# Аспектно - ориентированное программирование

Лекция №1

Антонов В.В.

# АОП и сквозная функциональность

*Аспектно-ориентированное программирование (АОП)* — новый перспективный подход к разработке программ. Суть данного подхода — поддержка разработки и модификации сквозной функциональности (cross-cutting concerns) в больших программных системах.

# *modular concern*

При проектировании, реализации и модификации любой программы ее архитектура полностью или частично описывается в виде *иерархии модулей — классов, процедур, функций, реализующих различные функциональные возможности (функциональности) программы.*

*Простейшая из таких возможностей — например, вычисление какой-либо математической функции по известной формуле, — может быть реализована всего одним модулем в классическом смысле этого слова — функцией, процедурой, статическим методом, макросом, — имеющим, функциональную прочность.*

*Более сложная по семантике функциональность реализуется в виде иерархии классов, библиотеки функций и др. Такова, например, любая компонента большой программной системы, реализующая часть ее бизнес-логики, т. е. решающая конкретную задачу из прикладной области, — например, расчет зарплаты, расчет курсов акций, планирование распределения заданий между сотрудниками.*

Такого рода функциональность в программе, реализация которой выразима в виде одного модуля или взаимосвязанной совокупности модулей, будем называть *модульной функциональностью (modular concern).*

# *cross-cutting concerns*

Как показывает анализ архитектуры программных систем, существует также *сквозная функциональность (cross-cutting concerns). Данное понятие объединяет такие виды функциональных возможностей, идей, принципов, методов, реализация которых принципиально не осуществима в виде лишь одной иерархии взаимосвязанных модулей, а, в дополнение к ним, требует вставки в физически рассредоточенные* точки программы фрагментов нового кода (либо исполнения новых фрагментов кода в рассредоточенных точках целевой программы, без их явной вставки). Как правило, фрагменты кода, реализующие новую сквозную функциональность, являются исполняемыми конструкциями (операторами), чаще всего — вызовами, однако они могут быть и описаниями (определениями) данных, также являющихся частью реализации рассматриваемой сквозной функциональности.

# Типичные классы задач, решаемые АОП

Типичные классы задач, решение которых требует реализации сквозной функциональности, относятся к *надежному и безопасному программированию (trustworthy computing)*

- ***безопасность (security)*** *— аутентификация пользователей и программ; авторизация (проверка полномочий кода или пользователя для выполнения тех или иных действий); криптографические операции над данными с целью обеспечения их конфиденциальности и т. д.;*
- ***надежность (reliability)*** *— проверка выполнения предусловий и постусловий в модулях и инвариантов в классах; обработка ошибок и др.;*
- ***безопасность многопоточного выполнения кода (multi-threaded safety)*** *— синхронизация по ресурсам или по событиям, выделение критических участков кода, взаимное исключение доступа к ним и др.;*
- ***протоколирование и профилирование работы программы (logging and profiling)*** *— трассировка начала и окончания выполнения каждой функции (каждого метода), вывод их аргументов и результатов, сбор и вывод статистической информации об исполнении различных фрагментов программы.*

# Основные понятия АОП

- **Точка соединения (joinpoint)** — точка в программе, где существует возможность выполнить дополнительный код средствами АОП. Различные реализации АОП имеют различные возможные точки соединения, таковыми могут являться момент вызова методов класса или обращений к полям объекта.
- **Совет (advice)** —класс, реализующий сквозную функциональность. Существуют различные типа советов:  выполняемые до точки соединения, после или вместо неё.
- **Срез (pointcut)** —точка соединения (joinpoint), которая выбрана для исполнения в ней сквозной функциональности, определенная советом (advice).
- **Аспект (aspect)** — под аспектом понимают комбинацию, состоящую из среза (pointcut) и реализующего сквозную функциональность  совета (advice). Аспект изменяет поведение остального кода, исполняя *совет* в *точках соединения*, определённых некоторым *срезом*. В Spring для этого используется также понятие advisor.
- **Внедрение или введение (introduction)** — под этим термином понимают процесс модификации объекта путем добавления дополнительных полей и /или методов. Внедрение также может быть использовано для реализации объектом интерфейса без явного указания этого в классе объекта.
- **Связывание (weaving) –** связывание аспектов с объектами для создания новых, «расширенных» объектов.
- **Цель или целевой объект (target) –** объект, являющийся результатом связывания (weaving), то есть реализующий первоначальную бизнес логику плюс сквозная функциональность, выполненная одним или несколькими аспектами.

# Различные типы Аспектно-Ориентированного Программирования

- Существует два различных способа реализации аспектно-ориентированного программирования: статический и динамический. Эти способы различаются моментами времени, когда происходит связывание (weaving) и способом, как это связывание происходит.

