно вернули книгу
В библиотеку вернули книгу Неизвестная книга
```

Advice @Around

3. Пробросить исключение дальше

```java
@Around("execution(public String returnBook())")
public Object aroundReturnBookLoggingAdvice(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {

    System.out.println("aroundReturnBookLoggingAdvice: в библиотеку пытаются вернуть книгу");

    Object targetMethodResult = null;

    try {
        targetMethodResult = proceedingJoinPoint.proceed();
    } catch (Exception e) {
        System.out.println("aroundReturnBookLoggingAdvice: исключение " + e);
        throw e;
    }

    System.out.println("aroundReturnBookLoggingAdvice: в библиотеку успешно вернули книгу");
    return targetMethodResult;
}
```

```
aroundReturnBookLoggingAdvice: в библиотеку пытаются вернуть книгу
aroundReturnBookLoggingAdvice: исключение java.lang.ArithmeticException: / by zero
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
```