

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.....	4
2. Java. Классы и интерфейсы.....	5
Класс.....	5
Абстрактный класс.....	5
Интерфейс.....	6
Основные отличия и сходства.....	7
Контроль доступа.....	7
Использованные источники.....	8
3. Java. Управление памятью. Сборщик мусора.....	9
Управление памятью.....	9
Сборщик мусора.....	11
Использованные источники.....	12
4. Java. Внутренние и анонимные классы. Lambda-выражения.....	13
5. Java. Generic.....	14
Шаблонные классы.....	14
Шаблонные методы.....	15
Шаблонные интерфейсы.....	15
Шаблонные конструкторы.....	16
Использование нескольких универсальных параметров.....	17
Использованные источники.....	17
6. Java. Типы потоков ввода-вывода, принцип использования.....	18
Классификация потоков.....	18
Принцип использования.....	19
Использованные источники.....	20
7. Java. Сериализация объектов в raw.....	21
8. Java. Сериализация объектов в XML.....	22
Алгоритм сериализации в Java.....	23
Форматы сериализации.....	23
Сериализация в XML.....	24
Альтернатива сериализации.....	25
Использованные источники.....	25
9. Java. Чтение и запись в сетевой ресурс.....	26
10. Java. Основные возможности библиотеки javax.swing.....	27
11. Java. Обработка событий в библиотеке javax.swing.....	28
12. Java. Многопоточность в библиотеке javax.swing.....	29
13. Java. Enterprise Edition. Библиотека Spring.....	30
14. Java. Stream API.....	31
Классификация Stream.....	31
Операторы класса Stream.....	32
Методы Collectors.....	34
Использованные источники.....	34
15. Java. Reflection API.....	35
16. Java. Аннотации.....	36
17. Java. Обработка исключений.....	37
Ключевые слова.....	37
Иерархия исключений Java.....	38
Блок try-with-resources.....	39
Пример генерации и обработки исключений.....	40
Использованные источники.....	40

18. Java. Threads. Создание и управление.....	41
19. Java. Threads. Синхронизация.....	42
Монитор.....	42
Способы синхронизации кода.....	42
Синхронизация статических методов.....	43
Методы и состояния блокировки.....	43
Приоритеты потоков.....	44
Использованные источники.....	44
20. Java. Асинхронность.....	45
21. Принципы GRASP.....	46
Основные шаблоны.....	46
Дополнительные шаблоны.....	48
Использованные источники.....	48
22. Принципы YAGNI, DRY и KISS.....	49
Don't Repeat Yourself.....	49
Keep It Simple, Stupid.....	49
You Aren't Gonna Need It.....	50
Использованные источники.....	51
23. Операции ввода-вывода. Блокируемый и неблокируемый вызов.....	52
24. Паттерн MVC.....	53
Модель.....	54
Представление.....	54
Контроллер.....	54
Проблема паттерна MVC.....	55
Разновидности паттерна MVC.....	56
Использованные источники.....	56
25. Паттерн MVP и MVVM.....	57
Паттерн MVP.....	57
Паттерн MVVM.....	58
Использованные источники.....	59
26. Паттерн BLOC.....	60
27. Внедрение зависимостей.....	61
28. Автоматное программирование.....	62
29. Реактивное программирование.....	63
30. Аспектно-ориентированное программирование.....	64

Илья и Антон

1. **Java. Структура программы**

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 (<i>extends</i>)		Только 1 другой класс		Любое количество других интерфейсов
Реализует (<i>implements</i>)		Любое количество интерфейсов		-
Поля		Любые модификаторы доступа, переменные и константы		Только <i>public static final</i>
Методы	Неабстрактные	Любые модификаторы доступа		<i>private, private static, public default</i>
	Абстрактные	-	Любые модификаторы доступа, кроме <i>private</i>	Только <i>public</i>

