Белорусский Государственный Университет Информатики и Радиоэлектроники
кафедра интеллектуальных информационных технологий

ЭКЗАМЕН

по курсу «Проектирование программ в интеллектуальных системах»
(4 семестр)

Минск 2021

Список теоретических вопросов

1. Java. Структура программы

Илья и Антон

2. Java. Классы и интерфейсы

Класс

Класс — шаблон (описание) объекта. В Java класс определяется с помощью ключевого слова ***class***.

Любой объект может обладать двумя основными характеристиками: состояние — некоторые данные, которые хранит объект, и поведение — действия, которые может совершать объект. Для хранения состояния объекта в классе применяются *поля* (или *переменные класса)*. Для определения поведения объекта в классе применяются *методы*.

Кроме обычных методов классы могут определять специальные методы, которые называются конструкторами. Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта. Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор без параметров.

Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Для переменных числовых типов это число 0, а для классов — это значение null (то есть фактически отсутствие значения). Если необходимо, чтобы при создании объекта производилась какая-то логика, например, чтобы поля класса получали какие-то определенные значения, то можно определить в классе свои конструкторы.

Абстрактный класс

Кроме обычных классов в Java есть абстрактные классы. Абстрактный класс похож на обычный класс — в нем также можно определить поля и методы, но в то же время нельзя создать объект или экземпляр абстрактного класса. Абстрактные классы призваны предоставлять базовый функционал для классов-наследников. А производные классы уже реализуют этот функционал.

При определении абстрактных классов используется ключевое слово ***abstract***.

Но главное отличие состоит в том, что мы не можем использовать конструктор абстрактного класса для создания его объекта.

Производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе. Также следует учитывать, что если класс имеет хотя бы один абстрактный метод, то данный класс должен быть определен как абстрактный.

Интерфейс

Механизм наследования очень удобен, но он имеет свои ограничения. В частности в Java мы можем наследовать только от одного класса, в отличие, например, от языка C++, где имеется множественное наследование.

В языке Java подобную проблему частично позволяют решить интерфейсы. Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. И один класс может применить множество интерфейсов. В парадигме ООП, интерфейс — декларирование некоторых обязательств.

Чтобы определить интерфейс, используется ключевое слово ***interface***.

Интерфейс может определять константы и методы, которые могут иметь, а могут и не иметь реализации. Методы без реализации похожи на абстрактные методы абстрактных классов. Чтобы класс применил интерфейс, надо использовать ключевое слово ***implements***. При этом надо учитывать, что если класс применяет интерфейс, то он должен реализовать все методы интерфейса. Если класс не реализует какие-то методы интерфейса, то такой класс должен быть определен как абстрактный, а его неабстрактные классы-наследники затем должны будут реализовать эти методы.

Методы по умолчанию.

Ранее интерфейс мог содержать только определения методов без конкретной реализации. В JDK 8 была добавлена такая функциональность как методы по умолчанию. И теперь интерфейсы кроме определения методов могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод. Метод по умолчанию — это обычный метод без модификаторов, который помечается ключевым словом ***default***.

Статические методы.

Начиная с JDK 8 в интерфейсах доступны также статические методы — они аналогичны статическим методам класса. Чтобы обратиться к статическому методу интерфейса также, как и в случае с классами, пишут название интерфейса и метод.

Приватные методы.

По умолчанию все методы в интерфейсе фактически имеют модификатор ***public***. Однако начиная с Java 9 мы также можем определять в интерфейсе методы с модификатором ***private***. Они могут быть статическими и нестатическими, но они не могут иметь реализации по умолчанию (не могут быть помечены ключевым словом ***default***).

Подобные методы могут использоваться только внутри самого интерфейса, в котором они определены.

Константы в интерфейсах.

Кроме методов в интерфейсах могут быть определены статические константы. Хотя такие константы также не имеют модификаторов, но по умолчанию они имеют модификатор доступа ***public static final***, и поэтому их значение доступно из любого места программы.

Множественная реализация интерфейсов.

Если нам надо применить в классе несколько интерфейсов, то они все перечисляются через запятую после слова ***implements***.

Наследование интерфейсов.

Интерфейсы, как и классы, могут наследоваться. Только в отличие от классов, интерфейс может наследовать любое количество других интерфейсов.

Основные отличия и сходства

		Класс	**Абстрактный класс**	**Интерфейс**
Наследует (***extends***)		Только 1 другой класс		Любое количество других интерфейсов
Реализует (***implements***)		Любое количество интерфейсов		-
Поля		Любые модификаторы доступа, переменные и константы		Только ***public static final***
Методы	Неабстрактные	Любые модификаторы доступа		***private***, ***private static***, ***public default***
	Абстрактные	-	Любые модификаторы доступа, кроме ***private***	Только ***public***

Контроль доступа

	Class	**Package**	**Subclass**	**World**
private	+	-	-	-
no specifier	+	+	-	-
protected	+	+	+	-
public	+	+	+	+

Использованные источники

1. Сердюков Р.Е. - Интерфейс (лекция).
2. Статья metanit.com – Классы и объекты.
3. Статья metanit.com – Абстрактные классы.
4. Статья metanit.com – Интерфейсы.

3. Java. Управление памятью. Сборщик мусора

Управление памятью

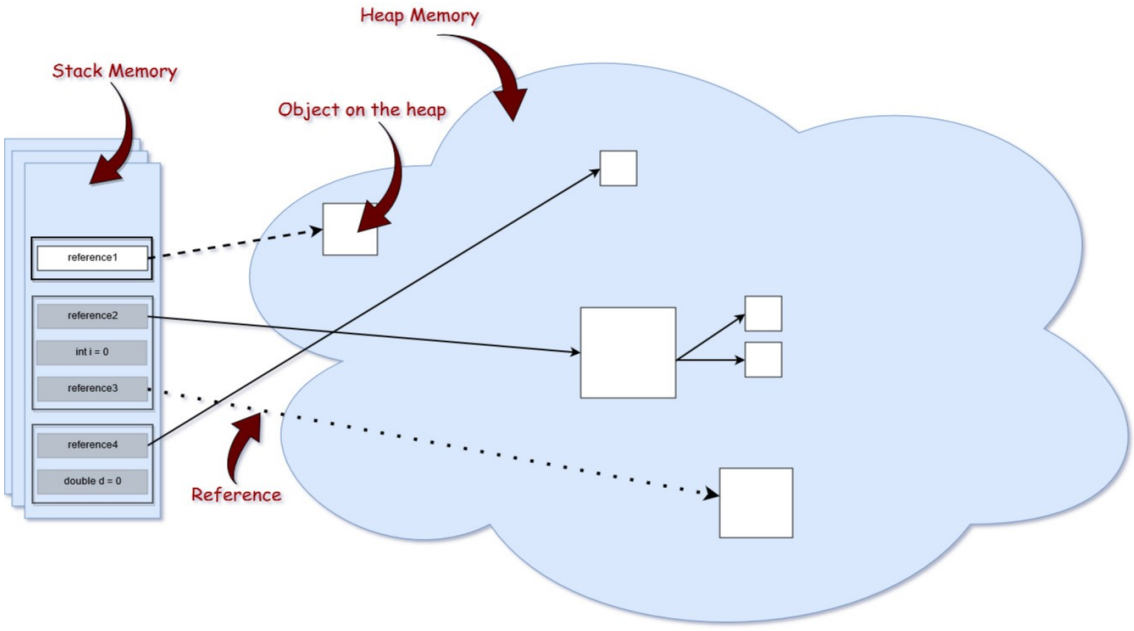
В Java память делится на две большие части:

- Стек (Stack) — отвечает за хранение ссылок на объекты кучи и за хранение примитивных типов, которые содержат само значение, а не ссылку на объект из кучи. Переменные в стеке имеют область видимости (если компилятор выполняет тело метода, он может получить доступ только к объектам из стека, которые находятся внутри тела метода, а когда метод завершается и возвращает значение, верхняя часть стека выталкивается, и активная область видимости изменяется). В Java стековая память выделяется для каждого потока (следовательно, каждый раз, когда поток создается и запускается, он имеет свою собственную стековую память и не может получить доступ к стековой памяти другого потока).
- Куча (Heap) — хранит в памяти фактические объекты, на которые ссылаются переменные из стека.

```
Object obj = new Object();
```

Ключевое слово **new** несет ответственность за обеспечение того, достаточно ли свободного места на куче.

Для каждого запущенного процесса JVM существует только одна область памяти в куче. Следовательно, это общая часть памяти независимо от того, сколько потоков выполняется.



Максимальные размеры стека и кучи не определены заранее - это зависит от работающей JVM машины.

Типы ссылок:

- Сильная ссылка. Это самые популярные ссылочные типы, к которым мы все привыкли. В приведенном выше примере с Object мы фактически храним сильную ссылку на объект из кучи. Объект в куче не удаляется сборщиком мусора, пока на него указывает сильная ссылка или если он явно доступен через цепочку сильных ссылок.
- Слабая ссылка. Попросту говоря, слабая ссылка на объект из кучи, скорее всего, не сохранится после следующего процесса сборки мусора.
- Мягкая ссылка. Эти типы ссылок используются для более чувствительных к памяти сценариев, поскольку они будут собираться сборщиком мусора только тогда, когда вашему приложению не хватает памяти. Следовательно, пока нет критической необходимости в освобождении некоторого места, сборщик мусора не будет касаться легко доступных объектов. В документации Javadocs говорится:

  «все мягкие ссылки на мягко достижимые объекты гарантированно очищены до того, как виртуальная машина выдаст OutOfMemoryError»

- Фантомная ссылка. Используется для планирования посмертных действий по очистке, поскольку мы точно знаем, что объекты больше не живы. Эти типы ссылок считаются предпочтительными для финализаторов.

Ссылки на String.

Ссылки на тип String в Java обрабатываются немного по- другому. Строки неизменяемы, что означает, что каждый раз, когда вы делаете что-то со строкой, в куче фактически создается другой объект. Для строк Java управляет пулом строк (String Pool) в памяти. Это означает, что Java сохраняет и повторно использует строки, когда это возможно.

```
String str1 = «string»;
```

```
String str2 = «string»;
```

В этом случае две ссылки типа String указывают на один и тот же объект в пуле строк, значит при попытке сравнения ссылок, получим true. Однако можно принудить Java создать новый объект в куче при помощи оператора **new**:

```
String str1 = «string»;
```

```
String str2 = new String(«string»);
```

В таком случае имеем два разных объекта в куче с одинаковым значением, и при попытке сравнения ссылок получим false.

Сборщик мусора

Сборщик мусора (Garbage Collector) — это низкоприоритетный процесс, который запускается периодически и освобождает память, использованную объектами, которые больше не нужны. Разные JVM имеют отличные друг от друга алгоритмы сбора мусора.

Поскольку это довольно сложный процесс и может повлиять на вашу производительность, он реализован разумно. Для этого используется так называемый процесс «Mark and Sweep». Java анализирует переменные из стека и «отмечает» все объекты, которые необходимо поддерживать в рабочем состоянии. Затем все неиспользуемые объекты очищаются.

Так что на самом деле Java не собирает мусор. Фактически, чем больше мусора и чем меньше объектов помечены как живые, тем быстрее идет процесс. Чтобы сделать это еще более оптимизированным, память кучи на самом деле состоит из нескольких частей.

Фактически, JVM имеет три типа сборщиков мусора, и программист может выбрать, какой из них следует использовать. По умолчанию Java выбирает используемый тип сборщика мусора в зависимости от базового оборудования.

1. **Serial GC** (Последовательный сборщик мусора) - однониточный коллектор. В основном относится к небольшим приложениям с небольшим использованием данных.
2. **Parallel GC** (Параллельный сборщик мусора) - даже по названию, разница между последовательным и параллельным будет заключаться в том, что параллельный сборщик мусора использует несколько потоков для выполнения процесса сбора мусора. Этот тип GC также известен как сборщик производительности.
3. **Mostly concurrent GC** (В основном параллельный сборщик мусора). Если вы помните, ранее в этой статье упоминалось, что процесс сбора мусора на самом деле довольно дорогостоящий, и когда он выполняется, все потоки приостанавливаются. Однако у нас есть в основном параллельный тип GC, который утверждает, что он работает одновременно с приложением. Однако есть причина, по которой он «в основном» параллелен. Он не работает на 100% одновременно с приложением. Есть период времени, на который цепочки приостанавливаются. Тем не менее, пауза делается как можно короче для достижения наилучшей

производительности сборщика мусора. На самом деле существует 2 типа в основном параллельных сборщиков мусора:

- **Garbage First** - высокая производительность с разумным временем паузы приложения.
- **Concurrent Mark Sweep** (Параллельное сканирование отметок) - время паузы приложения сведено к минимуму. Начиная с JDK 9, этот тип GC объявлен устаревшим.

Использованные источники.

1. Статья habr.com – Управление памятью Java.
2. Статья javarush.ru – Сборка мусора.

4. Java. Внутренние и анонимные классы. Lambda-выражения

Илья и Антон

5. Java. Generic

Начиная с JDK 1.5 (Java SE 5), в Java появились новые возможности для программирования. Одним из таких нововведений являются Generics.

Generics (обобщения) — это особые средства языка Java для реализации обобщённого программирования: особого подхода к описанию данных и алгоритмов, позволяющего работать с различными типами данных без изменения их описания.

Вот типичное использование коллекции без Generics:

```
List integerList = new LinkedList();
integerList.add(0);
Integer x = (Integer) integerList.iterator().next();
```

Как правило, программист знает, какие данные должны быть в List'е. Для того чтобы избежать лишних приведений типов и возможностей появления «Runtime Error» следует использовать дженерики**:**

```
List<Integer> integerList = new LinkedList<Integer>();
integerList.add(0);
Integer x = integerList.iterator().next();
```

Это явное указание типа объектов лежащих в integerList.

Задачи Generics:

- Явно указывать, какие данные хранятся в коллекции, избегать лишнего приведения типов и добавления объектов не нужных нам типов.
- Использование одного кода для разных типов объектов (то есть реализация полиморфизма). Писать для каждого отдельного типа свою версию класса не является хорошим решением, так как в этом случае мы вынуждены повторяться.

Шаблонные классы

Писать для каждого отдельного типа свою версию класса не является хорошим решением, так как в этом случае мы вынуждены повторяться. Поэтому определим класс Account как обобщенный:

```
class Account<T>{

    private T id;
    private int sum;

    Account(T id, int sum){
        this.id = id;
        this.sum = sum;
    }

    public T getId() { return id; }
    public int getSum() { return sum; }
    public void setSum(int sum) { this.sum = sum; }
}
```

С помощью буквы **T** в определении класса ***class Account<T>*** мы указываем, что данный тип **T** будет использоваться этим классом. Параметр **T** в угловых скобках называется универсальным параметром, так как вместо него можно подставить любой тип.

При определении переменной данного класса и создании объекта после имени класса в угловых скобках нужно указать, какой именно тип будет использоваться вместо универсального параметра. При этом надо учитывать, что они работают только с объектами, но не работают с примитивными типами. Вместо примитивных типов следует использовать классы-обертки (например Integer вместо int).

Шаблонные методы

Особенностью обобщенного метода является использование универсального параметра в объявлении метода после всех модификаторов и перед типом возвращаемого значения.

```
class Printer{

    public <T> void print(T[] items){
        for(T item: items){
            System.out.println(item);
        }
    }
}
```

Затем внутри метода все значения типа T будут представлять данный универсальный параметр. При вызове подобного метода перед его именем в угловых скобках указывается, какой тип будет передаваться на место универсального параметра:

```
printer.<String>print(people);
printer.<Integer>print(numbers);
```

Шаблонные интерфейсы

При реализации обобщённого интерфейса есть две стратегии:

- Когда при реализации для универсального параметра интерфейса задается конкретный тип (в примере это тип String):

```
interface Accountable<T>{
    T getId();
    int getSum();
    void setSum(int sum);
}
class Account implements Accountable<String>{

    private String id;
    private int sum;

    Account(String id, int sum){
        this.id = id;
        this.sum = sum;
    }

    public String getId() { return id; }
    public int getSum() { return sum; }
    public void setSum(int sum) { this.sum = sum; }
}
```

Тогда класс, реализующий интерфейс, жестко привязан к этому типу.