# Статическое АОП

- При статической реализации аспектно-ориентированного программирования связывание является отдельным шагом в процессе построения программного продукта (build process) путем модификации байт-кода (bytecode) классов, изменяя и дополняя его необходимым образом.
- Полученный в результате такого подхода код является более производительным, чем при использовании динамического АОП, так как во время исполнения (runtime) нет необходимости отслеживать момента, когда надо выполнить ту или иную сквозную функциональность, представленную в виде совета (aspect).
- Недостатком такого подхода реализации аспектно-ориентированного программирования является необходимость перекомпилирования приложения даже в том случае, когда надо только добавить новый срез (pointcut).

# Динамическое АОП

- Продукты, реализующие динамический вариант АОП отличается от статического тем, что процесс связывания (weaving) происходит динамически в момент исполнения. В Spring Framework используется именно такой способ связывания и это реализовано с помощью использования специальных объектов-посредников (proxy) для объектов, к которым должны быть применены советы (advice).
- Недостатки статического подхода АОП являются достоинствами динамического: поскольку связывание происходит динамически, то нет необходимости перекомпилировать приложение для изменения аспектов. Однако эта гибкость достигается ценой небольшой потери производительности.

# Инструменты АОП

- AspectJ
- JBoss
- Aspect.NET
- Spring Framework

# Исключения и их обработка

**Исключение** - это проблема(ошибка) возникающая во время выполнения программы. Исключения могут возникать во многих случаях, например:

- Пользователь ввел некорректные данные.
- Файл, к которому обращается программа, не найден.
- Сетевое соединение с сервером было утеряно во время передачи данных.

# Синтаксис исключений

- **try** - данное ключевое слово используется для отметки начала блока кода, который потенциально может привести к ошибке.
- **catch** - ключевое слово для отметки начала блока кода, предназначенного для перехвата и обработки исключений.
- **finally** - ключевое слово для отметки начала блока кода, которое является дополнительным. Этот блок помещается после последнего блока 'catch'. Управление обычно передаётся в блок 'finally' в любом случае.
- **throw** - служит для генерации исключений.
- **throws** - ключевое слово, которое прописывается в сигнатуре метода, и обозначающее что метод потенциально может выбросить исключение с указанным типом.

# Пример выброса исключения

```
public static void main(String[] args) {
    System.out.println(getAreaValue(-1, 100));
}

public static int getAreaValue(int x, int y){
    if(x < 0 || y < 0)
    throw new
    IllegalArgumentException("value of 'x' or 'y' is negative: x="+x+", y="+y);
    return x*y;
}
```

# Конструкция «поимки» исключения

```
try{
//здесь код, который потенциально может привести к ошибке
}
catch(SomeException e ){ //в скобках указывается класс конкретной ожидаемой ошибки
//здесь описываются действия, направленные на обработку исключений
}
finally{
//выполняется в любом случае ( блок finnaly не обязателен)
}
```

# Модифицированный пример

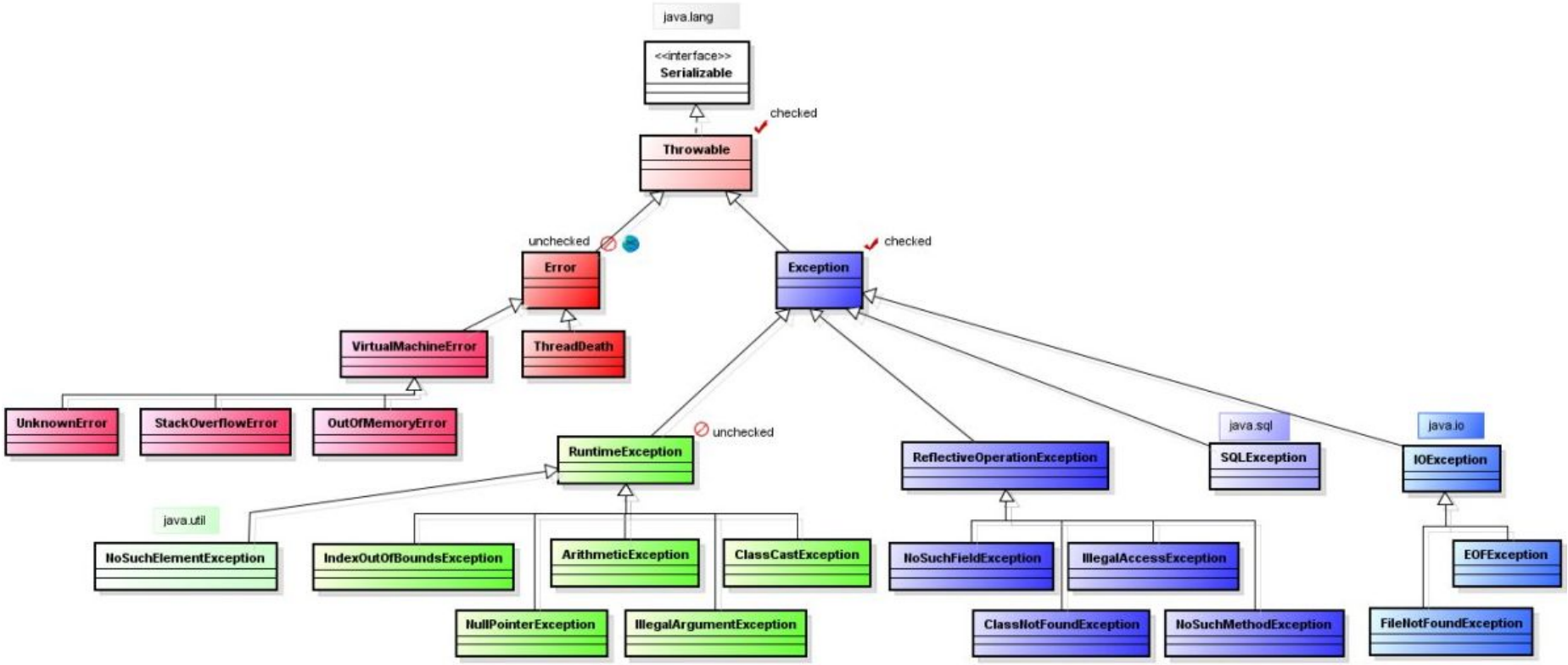
```
public static void main(String[] args) {

    int result = 0;

    try{
        result = getAreaValue(-1, 100);
    }catch(IllegalArgumentException e){
        Logger.getLogger(NewClass.class.getName()).log(
        new LogRecord(Level.WARNING,
        "В метод вычисления площади был передан аргумент с негативным значением!"));
        throw e;
    }
    System.out.println("Result is : "+result);
}

public static int getAreaValue(int x, int y){
    if(x < 0 || y < 0) throw new IllegalArgumentException(
    "value of 'x' or 'y' is negative: x="+x+", y="+y);
    return x*y;
}
```