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

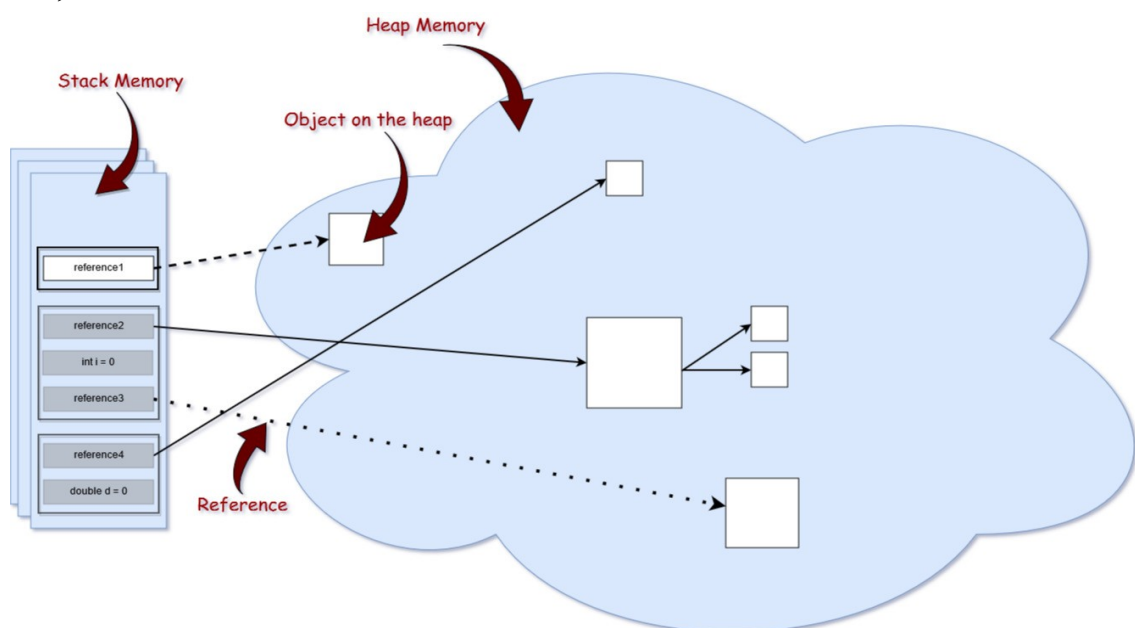
В Java память делится на две большие части:

- **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) - однопоточный 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; }  
}
```

Использованные источники

- 1 [Статья metanit.com - Обобщения \(Generics\).](https://metanit.com/java/learn/generics.html)
- 2 [Обобщение типа данных, generic.](https://habr.com/ru/post/144111/)
- 3 [Статья javarush.ru - Теория дженериков в Java или как на практике ставить скобки.](https://javarush.ru/guides/java/generics/)

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 то же. 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		

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.

```

1  import java.io.FileReader;
2  import java.io.FileWriter;
3  import java.io.IOException;
4
5  public class WriterReader {
6      public static void main(String[] args) {
7          writeText();
8          readText();
9      }
10
11     private static void writeText() {
12         // Эту строку мы посимвольно запишем в файл
13         String test = "TEST!!!";
14
15         /*
16          * Создание файлового потока для записи символов как автозакрываемый
17          * ресурс. Нам не надо вызывать fw.close(), т.к. в данном случае он
18          * будет закрыт автоматически.
19          */
20         try (FileWriter fw = new FileWriter("text.txt")) {
21             // Записываем посимвольно, обращаясь к каждому элементу строки
22             for (int i = 0; i < test.length(); i++) {
23                 fw.write(test.charAt(i));
24             }
25         } catch (IOException ex) {
26             ex.printStackTrace(System.out);
27         }
28     }
29
30     private static void readText() {
31         try (FileReader fr = new FileReader("text.txt")) {
32             // Переменная для хранения строки
33             StringBuilder sb = new StringBuilder();
34             int code = -1;
35             // Читаем посимвольно пока код символа не станет равным -1
36             while ((code = fr.read()) != -1) {
37                 // Вызов Character.toChars() преобразует int в char
38                 sb.append(Character.toChars(code));
39             }
40             System.out.println(sb.toString());
41         } catch (IOException ex) {
42             ex.printStackTrace(System.out);
43         }
44     }
45 }
46

```

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.

Использованные источники

1. [Статья java-course.ru](http://java-course.ru) - Потоки ввода-вывода.

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 десериализации 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:

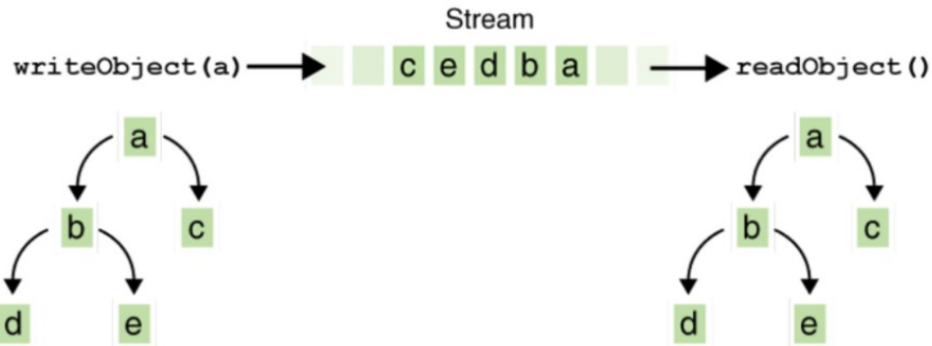
- Не сериализуются поля помеченные `static` и `transient`.
- Поля помеченные `transient` при десериализации инициализируются дефолтными значениями.
- При десериализации выделяется память под объект и поля инициализируются значениями из потока, конструктор этого объекта не вызывается.

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:

```
1 try (XMLEncoder xmlEncoder = new XMLEncoder (new BufferedOutputStream(new
2     FileOutputStream("file.xml")))) {
3     Student student = new Student("Biceps", 921704);
4     xmlEncoder.writeObject(student);
5     xmlEncoder.flush();
6 } catch (FileNotFoundException e) {
7     e.printStackTrace();
8 }
```


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 документа.
- 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 прочитать 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](http://javarush.ru) - Форматы сериализации в Java.
2. [Статья habr.com](http://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

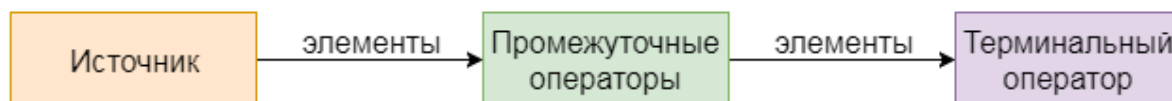
«Позволяет представить обработку нашей информации, некоторого потока, в виде последовательности применения функций» - Сердюков.

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.

Общая схема:



Классификация Stream

Получение объекта Stream (источник):

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

Стримы 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 стримов `Stream<T>`, существуют специальные стримы 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:

```
1  IntStream.of(120, 410, 85, 32, 314, 12)
2      .filter(x->x<300)
3      .map(x->x+11)
4      .limit(3)
5      .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”).

- ***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`).
- ***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.

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.
- ***long count()*** — возвращает количество элементов стрима.
- ***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.
- ***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.
- ***Optional min(Comparator comparator) / Optional max(Comparator comparator)*** — поиск минимального/максимального элемента, основываясь на переданном компараторе.
- ***Optional findFirst()*** — гарантированно возвращает первый элемент стрима, даже если стрим параллельный.
- ***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 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 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.

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 \(annimon.com\)](http://annimon.com) - Полное руководство по Java 8 Stream API.
2. [Статья \(metanit.com\)](http://metanit.com) - Введение в Stream API.
3. [Сердюков Р.Е. - Stream API \(лекция\).](#)

15. **Java. Reflection API**

Оля

16. **Java. Аннотации**

Оля

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

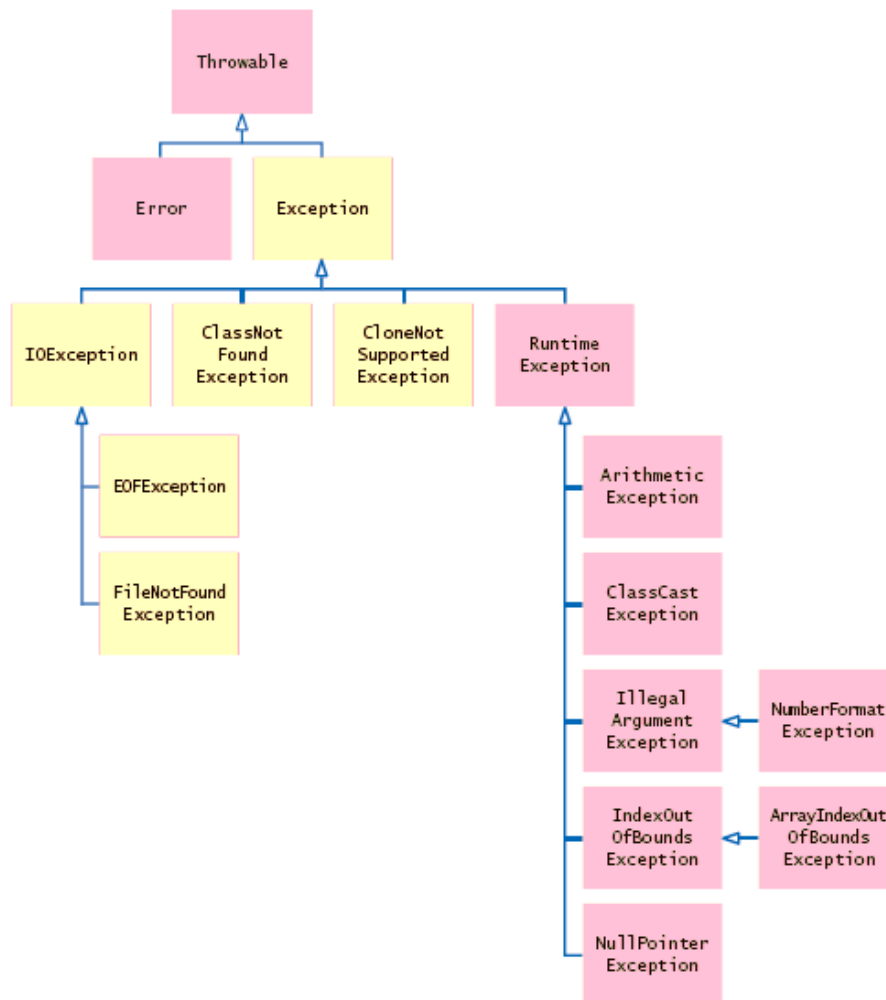
Обработка исключений в Java основана на использовании в программе следующих ключевых слов:

- *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.
- `getStackTrace()` – 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:

```
1 try (InputStream is = new FileInputStream("a.txt")) {  
2     readFromInputStream(is);  
3 }
```

Пример генерации и обработки исключений

```
1 public void testMethod(int a) throws Exception {  
2     if (a < 0) throw new Exception();  
3 }  
4  
5 public void method(int a) {  
6     try {  
7         testMethod(a);  
8     } catch (Exception e) {  
9         e.printStackTrace();  
10    }  
11 }
```

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

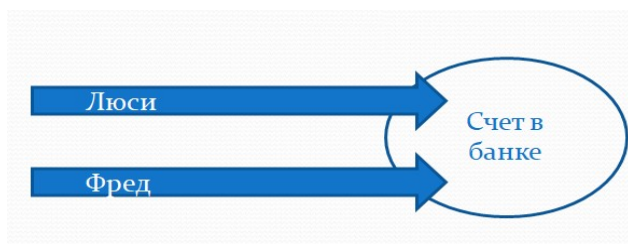
Каждый объект в Java имеет ассоциированный с ним монитор. Монитор — это объект, используемый в качестве взаимоисключающей блокировки. Только один поток исполнения может в одно, и то же время владеть монитором. Все другие потоки исполнения, пытающиеся войти в заблокированный монитор, будут приостановлены 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
<i>wait()</i>	<i>notify()</i>	<i>java.lang.Object</i>
	<i>join()</i>	<i>java.lang.Thread</i>
	<i>sleep()</i>	<i>java.lang.Thread</i>
	<i>yield()</i>	<i>java.lang.Thread</i>

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.

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.](#)

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 В обязанность создавать объекты другого класса А.

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

- **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 связаны, 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 Yourself) – 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 (некоторые 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 задачу 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 становится 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.

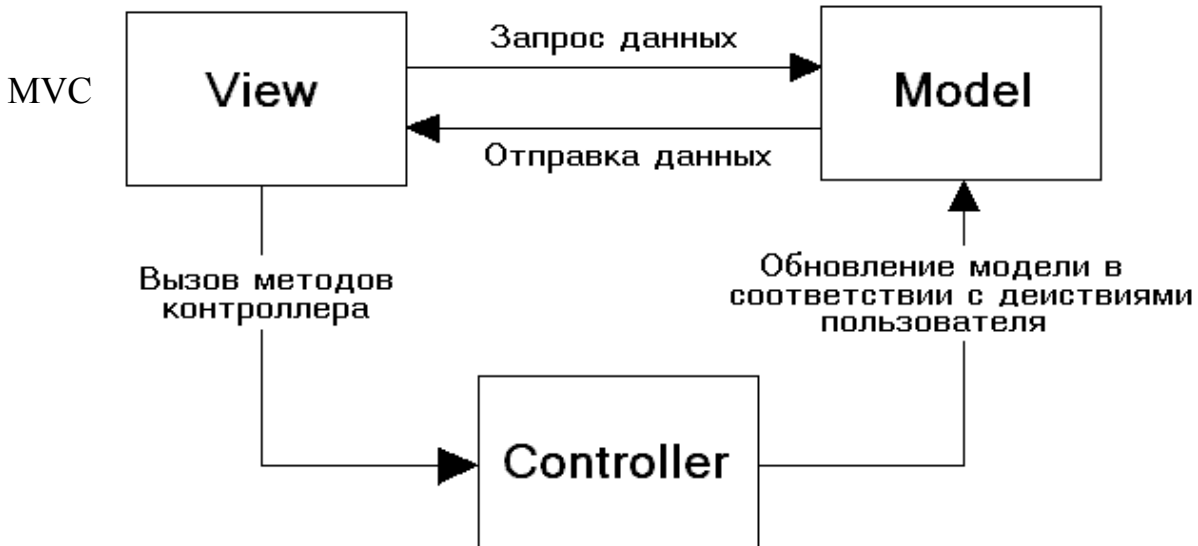
Использованные источники

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).