- Вторая стратегия представляет определение обобщенного класса, который также использует тот же универсальный параметр:

```
interface Accountable<T>{
    T getId();
    int getSum();
    void setSum(int sum);
}
class Account<T> implements Accountable<T>{

    private T id;
    private int sum;

    Account(T id, int sum){
        this.id = id;
        this.sum = sum;
    }

    public T getId() { return id; }
    public int getSum() { return sum; }
    public void setSum(int sum) { this.sum = sum; }
}
```

Шаблонные конструкторы

Конструкторы, как и методы также могут быть обобщенными. В этом случае перед конструктором также указываются в угловых скобках универсальные параметры:

```
class Account{

    private String id;
    private int sum;

    <T>Account(T id, int sum){
        this.id = id.toString();
        this.sum = sum;
    }
```

В данном случае конструктор принимает параметр id, который представляет тип **T.** В конструкторе его значение превращается в строку и сохраняется в поле класса.

Использование нескольких универсальных параметров

В угловых скобках можно указывать неограниченное количество универсальных параметров. В данном случае тип String будет передаваться на место параметра T, а тип Double — на место параметра S:

```
class Account<T, S>{

    private T id;
    private S sum;

    Account(T id, S sum){
        this.id = id;
        this.sum = sum;
    }

    public T getId() { return id; }
    public S getSum() { return sum; }
    public void setSum(S sum) { this.sum = sum; }
}
```

Ограничения обобщений

Когда мы указываем универсальный параметр у обобщений, то по умолчанию он может представлять любой тип. Однако иногда необходимо, чтобы параметр соответствовал только некоторому ограниченному набору типов. В этом случае применяются ограничения, которые позволяют указать базовый класс, которому должен соответствовать параметр.

Для установки ограничения после универсального параметра ставится слово ***extends***, после которого указывается базовый класс ограничения:

```
class Account{ }
class Transaction<T extends Account>{ }
```

То есть на место параметра T мы можем передать либо класс Account, либо один из его классов-наследников.

В качестве ограничений могут выступать и другие обобщения, которые сами могут иметь ограничения:

```
class Transaction<T extends Account<String>>{
}
```

В качестве ограничений могут выступать также интерфейсы. В этом случае передаваемый на место универсального параметра тип должен реализовать данный интерфейс:

```
class Transaction<T extends Accountable>{
```

Также можно установить сразу **несколько ограничений.** Например, пусть класс Transaction может работать только с объектами, которые одновременно реализуют интерфейс Accountable и являются наследниками класса Person:

```
class Person{}
interface Accountable{}

class Transaction<T extends Person & Accountable>{}
```

Использованные источники

1 Статья metanit.com - Обобщения (Generics).

2 Обобщение типа данных, generic.

3 Статья javarush.ru - Теория дженериков в Java или как на практике ставить скобки.

4 Статья metanit.com - Ограничения обобщений.

6. Java. Типы потоков ввода-вывода, принцип использования

Классификация потоков

Потоки бывают:

- Потоки «сырых» (raw) байт (Byte Streams).
- Потоки символов (Character Streams).
- Буферизированные потоки (Buffered Streams).
- Потоки примитивных данных (Data Streams).
- Потоки объектов (Object Streams).
- Фильтрующие потоки.

По направлению движения данных потоки можно разделить на две группы:

- Поток ввода (Input) — данные поступают из потока в нашу программу. Мы их читаем из этого потока.
- Поток вывода (Output) — данные поступают в поток из нашей программы. Мы их пишем в этот поток.

Вторым критерием разделения может служить ТИП передаваемых данных:

- Поток байтов.
- Поток символов.

В итоге мы получаем 4 типа потоков. Для каждого из этих типов Java предлагает отдельный базовый абстрактный класс:

- InputStream — поток для чтения байтов (поток ввода).
- OutputStream — поток для записи байтов (поток вывода).
- Reader — поток для чтения символов (поток ввода).
- Writer — поток для записи символов (поток вывода).

Классы являются абстрактными, потому что у нас есть специализация — файлы, сеть, память. И расширяя базовый класс специальный класс решает свои специальные задачи. Но базовые функции для всех одинаковые. Что удобно — все специальные потоки по своей сути одно и тоже. Это дает гибкость и универсальность.

InputStream	OutputStream	Reader	Writer
FileInputStream	FileOutputStream	FileReader	FileWriter
BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
DataInputStream	DataOutputStream		
ObjectInputStream	ObjectOutputStream		

Фильтрующие потоки.

Принцип работы FilterInputStream:

- Объект FilterInputStream получает ввод от другого объекта InputStream, некоторым образом обрабатывает (фильтрует) байты и возвращает результат.
- Фильтрующие потоки могут объединяться в последовательности, при этом несколько фильтров превращаются в один сквозной фильтр.

Аналогичным образом работает для фильтрация вывода.

Примеры фильтрующих потоков:

- DataInputStream.
- BufferedInputStream.
- LineNumberInputStream.
- PushbackInputStream.

Принцип использования

Схема работы с потоком в упрощенном виде выглядит так:

- Создается экземпляр потока.
- Поток открывается (для чтения или записи).
- Производится чтение из потока/запись в поток.
- Поток закрывается.

Первые два пункта часто совмещены в рамках одного действия. По сути потоки можно представить как трубу, в которую «заливаются» байты или символы.

Причем что еще интереснее, эти трубы можно «склеивать» друг с другом. Т.е. один поток может передавать данные в другой, предварительно как-то их модифицируя.

Пример.

```java
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class WriterReader {
    public static void main(String[] args) {
        writeText();
        readText();
    }

    private static void writeText() {
        // Эту строку мы посимвольно запишем в файл
        String test = "TEST!!!";

        /*
         *  Создание файлового потока для записи символов как автозакрываемый
         *  ресурс. Нам не надо вызывать fw.close(), т.к. в данном случае он
         *  будет закрыт автоматически.
         */
        try (FileWriter fw = new FileWriter("text.txt")) {
            // Записываем посимвольно , обращаясь к каждому элементу строки
            for (int i = 0; i < test.length(); i++) {
                fw.write(test.charAt(i));
            }
        } catch (IOException ex) {
            ex.printStackTrace(System.out);
        }
    }

    private static void readText() {
        try (FileReader fr = new FileReader("text.txt")) {
            // Переменная для хранения строки
            StringBuilder sb = new StringBuilder();
            int code = -1;
            // Читаем посимвольно пока код символа не станет равным -1
            while ((code = fr.read()) != -1) {
                // Вызов Character.toChars() преобразует int в char
                sb.append(Character.toChars(code));

            }
            System.out.println(sb.toString());
        } catch (IOException ex) {
            ex.printStackTrace(System.out);
        }
    }
}
```

Для записи строки в файл нам потребуется поток для символов — Writer. А для прочтения файла используем Reader. Но Reader и Writer — абстрактные классы. Поэтому для работы с файлами нам потребуются уже конкретные и это будут FileReader и FileWriter.

Пример цепочки из нескольких потоков.

```java
public static void main(String[] args) {
    String s;
    try {
        FileReader fr = new FileReader("C:\\rohit.txt");
        BufferedReader br = new BufferedReader(fr);
        LineNumberReader lr = new LineNumberReader(br);
        while ((s = lr.readLine()) != null)
            System.out.println(lr.getLineNumber() + " " + s);
    } catch (IOException e) {
        System.err.println(e.getMessage());
    }
}
```

Использованные источники

1. Статья java-course.ru - Потоки ввода-вывода.
2. Цепочки потоков.
3. Сердюков Р.Е. - Фильтрующие потоки (презентация).

7. **Java. Сериализация объектов в raw**

8. Java. Сериализация объектов в XML

Сериализация — это процесс сохранения состояния объекта в последовательность байт.

Десериализация — это процесс восстановления объекта из этих байт.

Для осуществления процессов сериализации и десериалиизации класс должен реализовывать интерфейс-маркер Serializable. Данный интерфейс не имеет ни одного метода для реализации — он нужен только для того, чтобы предоставить информацию JVM, о том, что объект подлежит сериализации.

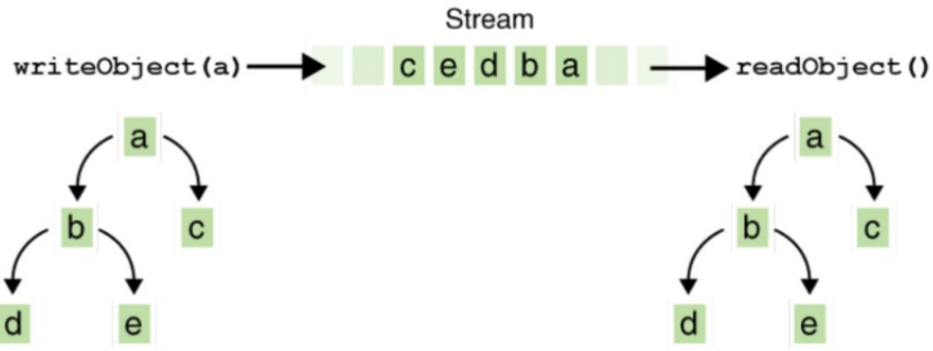
Объявление сериализуемого класса:

- Не сериализуется поля помеченные static и transient.
- Поля помеченные transient при десериализации инициализируются дефолтными значениями.
- При десериализации выделяется память под объект и поля инициализируются значениями из потока, конструктор этого объекта не вызывается.

  Но вызываются все конструкторы суперклассов в заданной последовательности до класса, имплементирующего Serializable.
- При сериализации объекта класса, реализующего интерфейс Serializable, учитывается порядок объявления полей в классе.
- ***private static final long serialVersionUID*** — это поле содержит уникальный идентификатор версии сериализованного класса.

  Идентификатор версии есть у любого класса, который имплементирует интерфейс Serializable. Он вычисляется по содержимому класса — полям, порядку объявления, методам. И если мы поменяем в нашем классе тип поля и/или количество полей, идентификатор версии моментально изменится. Поле serialVersionUID тоже записывается при сериализации класса. Когда мы пытаемся провести десериализацию, значение serialVersionUID сравнивается со значением serialVersionUID класса в нашей программе. Если значения не совпадают, будет выброшено исключение java.io.InvalidClassException.

Сериализация сложных объектов:



Алгоритм сериализации в Java

1. Запись метаданных о классе ассоциированном с объектом.
2. Рекурсивная запись описания суперклассов, до тех пор пока не будет достигнут java.lang.object.
3. После окончания записи метаданных начинается запись фактических данных ассоциированных с экземпляром, только в этот раз начинается запись с самого верхнего суперкласса.
4. Рекурсивная запись данных ассоциированных с экземпляром начиная с самого низшего суперкласса.

Форматы сериализации

Последовательность байт, полученная при сериализации, может быть представлена в разных форматах. Сделано это не просто так. Вероятно, у каждого из них есть свои преимущества и недостатки по сравнению с остальными.

Основные форматы сериализации:

- **JSON** (JavaScript Object Notation) — объекты Java, преобразованные в JSON, действительно выглядят точно так же, как объекты в языке JavaScript.

  Преимущества:

  – Human-readable («человеко-читаемый») формат.

  – Простота.

  – Широкая распространенность.

- **YAML** (Yet Another Markup Language или YAML Ain’t Markup Language).

Преимущества:

- Human-readable («человеко-читаемый») формат.
- Компактность.
- Поддержка структур данных, «родных» для языков программирования (map, set и т.п.).
- Возможность использования anchor (якорь: &) и alias (псевдоним: *).
- В YAML можно встроить данные в других форматах.

- **XML** (eXtensible Markup Language) — расширяемый язык разметки.

  Преимущества:

  - Human-readable («человеко-читаемый») формат.
  - Простота создания.
  - Простота использования.
  - Расширяемость.
  - Отладка и исправление ошибок.
  - Безопасность.
  - Поддержка метаданных.

Сериализация в XML

Условия сериализации в XML:

- Объявление класса со спецификатором public;
- Объявление public-конструктора без параметров (по умолчанию);
- Объявление инкапсулированных полей;
- Объявление корректных get-еров и set-еров для каждого нестатического поля;

В Java1.4 добавлена сериализация в XML без использования интерфейса Serializable.

Пример:

```
try (XMLEncoder xmlEncoder = new XMLEncoder (new BufferedOutputStream(new
    FileOutputStream("file.xml")))) {
    Student student = new Student("Biceps", 921704);
    xmlEncoder.writeObject(student);
    xmlEncoder.flush();
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

Альтернатива сериализации

В Java есть несколько API для работы с XML, которые представлены так называемыми парсерами (парсинг — процесс автоматического сбора данных и их структурирования).

Основные XML API:

- DOM (Document Object Model) — работает со всем XML-документом, загружает его в память и строит дерево представления докуента.
- SAX (Simple API for XML) — анализирует документ на основе обратных вызовов, не загружая его в память (более быстрый, но менее функциональный по сравнению с DOM).
- StAX — последовательно читает из документа события, анализирует их и обрабатывает подходящие (что-то среднее между DOM и SAX).
- JAXB (Java Architecture for XML Building) — наиболее удобный и часто используемый API для работы с XML, когда требуется почитать весь XML (он должен поместиться в память JVM) и выполнить с ним требуемые действия.
- Jackson XML — расширение Jackson JSON для чтения и записи данных в XML.

Использованные источники

1. Статья javarush.ru - Форматы сериализации в Java.
2. Статья habr.com - Сериализация в Java.
3. Сердюков Р.Е. - Сериализация объектов (лекция).

9. **Java. Чтение и запись в сетевой ресурс**

Илья и Антон

10. **Java. Основные возможности библиотеки javax.swing**

Илья и Антон

11. **Java. Обработка событий в библиотеке javax.swing**

Илья и Антон

12. **Java. Многопоточность в библиотеке javax.swing**

Илья и Антон

13. Java. Enterprise Edition. Библиотека Spring

Илья и Антон

14. Java. Stream API

«Позволяет представить обработку нашей информации, некоторого потока, в виде последовательности применения функций» - Сердюков.

Начиная с JDK 8 в Java появился новый API – Stream API (Application Programming Interface). Задача которого — упростить работу с наборами данных, в частности, упростить операции фильтрации, сортировки и другие манипуляции с данными. Вся основная функциональность данного API сосредоточена в пакете java.util.stream.

Применительно к Stream API *поток* представляет канал передачи данных из источника данных. Причем в качестве источника могут выступать как файлы, так и массивы и коллекции.

Одной из отличительных черт Stream API является применение лямбда-выражений, которые позволяют значительно сократить запись выполняемых действий.

Пример.

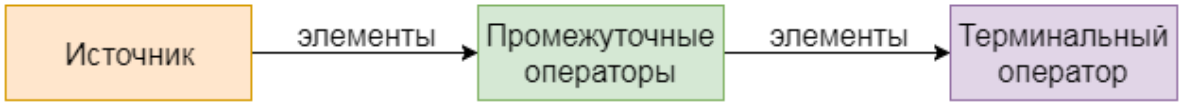
- с потоком:

```
IntStream.of(50, 60, 70, 80, 90, 100, 110, 120).filter(x -> x < 90).map(x -> x + 10)
.limit(3).forEach(System.out::print);
```

- без потока:

```
int[] arr = {50, 60, 70, 80, 90, 100, 110, 120
    int count = 0;
    for (int x : arr) {
        if (x >= 90) continue;
        x += 10;
        count++;
        if (count > 3) break;
        System.out.print(x);
    }
```

Общая схема:



Способы создания потоков.

Получение объекта Stream (источник):

- Пустой поток: Stream.empty().

- Поток из List: list.stream().
- Поток из Map: map.entrySet().stream().
- Поток из массива: Arrays.stream(array).
- Поток из указанных элементов: Stream.of("1", "2", "3").

Классификация Stream

Потоки бывают:

- **Последовательными** (sequential) — выполняются только в текущем потоке.
- **Параллельными** (parallel) — элементы разбиваются (если это возможно) на несколько групп и обрабатываются в каждом потоке отдельно. Затем на нужном этапе группы объединяются в одну для предоставления конечного результата. Чтобы получить параллельный стрим, нужно либо вызвать метод parallelStream() вместо stream(), либо превратить обычный стрим в параллельный, вызвав промежуточный оператор parallel.

Кроме объектных потоков Stream<T>, существуют специальные потоки для примитивных типов:

- IntStream для int.
- LongStream для long.
- DoubleStream для double.

Операторы класса Stream

Есть такое понятие как операторы (по сути методы класса Stream). Операторы можно разделить на две группы:

- **Промежуточные** (“intermediate”, ещё называют “lazy”) — обрабатывают поступающие элементы и возвращают стрим. Промежуточных операторов в цепочке обработки элементов может быть много.
- **Терминальные** (“terminal”, ещё называют “eager”) — обрабатывают элементы и завершают работу стрима, так что терминальный оператор в цепочке может быть только один.

У стримов есть некоторые особенности:

- Обработка не начнётся до тех пор, пока не будет вызван терминальный оператор.
- Стрим после обработки нельзя переиспользовать.

Как работает:

```
IntStream.of(120, 410, 85, 32, 314, 12)
    .filter(x->x<300)
    .map(x->x+11)
    .limit(3)
    .forEach(System.out::print);
```

Как только появился терминальный оператор forEach, он стал запрашивать элементы у стоящего перед ним оператора limit. Тот в свою очередь обращается к map, map к filter, а filter уже обращается к источнику. Затем элементы поступают в прямом порядке: источник, filter, map, limit и forEach.

Промежуточные операторы:

- ***filter(Predicate predicate)*** — фильтрует стрим, пропуская только те элементы, что проходят по условию (Predicate встроенный функциональный интерфейс, добавленный в Java SE 8 в пакет java.util.function. проверяет значение на “true” и “false”).

```
Stream.of(1, 2, 3)
    .filter(x -> x == 10)
    .forEach(System.out::print);
// Вывода нет, так как после фильтрации стрим станет пустым

Stream.of(120, 410, 85, 32, 314, 12)
    .filter(x -> x > 100)
    .forEach(System.out::println);
// 120, 410, 314
```

- ***map(Function mapper)*** — применяет функцию к каждому элементу и затем возвращает стрим, в котором элементами будут результаты функции. map можно применять для изменения типа элементов. (Функциональный интерфейс Function<T,R> представляет функцию перехода от объекта типа T к объекту типа R).

```
Stream.of("3", "4", "5")
    .map(Integer::parseInt)
    .map(x -> x + 10)
    .forEach(System.out::println);
// 13, 14, 15

Stream.of(120, 410, 85, 32, 314, 12)
    .map(x -> x + 11)
    .forEach(System.out::println);
// 131, 421, 96, 43, 325, 23
```

- ***flatMap(Function<T, Stream<R>> mapper)*** — как и в случае с map, служат для преобразования в примитивный стрим, но с одним отличием — можно преобразовать один элемент в ноль, один или множество других.

```
Stream.of(2, 3, 0, 1, 3)
    .flatMap(x -> IntStream.range(0, x))
    .forEach(System.out::println);
// 0, 1, 0, 1, 2, 0, 0, 1, 2
```

- ***distinct()*** – убирает повторяющиеся элементы и возвращаем стрим с уникальными элементами. Как и в случае с sorted, смотрит, состоит ли уже стрим из уникальных элементов и если это не так, отбирает уникальные и помечает стрим как содержащий уникальные элементы.

```
Stream.of(2, 1, 8, 1, 3, 2)
    .distinct()
    .forEach(System.out::println);
// 2, 1, 8, 3
```

- ***skip(long n)*** – пропускает n элементов стрима.

```
Stream.of(5, 10)
    .skip(40)
    .forEach(System.out::println);
// Вывода нет

Stream.of(120, 410, 85, 32, 314, 12)
    .skip(2)
    .forEach(System.out::println);
// 85, 32, 314, 12
```

При работе например с массивом стримов (массивов, списков и так далее) преобразует их в один стрим (массив, список и так далее): [stream1, stream2, stream3, stream4] => stream.

Терминальные операторы:

- ***void forEach(Consumer action)*** — выполняет указанное действие для каждого элемента стрима.

```
Stream.of(120, 410, 85, 32, 314, 12)
    .forEach(x -> System.out.format("%s, ", x));
// 120, 410, 85, 32, 314, 12
```

- ***long count()*** — возвращает количество элементов стрима.

```
long count = IntStream.range(0, 10)
    .flatMap(x -> IntStream.range(0, x))
    .count();
System.out.println(count);
// 45
```

- ***R collect(Collector collector)*** — с его помощью можно собрать все элементы в список, множество или другую коллекцию, сгруппировать элементы по какому-нибудь критерию, объединить всё в строку и т.д.

```
List<Integer> list = Stream.of(1, 2, 3)
    .collect(Collectors.toList());
// list: [1, 2, 3]

String s = Stream.of(1, 2, 3)
    .map(String::valueOf)
    .collect(Collectors.joining("-", "<", ">"));
// s: "<1-2-3>"
```

- ***T reduce(T identity, BinaryOperator accumulator)*** — позволяет преобразовать все элементы стрима в один объект. Например, посчитать сумму всех элементов, либо найти минимальный элемент. Сперва берётся объект identity и первый элемент стрима, применяется функция accumulator и identity становится её результатом. Затем всё продолжается для остальных элементов.

```
int sum = Stream.of(1, 2, 3, 4, 5)
    .reduce(10, (acc, x) -> acc + x);
// sum: 25
```

- ***Optional min(Comparator comparator)*** / ***Optional max(Comparator comparator)*** — поиск минимального/максимального элемента, основываясь на переданном компараторе.

```
int min = Stream.of(20, 11, 45, 78, 13)
    .min(Integer::compare).get();
// min: 11

int max = Stream.of(20, 11, 45, 78, 13)
    .max(Integer::compare).get();
// max: 78
```

- ***Optional findFirst()*** — гарантированно возвращает первый элемент стрима, даже если стрим параллельный.

```
int firstSeq = IntStream.range(4, 65536)
    .findFirst()
    .getAsInt();
// firstSeq: 4
```

- ***boolean allMatch(Predicate predicate)*** — возвращает true, если все элементы стрима удовлетворяют условию predicate. Если встречается какой-либо элемент, для которого результат вызова функции-предиката будет false, то оператор перестаёт просматривать элементы и возвращает false.

```
boolean result = Stream.of(1, 2, 3, 4, 5)
    .allMatch(x -> x <= 7);
// result: true
```

- ***boolean anyMatch(Predicate predicate)*** — возвращает true, если хотя бы один элемент стрима удовлетворяет условию predicate. Если такой элемент встретился, нет смысла продолжать перебор элементов, поэтому сразу возвращается результат.

```
boolean result = Stream.of(1, 2, 3, 4, 5)
    .anyMatch(x -> x == 3);
// result: true
```

- ***boolean noneMatch(Predicate predicate)*** — возвращает true, если, пройдя все элементы стрима, ни один не удовлетворил условию predicate. Если встречается какой-либо элемент, для которого результат вызова функции-предиката будет true, то оператор перестаёт перебирать элементы и возвращает false.

```
boolean result = Stream.of(1, 2, 3, 4, 5)
    .noneMatch(x -> x == 9);
// result: true
```

Методы Collectors

- ***toList()*** — собирает элементы в List.
- ***ToSet()*** — собирает элементы в множество.
- ***joining(CharSequence delimiter)*** — собирает элементы, реализующие интерфейс CharSequence, в единую строку.
- ***summingInt(ToIntFunction mapper)*** / ***summingLong(ToLongFunction mapper)*** / ***summingDouble(ToDoubleFunction mapper)*** — коллектор, который преобразовывает объекты в int/long/double и подсчитывает сумму.
- ***counting()*** — подсчитывает количество элементов.

Использованные источники

1. Статья (annimon.com) - Полное руководство по Java 8 Stream API.
2. Статья (metanit.com) - Введение в Stream API.
3. Сердюков Р.Е. - Stream API (лекция).

15. Java. Reflection API

Рефлексия — это механизм исследования данных о программе во время её выполнения. Рефлексия позволяет исследовать информацию о полях, методах и конструкторах классов. Сам же механизм рефлексии позволяет обрабатывать типы, отсутствующие при компиляции, но появившиеся во время выполнения программы.

Что позволяет рефлексия:

- Узнать/определить класс объекта.
- Получить информацию о модификаторах класса, полях, методах, константах, конструкторах и суперклассах.
- Выяснить, какие методы принадлежат реализуемому интерфейсу/интерфейсам.
- Создать экземпляр класса, причем имя класса неизвестно до момента выполнения программы.
- Получить и установить значение поля объекта по имени.
- Вызвать метод объекта по имени.
- Получать доступ ко всем переменным и методам, в том числе приватным.
- Преобразовывать классы одного типа в другой (cast).
- Делать все это во время исполнения программы (динамически, в Runtime).

Работа с классами

Объект java.lang.Class является точкой входа для всех операций рефлексии. Для каждого типа объекта, JVM создает неизменяемый экземпляр java.lang.Class, который предоставляет методы для получения свойств объекта, создания новых объектов, вызова методов.

Все типы в Java, включая примитивные типы и массивы имеют связанный с ними java.lang.Class объект.

- Получение объекта, если известно название класса во время компиляции:

```
Class mClassObject = SomeObject.class;
```

- Получение объекта, если не известно название класса во время компиляции, но известно во время выполнения:

```
Class mClassObject = Class.forName("имя_класса");
```

При использовании метода *Class.forName()* мы должны указать полное имя класса. То есть имя класса, включая все имена пакетов.

- Метод ***getName()*** – вернет полное имя класса с пакетом.
- Метода ***getSimpleName()*** – вернет только название класса без имени пакета.
- Метод ***getModifiers()*** – получение модификаторов.

Получение информации о пакете:

```
Class mClassObject = SomeObject.class;
Package package = mClassObject.getPackage();
```

Получение суперкласса:

```
Class mClassObject = SomeObject.class;
Class superclass = mClassObject.getSuperclass();
```

Получение списка интерфейсов:

```
Class mClassObject = SomeObject.class;
```

Поля

Используя рефлексию мы можем работать с полями — переменными-членами класса. В этом нам помогает Java класс java.lang.reflect.Field — с помощью него в рантайме мы можем устанавливать значения и получать данные с полей.

Получение поля класса:

```
Class mClassObject = SomeObject.class;
Field[] fields = mClassObject.getFields();
```

Получение поля класса, если известно имя поля:

```
Class mClassObject = SomeObject.class;
Field field = mClassObject.getField("fieldName");
```

Получение названия поля:

```
Field field = mClassObject.getField("fieldName");
String fieldName = field.getName();
```

Получение типа поля:

```
Field field = mClassObject.getField("fieldName");
```

```
Object fieldType = field.getType();
```

Получение и установка значения полей:

```
Class mClassObject = SomeObject.class

Field field = mClassObject.getField("fieldName");

SomeObject instance = new SomeObject();

Object value = field.get(instance);

field.set(instance, value);
```

Параметр *instance*, который передается в методы для получения и установки значения поля, должен быть экземпляром класса, которому принадлежит само поле. В приведенном выше примере используется экземпляр класса SomeObject, потому что поле fieldName является членом экземпляра этого класса.

Методы

Используя рефлексию в Java мы можем получать информацию о методах и вызывать их в рантайме. Для этого используется класс java.lang.reflect.Method.

Получение всех методов класса:

```
Class mClassObject = SomeObject.class

Method[] methods = mClassObject.getMethods();
```

Получение метода, если нам известны точные типы параметров метода, который мы хотим использовать:

```
Class mClassObject = SomeObject.class;

Method method = mClassObject.getMethod("sayHello", new Class[]{String.class});
```

Или **если метод** sayHello() **без параметров**:

```
Method method = mClassObject.getMethod("sayHello", null);
```

Получение параметров метода:

```
Class[] parameterTypes = method.getParameterTypes();
```

Получение типа возвращаемого значения:

```
Class returnType = method.getReturnType();
```

Вызов метода с помощью Java рефлексии:

```
Class mClassObject = SomeObject.class

Method method = mClassObject.getDeclaredMethod("sayHello", parameterTypes);
```

method.invoke(objectToInvokeOn, params);

objectToInvokeOn – имеет тип Object и является объектом, для которого мы хотим вызвать метод sayHello().

parameterTypes — имеет тип Class[] и представляет собой параметры, которые метод принимает на вход.

Params – имеет тип Object[] и представляет собой пераметры, которые переданы методы.

invoke(Object, Args) – для вызова объекта Method.

Методы установки и получения значения:

```
public static void printGettersOrSetters(Class aClass){
  Method[] methods = aClass.getMethods();

  for(Method method : methods){
    if(isGetter(method)) System.out.println("getter: " + method);
    if(isSetter(method)) System.out.println("setter: " + method);
  }
}

public static boolean isGetter(Method method){
  if (!method.getName().startsWith("get")) {
       return false;
  }
  if (method.getParameterTypes().length != 0) {
       return false;
  }
  if (void.class.equals(method.getReturnType()) {
       return false;
  }
  return true;
}

public static boolean isSetter(Method method){
  if (!method.getName().startsWith("set")) {
       return false;
  }
  if (method.getParameterTypes().length != 1) {
       return false;
  }
  return true;
}
```

Приватные поля и методы

Доступ к private полям с помощью рефлексии.

Методы Class.getDeclaredField(String name) или Class.getDeclaredFields() (методы Class.getField(String name) и Class.getFields() возвращают только public-поля)

Пример:

```
public class PrivateClass {

  private String mPrivateString = null;

  public PrivateClass(String privateString) {
    this.mPrivateString = privateString;
  }
}
```

```
PrivateClass privateObject = new PrivateClass("Значение приватного поля");

Field privateStringField = PrivateClass.class.
            getDeclaredField("mPrivateString");

privateStringField.setAccessible(true);

String fieldValue = (String) privateStringField.get(privateObject);
System.out.println("значение приватного поля = " + fieldValue);
```

getDeclaredField(String) – имеет доступ лишь к полям, объявленным в конкретном классе и не имеет доступ к полям суперкласса.

Field.setAcessible(true) – отключает проверку доступа для указанного поля.

Доступ к приватным методам с помощью рефлексии.

Методы Class.getDeclaredMethod(String name, Class[] parameterTypes) или Class.getDeclaredMethods() (Class.getMethod(String name, Class[] parameterTypes) и Class.getMethods() имеют доступ лишь к public-методам.)

Пример:

```
public class PrivateClass {

  private String mPrivateString = null;

  public PrivateClass(String privateString) {
    this.mPrivateString = privateString;
  }
  private String getPrivateString(){
    return this.mPrivateString;
  }
```

```
PrivateClass privateObject = new PrivateClass("Какое-то значение");

Method privateStringMethod = PrivateClass.class.
        getDeclaredMethod("getPrivateString", null);

privateStringMethod.setAccessible(true);

String returnValue = (String)
        privateStringMethod.invoke(privateObject, null);

System.out.println("значение, которое возвращает private метод = " + returnValue);
```

Аннотации

Пример:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Reflectable {
    public String name();
    public String value();
}
```

@Retention(RetentionPolicy.RUNTIME) и @Target(ElementType.TYPE) – указывают как будет использоваться пользовательская аннотация.