# finaly

```
class FinallyDemo {
    static void procA() {
        try {
            System.out.println("inside procA");
            throw new RuntimeException("demo");
        } finally {
            System.out.println("procA's finally");
        }
    }
    static void procB() {
        try {
            System.out.println("inside procB");
            return;
        } finally {
            System.out.println("procB's finally");
        }
    }
    public static void main(String args[]) {
        try {
            procA();
        } catch (Exception e) {
        }
        procB();
    }
}
```

*inside procA*
*procA's finally*
*inside procB*
*procB's finally*

# Иерархия исключений



# Иерархия исключений Error

- **Error** - это подкласс, который показывает серьезные проблемы возникающие во время выполнения приложения. Большинство из этих ошибок сигнализируют о ненормальном ходе выполнения программы, т.е. о каких-то критических проблемах. Эти ошибки не рекомендуется отмечать в методах посредством throws-объявления, поэтому они также очень часто называются не проверяемые (unchecked)

# Иерархия исключений Exception

- **Exception**. Эта иерархия также разделяется на две ветви: исключения, производные от класса RuntimeException, и остальные. Исключения типа RuntimeException возникают вследствие ошибок программирования. Все другие исключения являются следствием непредвиденного стечения обстоятельств, например, ошибок ввода-вывода, возникающих при выполнении вполне корректных программ.

# два типа исключений: checked и unchecked

- 1. **Checked** исключения, это те, которые должны обрабатываться блоком catch или описываться в сигнатуре метода. Unchecked могут не обрабатываться и не быть описанными.
- 2. **Unchecked** исключения в Java - наследованные от RuntimeException, checked - от Exception (не включая unchecked).

Примеры:

unchecked исключения - NullPointerException,

checked исключения - IOException

# Создание собственных исключений

```
public class MyException extends Exception{
    public MyException(Throwable e) {
        initCause(e);
    }
}

public class NewClass {
    public static void main(String[] args) throws MyException {

        int result = 0;

        try{
            result = getAreaValue(-1, 100);
        }catch(IllegalArgumentException e){
            Logger.getLogger(NewClass.class.getName()).log(
            new LogRecord(Level.WARNING,
            "В метод вычисления площади был передан аргумент с негативным значением!"));
            throw new MyException(e);
        }
        System.out.println("Result is : "+result);
    }

    public static int getAreaValue(int x, int y){
        if(x < 0 || y < 0) throw
        new IllegalArgumentException("value of 'x' or 'y' is negative: x="+x+", y="+y);
        return x*y;
    }
}
```

# Обработка нескольких исключений

```
class Main {
    public static void main(String[] args) {
        FileInputStream fis = null;
try{
fis = new FileInputStream(fileName);
}catch (FileNotFoundException ex){
System.out.println("Oops, FileNotFoundException caught")
}catch (IOException e) {
System.out.println("IOEXCEOTION");
}
        }
    }
}
```