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 которых 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 А, вызывался контроллер модуля В. По ссылке из представления модуля В вызывался контроллер модуля С. 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.

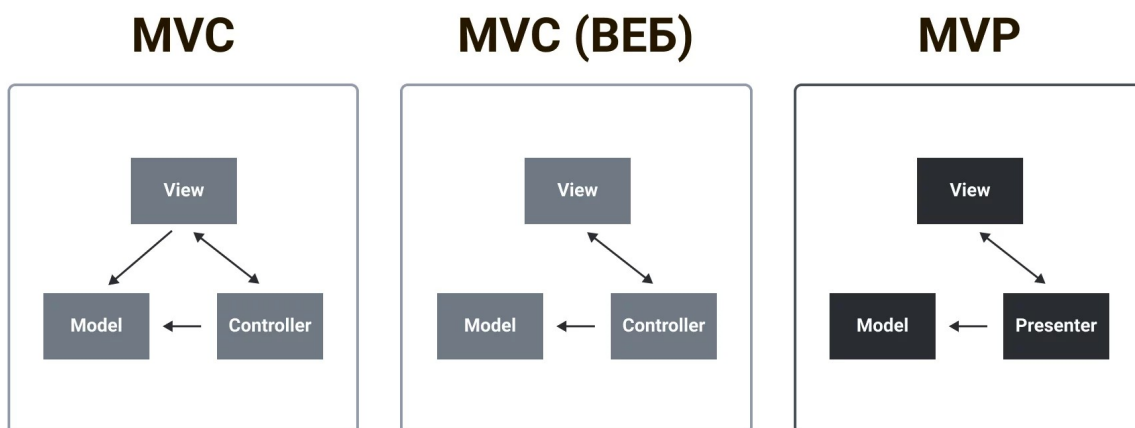
Паттерны MVP и MVVM чаще всего используются разработчиками мобильных приложений.

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 написания 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.

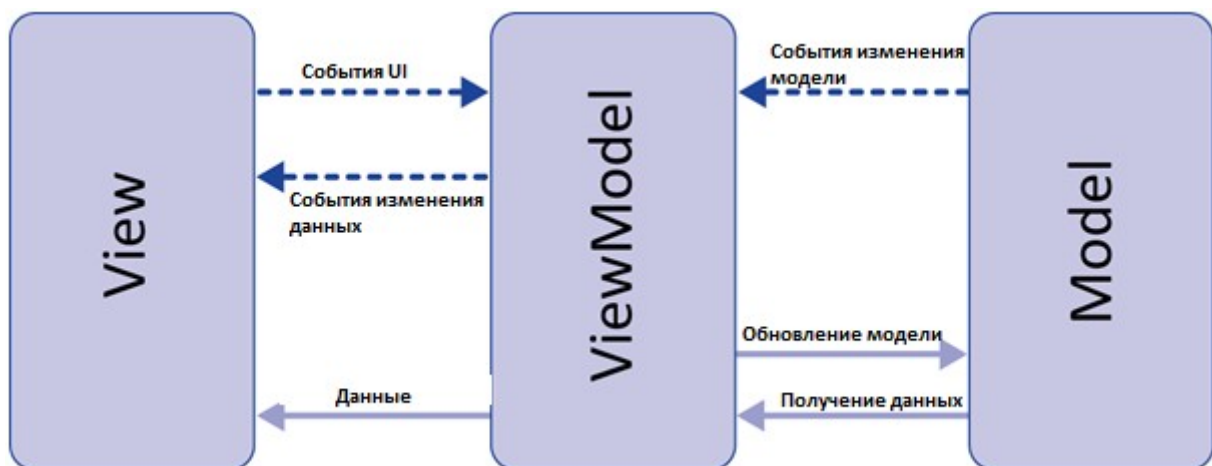
Недостатки:

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

26. Паттерн BLOC

Даша

27. Внедрение зависимостей

Даша

28. Автоматное программирование

Даша

29. Реактивное программирование

Даша

30. **Аспектно-ориентированное программирование**

Даша