@Retention(RetentionPolicy.RUNTIME) – указала на то, что аннотация будет использована в рантайме, следовательно мы сможем получить к ней доступ с помощью рефлексии.

@Target(ElementType.TYPE) – указывает на то, что мы можем использовать аннотацию на интерфейсах и классах.

Принцип:

1. «Повесим» нашу аннотацию на какой-то класс:

```
@Reflectable(name="reflectable",  value = "какие-то метаданные")
public class SimpleClass {
}
```

```
Class aClass = SimpleClass.class;
Annotation[] annotations = aClass.getAnnotations();

for (Annotation annotation : annotations) {
    if(annotation instanceof Reflectable) {
        Reflectable mAnnotation = (Reflectable) annotation;
        System.out.println("name: " + mAnnotation.name());
        System.out.println("value: " + mAnnotation.value());
    }
}
```

Метод ***getAnnotations()*** – для получения всех аннотаций класса в виде массива объектов Annotation. В цикле по всем аннотациям мы ищем пользовательскую аннотацию — Reflectable и получаем информацию о ней.

Если же нам интересна конкретная аннотация, то получить к ней доступ можно следующим образом:

```
Class aClass = SimpleClass.class;
Annotation annotation = aClass.getAnnotation(Reflectable.class);
```

```
public class SimpleClass {
  @Reflectable(name="reflectable",  value = "какие-то метаданные")
  public void sayHello(){}
}
```

```
Class mClassObject = SimpleClass.class
Method method =
    mClassObject.getMethod("sayHello", null);

Annotation[] annotations = method.getDeclaredAnnotations();

for(Annotation annotation : annotations) {
    if(annotation instanceof Reflectable) {
        Reflectable mAnnotation = (Reflectable) annotation;
        System.out.println("name: " + mAnnotation.name());
        System.out.println("value: " + mAnnotation.value());
    }
}
```

2. Работа с аннотациями на методах:

   Явно получаем с указанного метода:

```
Annotation annotation = method.getAnnotation(Reflectable.class);
```

3. Аннотации на параметрах метода:

Давайте добавим в наш класс еще один метод sayBye() с одним параметром, помеченным аннотацией:

```
public class SimpleClass {
  //...
  public static void sayBye(
        @Reflectable(name="reflectable", value="какие-то метаданные") String param){
  }
}
```

```
Class mClassObject = SimpleClass.class
Method method =
    mClassObject.getMethod("sayBye", new Class[]{String.class});

Annotation[][] paramAnnotations = method.getParameterAnnotations();
Class[] paramTypes = method.getParameterTypes();

int i = 0;
for (Annotation[] annotations : paramAnnotations){
  Class parameterType = paramTypes[i++];

  for (Annotation annotation : annotations) {
    if (annotation instanceof Reflectable) {
        Reflectable mAnnotation = (Reflectable) annotation;
        System.out.println("param: " + paramType.getName());
        System.out.println("name: " + mAnnotation.name());
        System.out.println("value: " + mAnnotation.value());
    }
  }
}
```

Получили двумерный массив аннотаций на методе: внутренний массив каждого элемента является набором аннотаций для каждого параметра (аргумента метода).

4. Аннотации на полях

```
public class SimpleClass {
  @Reflectable(name="reflectable",  value = "какие-то метаданные")
  public String mField = null;
}
```

```
Class mClassObject = SimpleClass.class
Field field = mClassObject.getField("mField");
Annotation annotation = field.getAnnotation(Reflectable.class);
```

Использованные источники.

1. Полное руководство по Java Reflection API. Рефлексия на примерах.

16. Java. Аннотации

Аннотация (Annotation) в языке Java – это специальная форма синтетических метаданных, которая может быть добавлена в исходный код. Аннотации используются для анализа кода, компиляции или выполнения. Аннотированы могут быть пакеты, классы, методы, переменные и параметры.

Функции:

- предоставляет необходимую информацию для компилятора.
- предоставляет информацию различным инструментам для генерации другого кода, конфигураций и т. д.
- может быть использована во время работы кода.

Виды:

- Инструкция для компилятора (Compiler).
- Инструкция во время построения (Build-time).
- Инструкция во время запуска (Runtime).

Встроенные аннотации

Встроенные аннотации отслеживаются средой разработки IDE и применяются к java-коду метода.

@Override	Проверка переопределения метода. IDE вызывает предупреждение компиляции, если метод не найден в родительском классе.
@Deprecated	IDE отмечает, что метод устарел и вызывает предупреждение компиляции, если метод используется.
@SuppressWarnings	Аннотация указывает IDE подавить предупреждения компиляции.

Мета-аннотации

Мета-аннотации – аннотации, применяемые к другим аннотациям:

@Retention	Определяет, как отмеченная аннотация будет храниться — в коде, в скомпилированном классе или во время работы кода.
@Documented	Отмечает аннотацию для включения в документацию.

@Target	Отмечает аннотацию как ограничивающую, какие элементы java-аннотации могут быть к ней применены.
@Inherited	Отмечает, что аннотация может быть расширенна подклассами аннотируемого класса.

Параметры аннотаций

@Retention

RetentionPolicy.SOURCE	аннотация используется на этапе компиляции и должна отбрасываться компилятором
RetentionPolicy.CLASS	аннтоация будет записана в class-файл компилятором, но не должна быть доступна во время выполнения (runtime)
RetentionPolicy.RUNTIME	аннотация будет записана в class-файл и доступна во время выполнения через reflection

@Target

@Target(ElementType.PACKAGE)	только для пакетов
@Target(ElementType.TYPE)	только для классов
@Target(ElementType.CONSTRUCTOR)	только для конструкторов
@Target(ElementType.METHOD)	только для методов
@Target(ElementType.FIELD)	только для атрибутов(переменных) класса
@Target(ElementType.PARAMATER)	только для параметров метода
@Target(ElementType.LOCAL_VARIABLE)	только для локальных переменных

В случае, если необходимо, что бы аннотация использовалась больше чем для одного типа параметров, то можно указать @Target следующим образом:

@Target({ ElementType.PARAMETER, ElementType.LOCAL_VARIABLE })

@Documented

Пример, класс, помеченный аннотацией без @Documented, будет выглядеть так:

public class TestClass extends java.lang.Object

А если в описание аннотации добавить @Documented, получим:

@ControlledObject(name="name")

public class TestClass extends java.lang.Object

Синтаксис

Аннотация задается описанием соответствующего интерфейса.

```
import java.lang.annotation.*;
@Target(value=ElementType.FIELD)
@Retention(value= RetentionPolicy.RUNTIME)
public @interface Name {
     String name();
     String type() default  “string”;
}
```

Как видно из примера, аннотация определяется описанием с ключевым словом ***@interface*** и может включать в себя несколько полей, которые можно задать как обязательными, так и не обязательными.

- Методы в пользовательской аннотации должны быть без параметров.
- Методы в пользовательской аннотации могут возвращать лишь примитивы, String, Перечисления (Enums), аннотации.
- Методы в пользовательской аннотации могут иметь значения по умолчанию.
- Аннотации могут иметь мета-аннотации, прикрепленные к ним. Мета аннотации используются для предоставления информации об аннотации

Примеры использования

1. Пример аннотации:

```java
package ua.com.prologistic;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Documented
@Target(ElementType.METHOD)
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface MethodInfo{
    String author() default "Andrew";
    String date();
    int revision() default 1;
    String comments();
}
```

Использование встроенных в Java аннотаций, а также использование аннотации, созданной нами в примере выше:

```java
package ua.com.prologistic;

import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.List;

public class AnnotationExampleClass {

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

    @Override
    @MethodInfo(author = "Andrew", comments = "Main method", date = "Aug 10 2015", revision = 1)
    public String toString() {
        return "Переопределили метод toString() ";
    }

    @Deprecated
    @MethodInfo(comments = "устаревший метод", date = "Aug 10 2015")
    public static void oldMethod() {
        System.out.println("Этот метод не стоит дальше использовать");
    }

    @SuppressWarnings({ "unchecked", "deprecation" })
    @MethodInfo(author = "Andrew", comments = "Main method", date = "Aug 10 2015", revision = 4)
    public static void genericsTest() throws FileNotFoundException {
        List l = new ArrayList();
        l.add("фыва");
        oldMethod();
    }

}
```

2. Представим себе, что у нас есть какой-то самодельный проект, который на вход получает класс, специально заанотированный, чтобы проект мог управлять жизненным циклом объектов этого класса, и пусть там будут аннотации StartObject, StopObject для описания методов класса, и ControlledObject для описания самого класса. Последней аннотации дадим еще поле name, путь там хранится якобы имя для поиска.

```
@Target(value=ElementType.METHOD)
@Retention(value= RetentionPolicy.RUNTIME)
public @interface StartObject {
}

@Target(value=ElementType.METHOD)
@Retention(value= RetentionPolicy.RUNTIME)
public @interface StopObject {
}

@Target(value=ElementType.TYPE)
@Retention(value= RetentionPolicy.RUNTIME)
public @interface ControlledObject {
     String name();
}
```

Напишем модуль, проверяющий подходит ли класс для загрузки в наш гипотетический проект или нет. Сперва определим сам проверяемый класс.

```
@ControlledObject(name="biscuits")
public class Cookies {

     @StartObject
     public void createCookie(){
       //бизнес логика
     }
     @StopObject
     public void stopCookie(){
       //бизнес логика
     }
}
```

Для того, чтобы работать с классом, сначала необходимо загрузить класс в контекст приложения. Используем:

Class cl = Class.forName("org.annotate.test.classes.Cookies");

Далее, через механизм reflection мы получаем доступ к полям и аннотациям класса.Проверим наличие аннотированных методов в классе и аннотации на самом классе:

```
if(!cl.isAnnotationPresent(ControlledObject.class)){
       System.err.println("no annotation");
       } else {
          System.out.println("class annotated ; name  -  " + cl.getAnnotation(Contr
olledObject.class));
       }
boolean hasStart=false;
boolean hasStop=false;
Method[] method = cl.getMethods();
for(Method md: method){
       if(md.isAnnotationPresent(StartObject.class)) {hasStart=true;}
       if(md.isAnnotationPresent(StopObject.class)) {hasStop=true;}
       if(hasStart && hasStop){break;}
}
System.out.println("Start annotaton  - " + hasStart + ";  Stop annotation  - " + ha
sStop );
```

Запустив, на выходе мы получим:

Start annotaton — true;

Stop annotation — true.

Если попробовать убрать одну из аннотаций, то вывод сообщит о несоответствии требованиям.

Использованные источники

1. Руководство по аннотациям в Java.
2. Статья habr.com - Аннотации в JAVA: обзор синтаксиса и создание собственных.
3. Аннотация методов, annotation.

17. Java. Обработка исключений

Нередко в процессе выполнения программы могут возникать ошибки, при том необязательно по вине разработчика. Некоторые из них трудно предусмотреть или предвидеть, а иногда и вовсе невозможно. Подобные ситуации называются исключениями.

В языке Java предусмотрены специальные средства для обработки подобных ситуаций. Одним из таких средств является конструкция *try...catch*. При использовании блока *try...catch* вначале выполняются все инструкции между операторами *try* и *catch*. Если в блоке *try* вдруг возникает исключение, то обычный порядок выполнения останавливается и переходит к инструкции *catch,* которая может обработать данное исключение. Если такого блока не найдено, то пользователю отображается сообщение о необработанном исключении, а дальнейшее выполнение программы останавливается. И чтобы подобной остановки не произошло, и надо использовать блок *try...catch*.

Выражение *catch* имеет следующий синтаксис: *catch*(тип_исключения имя_переменной). Но если возникшее исключение не является исключением типа, указанного в инструкции *catch*, то оно не обрабатывается, а программа просто зависает или выбрасывает сообщение об ошибке.

Конструкция *try...catch* также может иметь блок *finally*. Однако этот блок необязательный, и его можно при обработке исключений опускать. Блок *finally* выполняется в любом случае, возникло ли исключение в блоке *try* или нет.

Ключевые слова

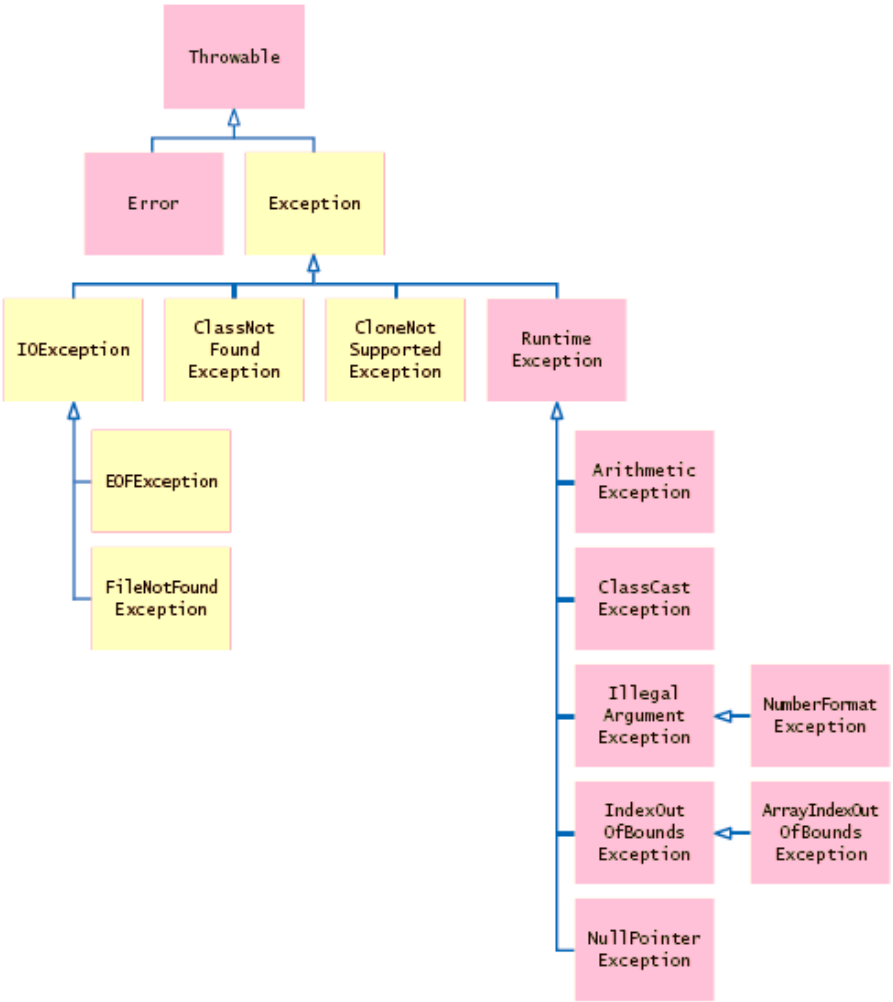
Обработка исключений в Java основана на использовании в программе следующих ключевых слов:

- *try* – определяет блок кода, в котором может произойти исключение.
- *catch* – определяет блок кода, в котором происходит обработка исключения (таких блоков может быть несколько, а после их завершения, программа продолжает свою работу, выполняя все остальные инструкции после блока/блоков *catch*).
- *finally* – определяет блок кода, который является необязательным, но при его наличии выполняется в любом случае независимо от результатов выполнения блока *try*.
- *throw* – используется для возбуждения исключения (то есть с помощью этого оператора мы сами можем создать исключение и вызвать его в процессе выполнения).

- *throws* – используется в сигнатуре методов для предупреждения, о том что метод может выбросить исключение (если метод генерирует исключение, то при вызове его можно обработать или пробросить дальше с помощью этого оператора).

Иерархия исключений Java

Базовым классом для всех исключений является класс Throwable. От него уже наследуются два класса: Error и Exception. Все остальные классы являются производными от этих двух классов.



Класс Error описывает внутренние ошибки в исполняющей среде Java, исключения наследуются от класса Exception. Среди этих исключений следует выделить класс RuntimeException: RuntimeException является базовым классом для группы непроверяемых исключений (unchecked exceptions) — компилятор не проверяет факт обработки таких исключений и их можно не указывать вместе с оператором *throws* в объявлении метода. Такие исключения являются следствием ошибок разработчика (например неверное преобразование типов или выход за пределы массива).

Все остальные классы, образованные от класса Exception, называются проверяемыми исключениями (checked exceptions). Подобные исключения обрабатываются с помощью конструкции *try...catch* (либо можно передать обработку методу, который будет вызывать данный метод).

Обработка иерархии исключений.

Предположим есть метод, который может выбросить несколько исключений, находящихся в одной иерархии (например IOException и FileNotFoundException). Общее правило – обрабатывать исключения нужно от младшего к старшему, т.е. нельзя поставить в первый блок *catch(Exception e)*, иначе все дальнейшие блоки *catch* уже ничего не смогут обработать, т.к. любое исключение будет попадать под Exception. Таким образом сначала нужно обработать FileNotFoundException (т.к. он наследует IOException), а затем уже IOException.

Методы класса Exception.

Поскольку все классы исключений наследуются от класса Exception, то все они наследуют ряд его методов, которые позволяют получить информацию о характере исключения. Среди этих методов наиболее важными являются:

- ***getMessage***() – возвращает сообщение об исключении.
- ***petStackTrace***() – возвращает массив, содержащий трассировку стека исключения.
- ***printStackTrace***() – отображает трассировку стека.

Создание своих классов исключений.

Хотя имеющиеся в стандартной библиотеке классов Java классы исключений описывают большинство исключительных ситуаций, которые могут возникнуть при выполнении программы, все таки иногда требуется создать свои собственные классы исключений со своей логикой.

Чтобы создать свой класс исключений, надо унаследовать его от класса Exception (либо от любого из наследников). Программист сам решает: делать свое исключение проверяемым или нет. Для того чтобы получить unchecked exception следует наследоваться от соответствующих классов (например RuntimeException).

Блок try-with-resources

Начиная с седьмой версии Java предлагает улучшенное управление ресурсами, которые должны быть закрыты после окончания работы с ними. К таким

ресурсам относятся, например, файлы потоки, соединения с базами данных и сокетами. Этой цели служит специальная языковая конструкция try-with-resources. Для того чтобы это автоматическое закрытие работало создан специальный интерфейс AutoCloseable.

В Java 7 все классы ресурсов реализуют этот интерфейс. Интерфейс объявляет метод close(), который автоматически вызывается для объектов, обслуживаемых конструкцией try-with-resources.

Пример использования конструкции:

```
try (InputStream is = new FileInputStream("a.txt")) {
    readFromInputStream(is);
}
```

Пример генерации и обработки исключений

```
public void testMethod(int a) throws Exception {
    if (a < 0) throw new Exception();
}

public void method(int a) {
    try {
        testMethod(a);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Использованные источники

1. Статья metanit.com — Введение в обработку исключений.
2. Статья metanit.com — Оператор throws.
3. Статья metanit.com — Классы исключений.
4. Статья metanit.com — Создание своих классов исключений.
5. Статья javarush.ru — Исключения в Java.
6. Статья javarush.ru — Java 7 try-with-resources.
7. Собеседование по Java – исключения (exceptions).

18. **Java. Threads. Создание и управление**

Стас

19. Java. Threads. Синхронизация

Все потоки, принадлежащие одному процессу, разделяют некоторые общие ресурсы (адресное пространство, открытые файлы). Что произойдет, если один поток еще не закончил работать с каким-либо общим ресурсом, а система переключилась на другой поток, использующий тот же ресурс?

Когда два или более потоков имеют доступ к одному разделенному ресурсу, они нуждаются в обеспечении того, что ресурс будет использован только одним потоком одновременно. Процесс, с помощью которого это достигается, называется **синхронизацией.**

Монитор

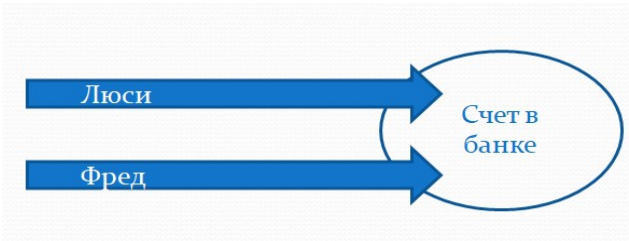
Каждый объект в Java имеет ассоциированный с ним монитор. Монитор — это объект, используемый в качестве взаимоисключающей блокировки. Только один поток исполнения может в одно, и то же время владеть монитором. Все другие потоки исполнения, пытающиеся войти в заблокированный монитор, будут приостановлены до тех пор, пока первый поток не выйдет из монитора.

Взаимодействие потока и монитора:

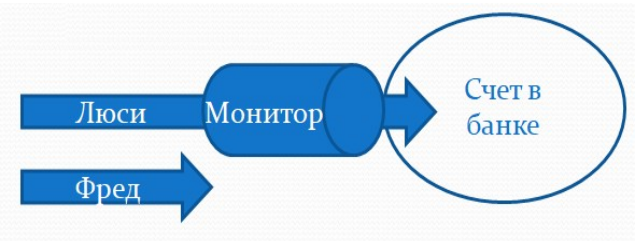
- Если поток засыпает, то он удерживает монитор.
- Поток может захватить сразу несколько мониторов.

Рассмотрим разницу между доступом к объекту без синхронизации и из синхронизированного кода. Доступ к банковскому счету:

без синхронизации:



с синхронизацией:



Способы синхронизации кода

Синхронизировать прикладной код можно двумя способами:

- С помощью синхронизированных методов. Метод объявляется с использованием ключевого слова ***synchronized:***

```
public synchronized void someMethod(){}
```

- Заключить вызовы методов в блок оператора ***synchronized*:**

```
synchronized(объект) {
  // операторы, подлежащие синхронизации
}
```

Смысл прост. Если один поток зашел внутрь блока кода, который помечен словом ***synchronized***, он моментально захватывает монитор объекта, и все другие потоки, которые попытаются зайти в этот же блок или метод вынуждены ждать, пока предыдущий поток не завершит свою работу и не освободит монитор.

Синхронизация статических методов

Статические методы тоже могут быть синхронизированы с помощью ключевого слова ***synchronized***.

Для синхронизации статических методов используется один монитор для одного класса. Каждый загруженный в Java класс имеет соответствующий объект класса Class, представляющий этот класс. Монитор именно этого объекта используется для синхронизации статических методов (если они синхронизированы).

```
public static int getCount() {
    synchronized(MyClass.class) {
        return count;
    }
}
```

```
public static synchronized int getCount() {
    return count;
}
```

Методы и состояния блокировки

Освобождает монитор	Удерживает монитор	Класс определяющий метод
wait()	*notify()*	*java.lang.Object*
	join()	*java.lang.Thread*
	sleep()	*java.lang.Thread*
	yield()	*java.lang.Thread*

Методы класса Object:

- ***wait()*** — освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify().
- ***notify()*** **—** продолжает работу потока, у которого ранее был вызван метод wait().
- ***notifyAll()*** **—** возобновляет работу всех потоков, у которых ранее был вызван метод wait().

Методы класса Thread:

- В Java предусмотрен механизм, позволяющий одному потоку ждать завершения выполнения другого. Для этого используется метод ***join()*.**
- ***Thread.sleep()*** — статический метод класса Thread, который приостанавливает выполнение потока, в котором он был вызван.
- ***Thread.yield()* —** статический метод класса Thread, который заставляет процессор переключиться на обработку других потоков системы.

Приоритеты потоков

Каждый поток в системе имеет свой приоритет. Приоритет – это некоторое число в объекте потока, более высокое значение которого означает больший приоритет. Система в первую очередь выполняет потоки с большим приоритетом, а потоки с меньшим приоритетом получают процессорное время только тогда, когда их более привилегированные собратья простаивают.

Работать с приоритетами потока можно с помощью двух функций:

- ***void setPriority(int priority)*** – устанавливает приоритет потока.

  Возможные значения priority:

  - MIN_PRIORITY
  - NORM_PRIORITY
  - MAX_PRIORITY.
- ***int getPriority()*** – получает приоритет потока.

Проблемы при синхронизации потоков

Особый вид трудноотлавливаемых ошибок программирования связан с взаимными блокировками, иначе называемых deadlock'ами. Возникают deadlock'и между двумя потоками, когда каждый из потоков захватывает один объект и начинают выполнять синхронизируемый блок кода, внутри которого обращается к другому объекту, захваченному параллельным потоком. В этом случае оба потока переходят в режим ожидания освобождения объектов, блокированных параллельными потоками.

Голодание потоков. В Java имеется планировщик потоков (Thread Scheduler), который контролирует все запущенные потоки и решает, какие потоки должны быть запущены и какая строка кода должна выполняться. Решение основывается на приоритете потока. Поэтому потоки с меньшим приоритетом получают меньше процессорного времени по сравнению с потоками с бóльшим

приоритет. Данное разумное решение может стать причиной проблем при злоупотреблении. То есть, если бóльшую часть времени исполняются потоки с высоким приоритетом, то низкоприоритетные потоки начинают «голодать», поскольку не получают достаточно времени для того, чтобы выполнить свою работу должным образом.

Пример с «голоданием» может возникнуть от того, что не был определен порядок прохождения потоком блока *synchronized*. Когда несколько параллельных потоков должны выполнить некоторый код, оформленный блоком synchronized, может получиться так, что одним потокам придётся ждать дольше других, прежде чем войти в блок. Теоретически они могут вообще туда не попасть.

Использованные источники

- Статья habr.com - Многопоточность в Java.
- Статья metanit.com - Методы wait и notify.
- Статья examclouds.com - Синхронизация потоков.
- Статья javarush.ru - Синхронизация потоков. Оператор synchronized в Java.
- Вопросы по Java на собеседовании.

20. Java. Асинхронность

Егор

21. Принципы GRASP

GRASP (General Responsibility Assignment Software Patterns – общие шаблоны распределения ответственности) — шаблоны, используемые в объектно-ориентированном проектировании для решения общих задач по назначению ответственности классам и объектам. GRASP паттерны не имеют выраженной структуры, четкой области применения и конкретной решаемой проблемы, а лишь представляют собой обобщенные подходы/рекомендации/принципы, используемые при проектировании дизайна системы.

GRASP состоит из 5 основных и 4 дополнительных шаблонов.

Основные шаблоны

- **Информационный эксперт (Information Expert).** Шаблон определяет базовый принцип распределения ответственности:

  «*Ответственность должна быть назначена тому, кто владеет максимумом необходимой информации для исполнения — информационному эксперту*»

  Проблема: В системе должна аккумулироваться, рассчитываться и т.п. необходимая информация.

  Решение: Назначить обязанность аккумуляции информации, расчета и т.п. некоему классу (информационному эксперту), обладающему необходимой информацией.

  Рекомендации: Информационным экспертом может быть не один класс, а несколько.

- **Создатель (Creator).** Суть ответственности такого объекта в том, что он создает другие объекты. Есть ряд моментов, которые должны выполняться, когда мы наделяем объект ответственностью создателя:
  - Создатель содержит или агрегирует создаваемые объекты.
  - Создатель использует создаваемые объекты.
  - Создатель знает, как проинициализировать создаваемый объект.
  - Создатель записывает создаваемые объекты.
  - Создатель имеет данные инициализации для объекта.

  Проблема: Кто должен отвечать за создание экземпляров класса?

  Решение: Назначить классу B обязанность создавать объекты другого класса A.

Рекомендации: Логично использовать паттерн если класс B содержит, агрегирует, активно использует и т.п. объекты класса A.

- **Контроллер (Controller).** Контроллер призван решить проблему разделения интерфейса и логики в интерактивном приложении. Это не что иное, как контроллер из MVC парадигмы. Контролер отвечает за обработку запросов и решает кому должен делегировать запросы на выполнение.

  Проблема: Кто должен отвечать за обработку входных системных событий?

  Решение: Обязанности по обработке системных сообщений делегируются специальному классу. Контроллер — это объект, который отвечает за обработку системных событий и не относится к интерфейсу пользователя. Контроллер определяет методы для выполнения системных операций.

  Рекомендации: Для различных прецедентов логично использовать разные контроллеры (контроллеры прецедентов) - контроллеры не должны быть перегружены. Внешний контроллер представляет всю систему целиком, его можно использовать, если он будет не слишком перегруженным (то есть, если существует лишь несколько системных событий).

- **Низкая связанность (Low Coupling).** Если объекты в приложении сильно связанны, то любой их изменение приводит к изменениям во всех связанных объектах. А это неудобно и порождает множество проблем. Low coupling как раз говорит о том что необходимо, чтобы код был слабо связан и зависел только от абстракций.

  Проблема: Обеспечить низкую связанность при создании экземпляра класса и связывании его с другим классом.

  Решение: Распределить обязанности между объектами так, чтобы степень связанности оставалась низкой.

- **Высокое зацепление (High Cohesion).** High Cohesion принцип говорит о том, что класс должен стараться выполнять как можно меньше не специфичных для него задач, и иметь вполне определенную область применения.

  Проблема: Необходимо обеспечить выполнение объектами разнородных функций.

  Решение: Обеспечить распределение обязанностей с высоким зацеплением.

Дополнительные шаблоны

- **Чистая выдумка (Pure Fabrication).** Существует понятие модели программирования по предметной области, согласно которой, каждой сущности из предметной области соответствует один или более классов программной среды. При этом, обязанности взаимодействия сущностей, как правило накладываются на них самих. Такой подход имеет очевидный недостаток — высокая связность модулей системы. Шаблон Pure Fabrication позволяет решить данную проблему, путем введения в программную среду дополнительного класса (не отражающего реальной сущности из предметной области) и наделение его требуемыми обязанностями.
- **Посредник (Indirection).** Шаблон Indirection реализует низкую связность между классами, путем назначения обязанностей по их взаимодействию дополнительному объекту — посреднику.
- **Полиморфизм (Polymorphism).** Шаблон Polymorphism позволяет обрабатывать альтернативные варианты поведения на основе типа. При этом, альтернативные реализации приводятся к обобщенному интерфейсу.
- **Устойчивость к изменениям (Protected Variations).** Сущность шаблона Protected Variations заключается в устранении точек неустойчивости, путем определения их в качестве интерфейсов и реализации для них различных вариантов поведения.

Использованные источники

1. Sergey Nemchinskiy — Введение в шаблоны GRASP (видео).
2. Sergey Nemchinskiy — Введение в шаблоны GRASP (презентация).
3. Статья bool.dev – GRASP принципы.
4. Статья habr.com – GRASP паттерны проектирования.
5. Сердюков Р.Е. - GRASP (лекция).

22. Принципы YAGNI, DRY и KISS

Don't Repeat Yourself

DRY (Don’t Repeat Your self) – принцип хорошего кода, который гласит:

«*Каждая часть знаний должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы*»

Второе название — SSOT (Single Source Of Truth).

Антипринцип — WET (Write Everything Twice / We Enjoy Typing).

Основные идеи:

- Если дублировать код, то:
  - Придется поддерживать одну и ту же логику и тестировать код сразу в двух местах.
  - Если меняется код в одном месте, его нужно будет изменить и в другом.
- В большинстве случаев дублирование кода происходит из-за незнания системы.

Когда DRY не работает:

- Многопользовательские игры (некоторый вычисления происходят и на сервере и на клиенте, чтобы оптимизировать задержку получения данных с сервера).
- Браузеры (многие браузеры совершают предзагрузку страниц, на которые вероятно пойдет пользователь, чтобы время отклика было меньше).
- Валидация (проверка) данных (на клиенте и сервере).

Keep It Simple, Stupid

KISS (Keep It Simple, Stupid) – принцип хорошего кода, который гласит:

«Большинство систем работают лучше всего, если они остаются простыми, а не усложняются»

Основные идеи:

- Простые системы будут работать лучше и надежнее.
- Не придумывайте задаче более сложного решения, чем ей требуется.
- Всегда следует искать простой путь, даже если он неочевиден.

- Не стоит перебарщивать с гибкостью и абстракцией в приложении.

Р. Е. Сердюков привел пример:

«Автомат Калашникова — простой механизм, который в общем-то нормально работает»

Правила простого дизайна Кента Бека (Extreme Programming Explained):

- У вас запускаются все тесты.
- Не дублируйте логику. Старайтесь избегать скрытых дубликатов, таких как параллельные иерархии классов.
- Все намерения, важные для программиста, должны быть явно видны.
- Код должен иметь наименьшее возможное количество классов и методов.

You Aren't Gonna Need It

YAGNI (You Aren't Gonna Need It) – принцип хорошего кода, который гласит:

«*Возможности, которые не описаны в требованиях к системе, просто не должны реализовываться*»

Основные идеи:

- Если пишете код, будьте уверены, что он вам понадобится.
- Не пишите код, если думаете, что он пригодится позже.
- Любые бонусные возможности усложняют сопровождение.
- Если написать ненужный код, то:
  - Тратится время, которое можно было использовать на написание нужного кода.
  - Он (ненужный код) может помешать добавлению нужного функционала в будущем.
  - Он (ненужный код) требует отладки, документирования, тестирования и сопровождения.
  - ПО становиться сложнее.

Р. Е. Сердюков привел пример:

«Надо готовить только то, что вы собираетесь есть. Если вы приготовите больше, оно просто пропадет»

Однако принцип — не догма: разработчик должен заглядывать в будущее (на пару версий ПО вперед). В общем и целом следует соблюдать баланс.

Использованные источники

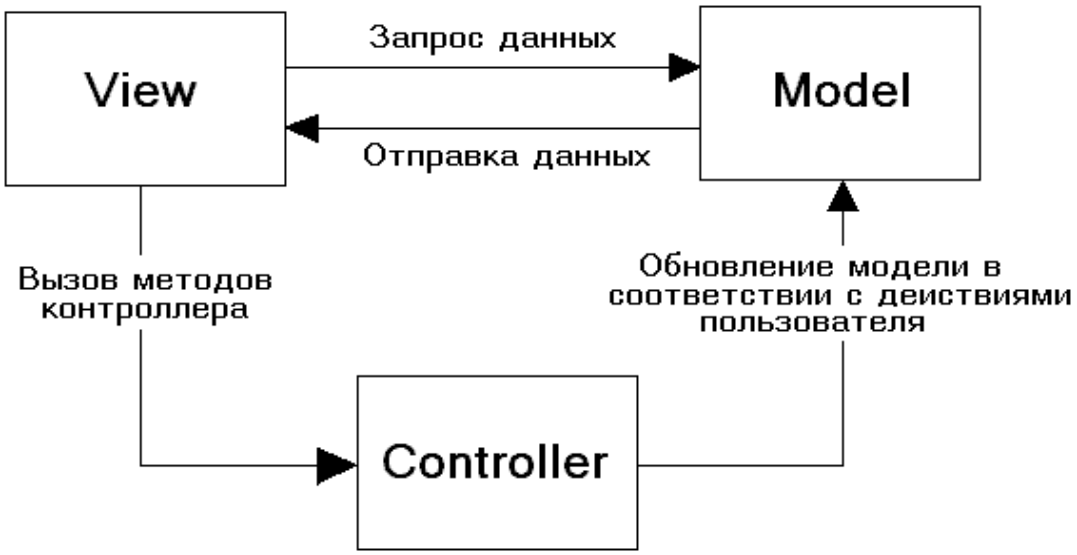
1. Сердюков Р.Е. - YAGNI, DRY, KISS (лекция).
2. Sergey Nemchinskiy — Принцип хорошего кода DRY (видео).
3. Sergey Nemchinskiy — Принцип хорошего кода KISS (видео).
4. Sergey Nemchinskiy — Принцип хорошего кода YAGNI (видео).

23. Операции ввода-вывода. Блокируемый и неблокируемый вызов

24. Паттерн MVC

В графических приложениях для построения интерфейсов пользователя часто применяется тройка классов модель/представление/контроллер (Model/View/Controller — MVC).

MVC



состоит из объектов трех видов. *Модель* — это объект приложения, а *представление* — его внешний вид на экране. *Контроллер* описывает, как интерфейс реагирует на управляющие воздействия пользователя. До появления схемы MVC эти объекты в пользовательских интерфейсах смешивались. MVC отделяет их друг от друга, за счет чего повышается гибкость и улучшаются возможности повторного использования.

MVC отделяет представление от модели, устанавливая между ними протокол взаимодействия «подписка/уведомление». Представление должно гарантировать, что внешнее представление отражает состояние модели. При каждом изменении внутренних данных модель уведомляет все зависящие от нее представления, в результате чего представление обновляет себя. Такой подход позволяет присоединить к одной модели несколько представлений, обеспечив тем самым различные представления. Можно создать новое представление, не переписывая модель.

MVC позволяет также изменять реакцию представления на действия пользователя. При этом визуальное воплощение остается прежним. Например, можно изменить реакцию на нажатие клавиши или использовать открывающиеся меню вместо командных клавиш. MVC инкапсулирует механизм определения реакции в объекте *контроллер*. Существует иерархия классов контроллеров, и это позволяет без труда создать новый контроллер как вариант уже существующего.

Модель

Под *моделью*, обычно понимается часть содержащая в себе функциональную бизнес-логику приложения. *Модель* должна быть полностью независима от остальных частей продукта. Модельный слой ничего не должен знать об элементах дизайна, и каким образом он будет отображаться. Достигается результат, позволяющий менять представление данных, то как они отображаются, не трогая саму *модель*.

Модель обладает следующими признаками:

- Модель — это бизнес-логика приложения.
- Модель обладает знаниями о себе самой и не знает о контроллерах и представлениях.

Представление

В обязанности *представления* входит отображение данных полученных от *модели*. Однако, представление не может напрямую влиять на модель. Можно говорить, что представление обладает доступом «только на чтение» к данным.

Представление обладает следующими признаками:

- В представлении реализуется отображение данных, которые получаются от модели любым способом.
- В некоторых случаях, представление может иметь код, который реализует некоторую бизнес-логику.

Примеры представления: HTML-страница, WPF форма, Windows Form.

Контроллер

Контроллер перехватывает событие извне и в соответствии с заложенной в него логикой, реагирует на это событие изменяя *модель*, посредством вызова соответствующего метода. После изменения *модель* использует событие о том что она изменилась, и все подписанные на это события *представления*, получив его, обращаются к *модели* за обновленными данными, после чего их и отображают.

Признаки контроллера:

- Контроллер определяет, какое представление должно быть отображено в данный момент.

- События представления могут повлиять только на контроллер. Контроллер может повлиять на модель и определить другое представление.
- Возможно несколько представлений только для одного контроллера.

Проблема паттерна MVC

Для построения приложений MVC-шаблон можно использовать по-разному. Наиболее распространенный вариант — разбить систему на множество MVC-модулей, каждый из который отвечает за решение определенной задачи. В таком случае имеем две крайности:

- Можно организовать систему в модули, которые будут слабо связаны между собой.

  Преимущества:

  – Повторное использование модулей в системе.

  – Модули можно отдельно тестировать от всей системы.

  – Относительно легко заменять, добавлять и удалять модули в процессе эксплуатации, не внося в систему больших изменений.

  Недостатки:

  – Пользователи должны чётко представлять себе процесс работы с ней (в какие модули, в какой последовательности заходить, чтобы выполнить задачу и т.п.).

  – Каждый новый пользователь должен проходить специальное обучение для работы с ней.

  – Система не подходит для массового использования.

- Можно связать модули между собой (например, ссылками: чтобы по нажатию на ссылку в представлении модуля A, вызывался контроллер модуля B. По ссылке из представления модуля B вызывался контроллер модуля C. и т.д).

  Преимущества:

  – Пользователям не нужно держать в голове как и в какой последовательности работать с системой.

  – Система подходит для массового использования.

  Недостатки:

- Поддержка системы значительно усложняется
- Любая модификация потребует тестирование всей системы.
- Весь процесс работы с системой будет распределён среди кучи файлов.

И это, как правило, часто происходит при построении приложений. А причина состоит в том, что думая в терминах Model, View, Controller легко выйти за рамки определённого модуля и превратить всю систему в один сплошной монолитный модуль.

Решение проблемы.

Для решения этой проблемы понадобится посредник — Root Controller или Front Controller. И сделать так, чтобы все модули общались через него. А он сам руководствовался бы конфигурацией, какой модуль, при каких условиях показывать пользователю для работы.

Разновидности паттерна MVC

Наиболее распространенные виды MVC-паттерна, это:

- Model-View-Controller.
- Model-View-Presenter.
- Model-View-ViewModel.

Использованные источники

1. Паттерны объектно-ориентированного проектирования (книга, глава 1.2).
2. Статья habr.com – Паттерны для новичков: MVC vs MVP vs MVVM.
3. Статья habr.com – Есть ли будущее у MVC-шаблона?

25. Паттерн MVP и MVVM

Цель MV*-паттернов: отделить друг от друга отображение, логику интерфейса и данные (их получение и обработку). Использование MV*-паттерна позволяет ускорить разработку и разделить ответственность разных специалистов; приложение удобнее тестировать и поддерживать.

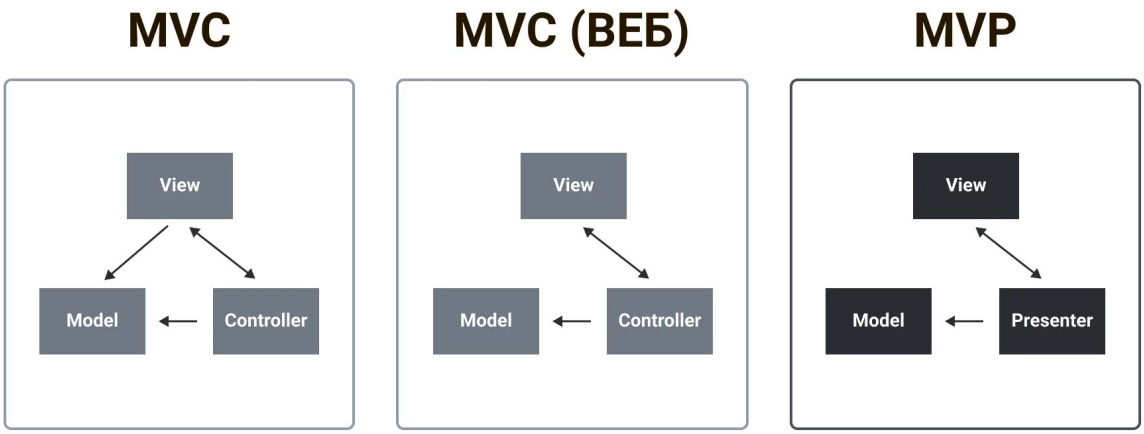
Паттерны MVP и MVVM чаще всего используются разработчиками мобильных приложений.

Паттерн MVP

MVP — это паттерн программирования графических интерфейсов. В нём приложение делится на три компонента:

- Model (Модель) работает с данными, проводит вычисления и руководит всеми бизнес-процессами.
- View (Вид или Представление) показывает пользователю интерфейс и данные из модели.
- Presenter (Представитель) служит прослойкой между моделью и видом.

Кроме мобильной разработки этот паттерн активно применяется в web-разработке, хотя там его и называют MVC:



Преимущества:

- Данная архитектура облегчает unit-тестирование: Представитель прост для написание тестов, а за счет того, что Представитель и Вид общаются через интерфейс, их можно тестировать по отдельности.
- Представитель может многократно использоваться, потому что представление может реализовать несколько интерфейсов.
- Представители могут использоваться в общих модулях.

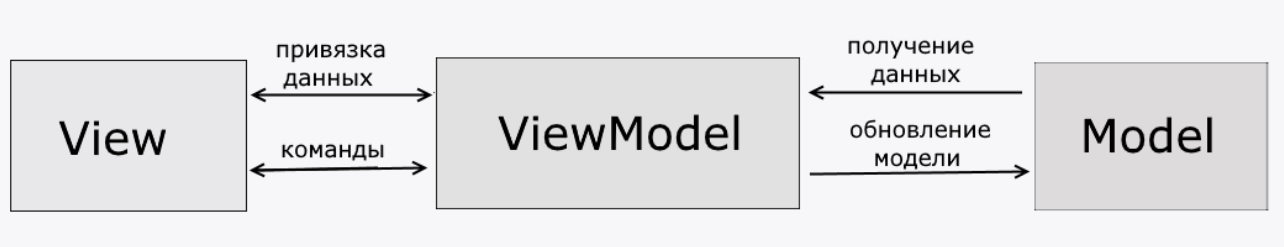
Недостатки:

- Необходимо создавать и поддерживать интерфейсы для представлений.
- Лишний шаблонный код (нарушений принципов SOLID, DRY).

Паттерн MVVM

Паттерн MVVM (Model-View-ViewModel) позволяет отделить логику приложения от визуальной части (представления). В нём приложение делится на три компонента:

- Model (Модель) описывает используемые в приложении данные. Не знает о существовании других компонентов, но уведомляет ViewModel о своих изменениях.
- View (Вид или Представление) определяет визуальный интерфейс, через который пользователь взаимодействует с приложением. Вызывает методы ViewModel, данные получает через Databinding.
- ViewModel (Модель представления) связывает модель и представление через механизм привязки данных. ViewModel является отражением текущего состояния экрана. Вызывает методы модели.



Паттерн MVVM используется в случаях, когда нужно отображать большое количество данных без обработке на View-стороне (например приложение текущего курса валют).

Преимущества:

- Компоненты слабо связаны — их можно тестировать по отдельности, и каждый компонент может быть относительно легко модифицирован.
- Databinding уменьшает количество кода.
- К одной ViewModel можно привязать несколько View.
- Паттерн ViewModel не заставляет вас использовать или избегать code behind для View.

Недостатки:

- Всплывающие сообщения и диалоги сложно обработать в данной архитектуре, т.к. они не являются частью экрана и ViewModel не хранит их состояние.
- Проблема показа анимаций или данных с задержкой, т.к. данные во View всегда актуальны ее ViewModel.
- Необходимость обработки команд во View без передачи в ViewModel.

Использованные источники

1. Статья skillbox.ru - Что такое MVP и как это использовать.
2. Урок ru.coursera.org - Обзор Model/View/ViewModel (Видео).
3. Урок ru.coursera.org - Плюсы и минусы MVVM (Видео).
4. Статья habr.com - Различия между MVVM и остальными MV*-паттернами.
5. Статья habr.com - Паттерны разработки: MVC vs MVP vs MVVM vs MVI.

26. Паттерн BLOC

Область назначения

Для грамотной композиции и составления архитектуры кода проекта с бизнес-логикой мы можем применить BLoC. BLoC — это акроним от «Business Logic Component» (компонент бизнес-логики). BloC используется для создания мобильных приложений на Flutter, а точнее их архитектуры.

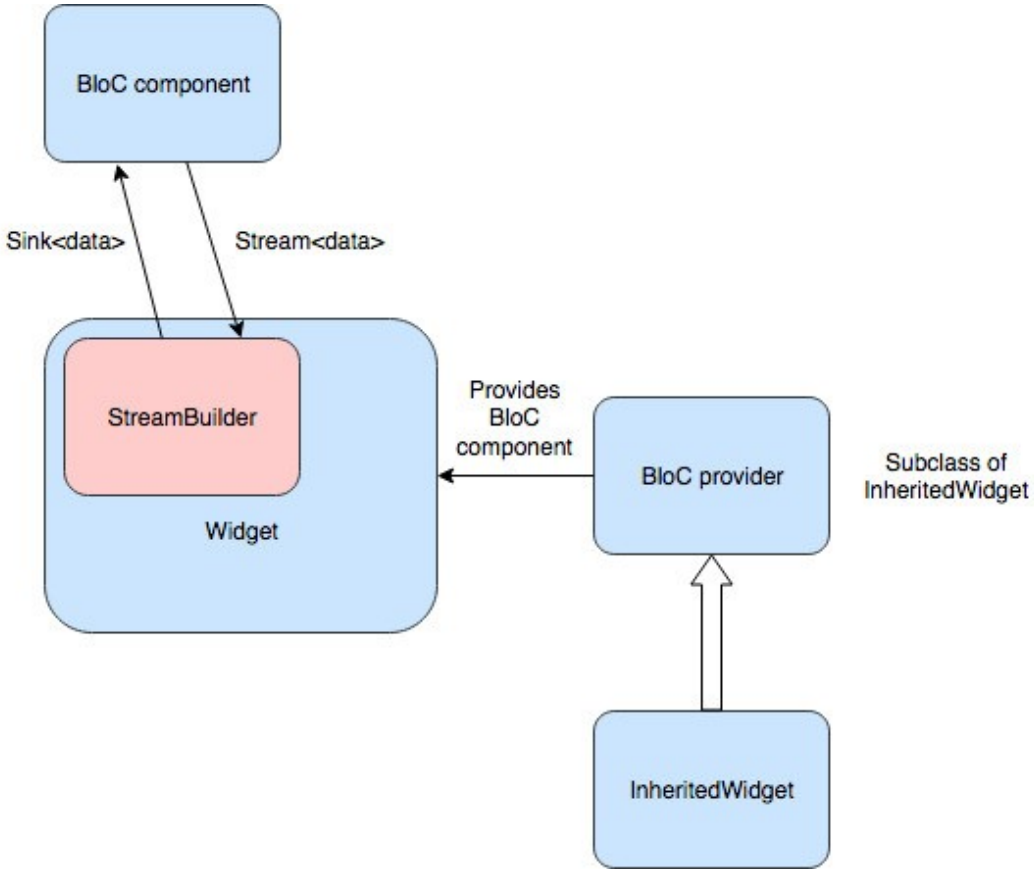
Архитектура

По сути, это просто класс с реактивными потоками внутри себя, которые, в свою очередь, манипулируют поступаемыми извне данными и позволяют UI подписывается на результат вычислений.

BLoC — это простой класс у которого:

- все выходы потоки.
- все входы потоки.

Этот класс должен убирать логику из визуального интерфейса:



На данной диаграмме представлена архитектура паттерна.

Когда пользователь взаимодействует с пользовательским интерфейсом, он отправляет действие или событие на компонент BLoC. Основная задача компонента BLoC интерпретировать событие, словить его и вернуть новое или обновлённое состояние компонента пользовательского интерфейса.

Компоненты BLoC должны выполнять всю работу, связанную с бизнес-логикой. После этого он должен отправить состояние на компонент пользовательского интерфейса, который отвечает за обработку компонента состояния и его надлежащее отображение.

Одной из отличительных особенностей паттерна является то, что он полностью базируется на реактивности (реактивное программирование — это программирование с асинхронными потоками данных).

Пример распределения логики приложения

1. Убрать всю логику из виджетов.
2. В классе BLoC получать только потоки на вход и выдавать только потоки на выход.
3. Создаем отдельный класс, где реализуем BLoC паттерн:
    1. Все свойства и метода класса, скрыты.
    2. Для получения и передачи состояния используем потоки, которые видны снаружи при помощи getters.

Использованные источники

1. Статья habr.com - BLoC паттерн на простом примере.
2. How to Implement the BLoC Architecture in Flutter: Benefits and Best Practices.
3. Разделение бизнес-логики и UI во Flutter с помощью BLoC-архитектуры.
4. Flutter BLoC паттерн.

27. Внедрение зависимостей

Концепция Dependency Injection состоит в том, чтобы перенести ответственность за создание экземпляра объекта из тела метода за пределы класса и передать уже созданный экземпляр объекта обратно.

Зависимость — это объект, который может быть использован (как сервис).

```
class ClassA {
  ClassB classB;
}

class ClassB {
  ClassC classC;
}

class ClassC {
}
```

Класс A зависит от класса B, класс B зависит от класса C. Т.е для конкретной работы классу A и B необходимы B и C соответственно.

Внедрение зависимостей — это не технология, фреймворк, библиотека или что-то подобное. Это просто идея. Идея работать с зависимостями вне зависимого класс.

Внедрение зависимостей ответственно за:

- Создание объектов.
- Представление о том, какие классы требуются этим объектам.
- Предоставление зависимостей этим объектам.

Причины использования принципа

1. Зависимости можно внедрять во время выполнения, а не во время компиляции.
2. Все максимально просто. И зависимый класс, и класс, предоставляющий зависимости, понятны и просты.
3. Классы слабо связаны и легко заменяемы другими классами.
4. Тестируемость. Можно легко заменить зависимости тестовыми версиями во время тестирования.
5. Улучшение структуры кода, т.к. в приложении есть отдельное место для обработки зависимостей. В результате остальные части приложения

могут сосредоточиться на выполнении исключительно своих функций и не пересекаться с зависимостями.

6. Обработка зависимостей без Dependency Injection возможна, но это может привести к сбоям работы приложения.

Способы внедрения зависимостей

Существует три основных типа внедрения зависимостей:

- **constructor injection** — все зависимости передаются через конструктор класса.
- **setter injection** — разработчик добавляет setter-метод, с помощью которого инжектор внедряет зависимость.
- **interface injection** — зависимость предоставляет инжектору метод, с помощью которого инжектор передаст зависимость. Разработчики должны реализовать интерфейс, предоставляющий setter-метод, который принимает зависимости.

Пример использования

Внедрение в конструкторе.

Этот метод требует, чтобы клиент предоставил параметр в конструкторе для зависимости.

```
// Контруктор
Client (Service service) {
    // Сохраняем ссылку на переданный сервис внутри этого клиента
    this.service = service;
}
```

Внедрение через сеттер.

Этот метод требует, чтобы клиент предоставил метод сеттер для зависимости.

```
// Метод сеттер
public void setService(Service service) {
    // Сохраняем ссылку на переданный сервис внутри этого клиента.
    this.service = service;
}
```

Внедрение через интерфейс.

Это просто клиент, публикующий интерфейс для методов установки зависимостей клиента. Его можно использовать для определения того, как инжектор должен общаться с клиентом при внедрении зависимостей.

```java
// Интерфейс сеттера сервиса.
public interface ServiceSetter {
    public void setService(Service service);
}

// Класс клиента
public class Client implements ServiceSetter {
    // Внутренняя ссылка на сервис, используемый этим клиентом.
    private Service service;
    // Установить сервис, который должен использовать этот клиент.
    @Override
    public void setService(Service service) {
        this.service = service;
    }
}
```

Использованные источники

1. Внедрение зависимостей (dependency injection): примеры.
2. Назад к основам: Внедрение зависимости (DI).
3. Краткое введение во внедрение зависимостей: что это и когда это необходимо использовать.
4. Основы внедрения зависимостей.

28. Автоматное программирование

Автоматное программирование — это парадигма программирования, при использовании которой программа или её фрагмент осмысливается как модель какого-либо формального автомата. Известна также и другая "парадигма автоматного программирования, состоящая в представлении сущностей со сложным поведением в виде автоматизированных объектов управления, каждый из которых представляет собой объект управления и автомат". При этом о программе, как в автоматическом управлении, предлагается думать как о системе автоматизированных объектов управления.

Идея автоматного программирования заключается в построении управляющего автомата. Для этой цели программа описывается, как набор состояний, событий и переходов между состояниями.

Область назначения:

1. Разработка микроконтроллеров.
2. Повсеместно.
3. Упрощению по крайней мере разработки пользовательских интерфейсов.
4. Для организации и представления потока выполнения чего-либо.

Структура конечного автомата

Конечный автомат — это некоторая абстрактная модель, содержащая конечное число состояний чего-либо. Используется для представления и управления потоком выполнения каких-либо команд. Конечный автомат идеально подходит для реализации искусственного интеллекта в играх, получая аккуратное решение без написания громоздкого и сложного кода.

Конечный автомат можно представить в виде графа, вершины которого являются состояниями, а ребра — переходы между ними. Каждое ребро имеет метку, информирующую о том, когда должен произойти переход.

Реализация конечного автомата начинается с выявления его состояний и переходов между ними.

Способы задания в приложениях

В теории объектно-ориентированного программирования считается, что объект имеет внутреннее состояние и способен получать сообщения, отвечать на них, отправлять сообщения другим объектам и в процессе обработки сообщений изменять своё внутреннее состояние. Более приближенное к практике понятие

вызова метода объекта считается синонимом понятия отправки сообщения объекту.

Таким образом, с одной стороны, объекты в объектно-ориентированном программировании могут рассматриваться как конечные автоматы (или, если угодно, модели конечных автоматов), состояние которых представляет собой совокупность внутренних полей, в качестве же шага автомата могут рассматриваться один или несколько методов объекта при условии, что эти методы не вызывают ни сами себя, ни друг друга ни прямо, ни косвенно.

С другой стороны, очевидно, что понятие объекта представляет собой удачный инструмент реализации модели конечного автомата. При применении парадигмы автоматного программирования в объектно-ориентированных языках обычно модели автоматов представляются в виде классов, состояние автомата описывается внутренними (закрытыми) полями класса, а код шага автомата оформляется в виде метода класса, причём такой метод скорее всего оказывается единственным открытым методом (не считая конструкторов), изменяющим состояние автомата. Другие открытые методы могут служить для получения информации о состоянии автомата, но не меняют его. Все вспомогательные методы (например, методы-обработчики отдельных состояний или их категорий) в таких случаях обычно убирают в закрытую часть класса.

Пример использования конечного автомата для управления каким-либо объектом:

1. Регулярные выражения являются конечными автоматами.
2. Сейф (событие вызвано)

   Состояния: несколько «заблокированных» состояний, одно «разблокированное» состояние.

   Переходы: правильные комбинации / ключи перемещают вас из начальных заблокированных состояний в заблокированные состояния ближе к разблокированному, пока вы, наконец, не получите разблокированную. Неправильные комбинации / ключи возвращают вас в исходное заблокированное состояние (иногда называемое простоя) .
3. Светофор (срабатывает время | срабатывает датчик [событие])

   Состояния : красный, желтый, зеленый.

   Переходы : после таймера измените КРАСНЫЙ на ЗЕЛЕНЫЙ, ЗЕЛЕНЫЙ на ЖЕЛТЫЙ и ЖЕЛТЫЙ на КРАСНЫЙ. Может также запускаться при обнаружении автомобилей в различных (более сложных) состояниях.

4. Торговый автомат (событие сработало, вариант сейфа)

   Состояния : IDLE, 5_CENTS, 10_CENTS, 15_CENTS, 20_CENTS, 25_CENTS, ..., VEND, CHANGE

   Переходы : состояние изменяется при вставке монет, купюр, переходе к VEND при правильной сумме покупки (или более), затем переходе к CHANGE или IDLE (в зависимости от того, насколько этичен ваш торговый автомат).

5. Одним из примеров может быть конечный автомат, который сканирует строку, чтобы увидеть, имеет ли он правильный синтаксис.

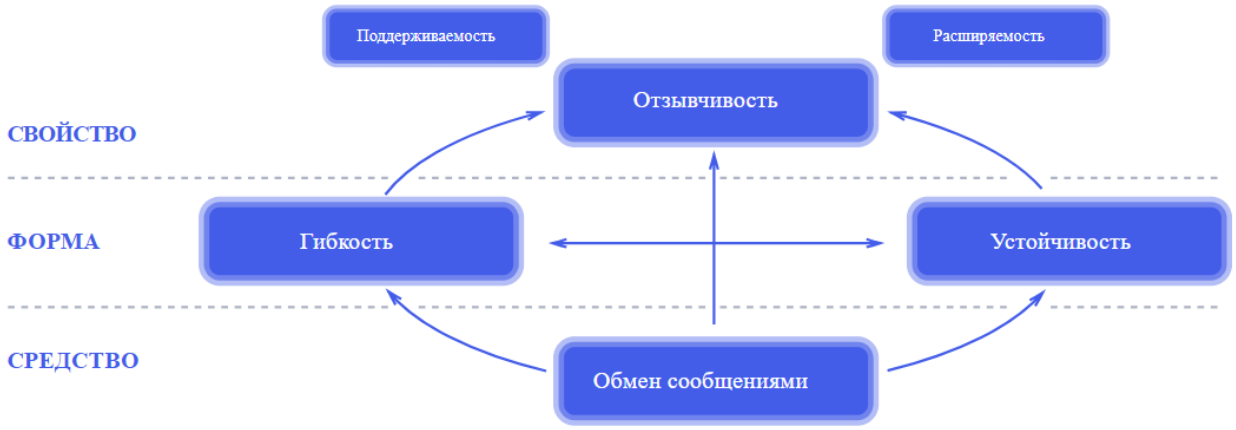
Использованные источники

1. https://qastack.ru/software/47806/examples-of-finite-state-machines (примеры)
2. https://studopedia.ru/view_informatika.php?id=125
3. https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B5%D1%87%D0%BD%D1%8B%D0%B9_%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82
4. https://yamakarov.ru/software/2018/09/25/switch-programming.html

29. Реактивное программирование

Реактивное программирование — это программирование с асинхронными потоками (streams) данных.

Манифест реактивных систем



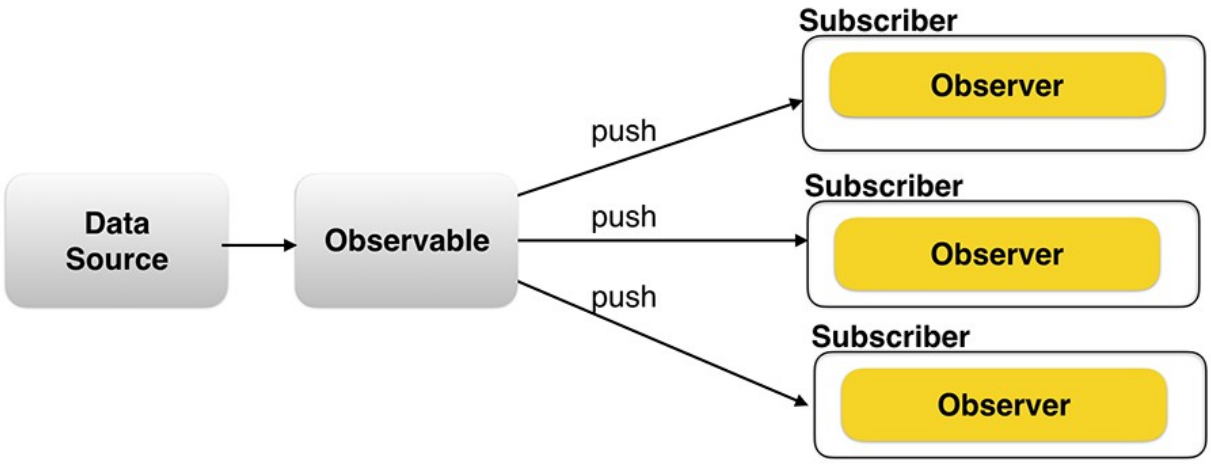
Реактивные системы:

1. Отзывчивые: система отвечает своевременно, если это вообще возможно. Отзывчивость является краеугольным камнем удобного и полезного приложения, но, помимо этого, она позволяет быстро обнаруживать проблемы и эффективно их устранять. Отзывчивые системы ориентированы на обеспечение быстрого и согласованного времени отклика, устанавливая надежные верхние границы, чтобы обеспечить стабильное качество обслуживания. Такое предсказуемое поведение, в свою очередь, упрощает обработку ошибок, повышает уверенность конечного пользователя в работоспособности и способствует дальнейшему взаимодействию с системой.
2. Устойчивые: система остается доступной даже в случае отказов. Это относится не только к высокодоступным, критически важным приложениям — без устойчивости любая система при сбое теряет отзывчивость. Устойчивость достигается за счет репликации, сдерживания, изоляции и делегирования. Эффект от отказов удерживается внутри компонентов, изолируя их друг от друга, что позволяет им выходить из строя и восстанавливаться, не нарушая работу системы в целом. Восстановление каждого компонента делегируется другому (внешнему) модулю, а высокая доступность обеспечивается за счет репликации там, где это необходимо. Клиент компонента не отвечает за обработку его сбоев.

3. Гибкие: система остается отзывчивой под разными нагрузками. Реактивные системы способны реагировать на колебания в скорости входящих потоков, увеличивая или уменьшая количество выделенных на их обслуживание ресурсов. Для этого архитектура не должна допускать наличия централизованных узких мест или конкуренции за ресурсы, что позволяет сегментировать или реплицировать компоненты, распределяя между ними входные данные. Реактивные Системы поддерживают предсказывающие и Реактивные алгоритмы масштабирования, позволяя делать измерения производительности в режиме реального времени. Гибкость достигается применением экономически эффективных аппаратных и программных платформ.
4. Основаны на обмене сообщениями: реактивные системы используют асинхронный обмен сообщениями, чтобы установить границы между компонентами и обеспечить слабую связанность, изоляцию и прозрачность размещения. Эти границы также позволяют преобразовывать и передавать информацию о сбое в виде сообщений. Открытый обмен сообщениями делает возможными регулирование нагрузки, гибкость и управление потоком, для чего в системе создаются и отслеживаются очереди сообщений и в случае необходимости используется обратное давление. Прозрачность размещения при взаимодействии на основе сообщений позволяет применять к механизму обработки ошибок одни и те же ограничения и семантику как в пределах одного компьютера, так и в масштабах целого кластера. Благодаря неблокирующему взаимодействию принимающая сторона потребляет ресурсы только при активной работе, что позволяет снизить накладные расходы.

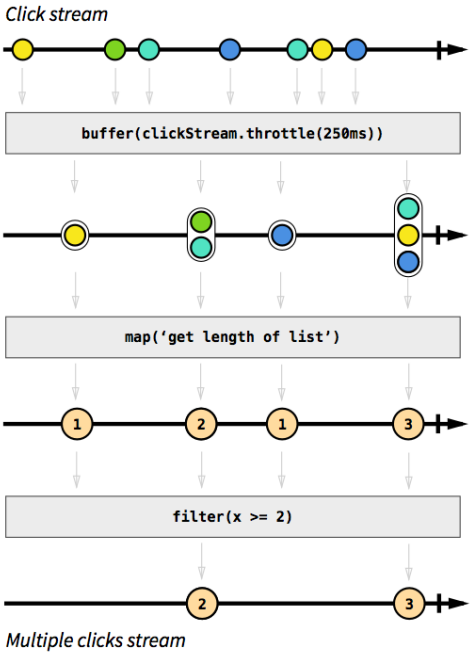
Использование в Java

Event bus’ы или обычные события клика — это тоже асинхронные потоки данных, которые вы можете прослушивать, чтобы реагировать какими-либо действиями. Реактивность — это та же идея, возведенная в абсолют. Вы можете создавать потоки данных не только из событий наведения или событий мыши. Потоком может быть что угодно: переменные, пользовательский ввод, свойства, кэш, структуры данных и т.п. Например, представьте, что ваша лента новостей в Твиттере — поток событий. Вы можете слушать этот поток и реагировать на события соответственно.



Кроме этого, вы получаете удивительный набор функций для комбинирования, создания и фильтрации этих потоков. Один или несколько потоков могут использоваться как входные данные для другого потока. Вы можете объединять два потока. Также вы можете фильтровать поток, выбирая только те события, которые вам интересны.

Пример использования обработки потока сообщений:



Серые прямоугольники — это функции, которые трансформируют один поток в другой. Сначала мы собираем клики в списки. Если прошло 250 миллисекунд без единого нажатия кнопки — мы применяем функцию *map()* на каждом из списков, чтобы вычислить его длину. В конце мы фильтруем списки с длиной 1, используя функцию *filter(x >= 2)*. Вот так, в три действия, мы получаем

результат — поток событий множественных кликов. Мы можем подписаться на него и использовать, как пожелаем.

Этот пример показывает всю простоту, с которой реализовывается достаточно сложная на первый взгляд задача, если мы используем реактивный подход.

Библиотеки реактивного программирования в различных языках:

Язык	Библиотека
Java	RxJava
JavaScript	RxJS
.Net	Rx.Net

Использованные источники

1. https://www.reactivemanifesto.org/ru (его источник)
2. https://tproger.ru/translations/reactive-programming/
3. https://docs.google.com/presentation/d/1a70JieCly8XJiveGOclRRZnPN09CbkS5QqXlaq9azvM/edit?usp=sharing (ссылка на его презентацию)

30. Аспектно-ориентированное программирование

АОП — аспектно-ориентированное программирование — это парадигма, направленная на повышение модульности различных частей приложения за счет разделения сквозных задач. Для этого к уже существующему коду добавляется дополнительного поведение, без изменений в изначальном коде.

Иными словами, мы как бы навешиваем сверху на методы и классы дополнительную функциональность, не внося поправки в модифицируемый код.

Применение

- Предназначено для решения сквозных задач, которые могут представлять собой любой код, многократно повторяющийся разными методами, который нельзя полностью структурировать в отдельный модуль.

  В качестве примера можно привести применение политики безопасности в каком-либо приложении.

  Как правило безопасность проходит сквозь многие элементы приложения. Тем более, политика безопасности приложения должна применяться одинаково ко всем существующим и новым частям приложения. При этом используемая политика безопасности может и сама развиваться.

  Также в качестве еще одного примера можно привести логирование.
- Используется для обработки исключений.
- Используется для кеширования.
- Используется для выноса некоторого функционала, чтобы сделать его переиспользуемым.
- Обработка транзакций.

Основная задача АОП — модуляризация сквозной функциональности, выделение её в аспекты.

Для этого языки, поддерживающие концепцию АОП, реализуют следующие средства для выделения сквозной функциональности:

- **аспект (aspect)** – модуль или класс, реализующий сквозную функциональность. Аспект изменяет поведение остального кода, применяя совет в точках соединения, определённых некоторым срезом. Так же аспект может использоваться для внедрения функциональности.

- **совет (advice)** – дополнительная логика — код, который должен быть вызван из точки соединения. Совет может быть выполнен до, после или вместо точки соединения.
- **точка соединения (join point)** — точка в выполняемой программе (вызов метода, создание объекта, обращение к переменной), где следует применить совет.
- **срез (pointcut)** — набор точек соединения. Срез определяет, подходит ли данная точка соединения к заданному совету.
- **внедрение (introduction)** — изменение структуры класса и/или изменение иерархии наследования для добавления функциональности аспекта в инородный код.
- **цель (target)** – объект, к которому будут применяться советы.
- **переплетение (weaving)** – связывание объектов с соответствующими аспектами (возможно на этапе компиляции, загрузки или выполнения программы).

Примеры

1.
```
public class Main {

  public static void main(String[] args) {
  printName("Толя");
  printName("Вова");
  printName("Саша");
  }

  public static void printName(String name) {
     System.out.println(name);
  }
}
```
Если мы сейчас запустим, в консоли будет выведено:

Толя
Вова
Саша
Нужно создать файл — аспект. Они бывают двух видов: первый — файл с расширением .aj, второй — обычный класс, который реализует возможности АОП при помощи аннотаций.

Сперва рассмотрим файл с расширением .aj:

```
public aspect GreetingAspect {

  pointcut greeting() : execution(* Main.printName(..));

  before() : greeting() {
     System.out.print("Привет ");
  }
}
```

pointcut — срез или набор точек соединения.

greeting() — название данного среза.

: execution — при выполнении * (всех, вызов) метода ***Main.printName(..)***.

Далее идёт конкретный совет — ***before()*** — который выполняется до вызова целевого метода, : ***greeting()*** — срез, на который данный совет реагирует, ну а ниже мы видим само тело метода.

При запуске main с наличием данного аспекта мы получим вывод в консоль:

Привет Толя
Привет Вова
Привет Саша

Мы видим, что каждый вызов метода printName был модифицирован при помощи аспекта.

Аспект, но уже как класс Java с аннотациями:

```
@Aspect
public class GreetingAspect{

  @Pointcut("execution(* Main.printName(String))")
  public void greeting() {
  }

  @Before("greeting()")
  public void beforeAdvice() {
     System.out.print("Привет ");
  }
}
```

@Aspect – обозначает, что данный класс является аспектом.

@Pointcut("execution(* Main.printName(String))") — точка среза, которая срабатывает на все вызовы ***Main.printName*** с входящим аргументом типа String.

@Before("greeting()") — совет, который применяется до вызова кода описанного в точке среза ***greeting()***.

При запуске main с этим аспектом вывод в консоли не изменится:

Привет Толя
Привет Вова
Привет Саша

2.
```
public class Main {

   public static void main(String[] args) {
   makeSomeOperation("Толя");
   }

   public static void makeSomeOperation(String clientName) {
      System.out.println("Выполнение некоторых операций для клиента - " + clientName);
   }
}
```

С помощью аннотации ***@Around*** сделаем что-то типа “псевдотранзакции”:

```
@Aspect
public class TransactionAspect{

  @Pointcut("execution(* Main.makeSomeOperation(String))")
  public void executeOperation() {
  }

  @Around(value = "executeOperation()")
  public void beforeAdvice(ProceedingJoinPoint joinPoint) {
     System.out.println("Открытие транзакции...");
     try {
        joinPoint.proceed();
        System.out.println("Закрытие транзакции....");
     }
     catch (Throwable throwable) {
        System.out.println("Операция не удалась, откат транзакции...");
     }
  }
  }
```

С помощью метода ***proceed()*** объекта ***ProceedingJoinPoint*** мы вызываем оборачиваемый метод, чтобы определить его место в совете и, соответственно, код в методе, который выше ***joinPoint.proceed()*** — это Before, который ниже — After.

Если мы запустим main, в консоли мы получим:

Открытие транзакции...
Выполнение некоторых операций для клиента - Толя
Закрытие транзакции....

Если же мы добавим бросок исключения в наш метод (вдруг выполнение операции дало сбой):

```
public static void makeSomeOperation(String clientName)throws Exception {
  System.out.println("Выполнение некоторых операций для клиента - " + clientName);
  throw new Exception();
}
```

То мы получим вывод в консоли:

Открытие транзакции...
Выполнение некоторых операций для клиента - Толя
Операция не удалась, откат транзакции…

Получилась такая себе псевдообработка неудачи.

3. Сделаем что-то типа логирования в консоли. Для начала посмотрим в Main, где у нас происходит псевдо-бизнес-логика:

```
public class Main {
  private String value;

  public static void main(String[] args) throws Exception {
     Main main = new Main();
     main.setValue("<некоторое значение>");
     String valueForCheck = main.getValue();
     main.checkValue(valueForCheck);
  }

  public void setValue(String value) {
     this.value = value;
  }

  public String getValue() {
     return this.value;
  }

  public void checkValue(String value) throws Exception {
     if (value.length() > 10) {
        throw new Exception();
     }
  }
}
```

В main с помощью ***setValue*** мы зададим значение внутренней переменной — ***value***, далее с помощью ***getValue*** возьмём это значение и в ***checkValue*** проверим, длиннее ли это значение 10 символов. Если да, будет брошено исключение.

Теперь посмотрим на аспект, с помощью которого мы будем логировать работу методов:

```
@Aspect
public class LogAspect {

  @Pointcut("execution(* *(..))")
  public void methodExecuting() {
  }

  @AfterReturning(value = "methodExecuting()", returning = "returningValue")
  public void recordSuccessfulExecution(JoinPoint joinPoint, Object returningValue) {
     if (returningValue != null) {
        System.out.printf("Успешно выполнен метод - %s, класса- %s, с результатом выполнения - %s\n",
              joinPoint.getSignature().getName(),
              joinPoint.getSourceLocation().getWithinType().getName(),
              returningValue);
     }
     else {
        System.out.printf("Успешно выполнен метод - %s, класса- %s\n",
              joinPoint.getSignature().getName(),
              joinPoint.getSourceLocation().getWithinType().getName());
     }
  }


  @AfterThrowing(value = "methodExecuting()", throwing = "exception")
  public void recordFailedExecution(JoinPoint joinPoint, Exception exception) {
     System.out.printf("Метод - %s, класса- %s, был аварийно завершен с исключением - %s\n",
           joinPoint.getSignature().getName(),
           joinPoint.getSourceLocation().getWithinType().getName(),
           exception);
  }
}
```

@Pointcut("execution(* *(..))") — будет соединяться со всеми вызовами всех методов;

@AfterReturning(value = "methodExecuting()", returning = "returningValue") — совет, который будет выполнен после успешного выполнения целевого метода. У нас тут есть два случая:

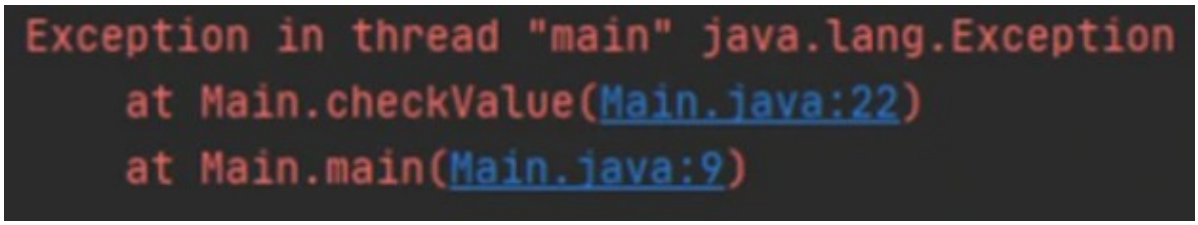
- Когда у метода есть возвращаемое значение ***if (returningValue != null)***.
- Когда возвращаемого значения нет ***else***.

@AfterThrowing(value = "methodExecuting()", throwing = "exception") — совет, который будет срабатывать при ошибке, то есть при падении исключения из метода.

И соответственно, запустив main, мы получим своеобразное логирование в консоли:

Успешно выполнен метод - setValue, класса- Main
Успешно выполнен метод - getValue, класса- Main, с результатом выполнения - <некоторое значение>
Метод - checkValue, класса- Main, был аварийно завершен с исключением - java.lang.Exception
Метод - main, класса- Main, был аварийно завершен с исключением — java.lang.Exception

Ну и так как мы не обработали исключения, еще получим его стектрейс:



Использованные источники

https://javarush.ru/groups/posts/3137-chto-takoe-aop-osnovih-aspektno-orientirovannogo-programmirovanija

https://habr.com/ru/post/114649/