

# ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ: КЛЮЧЕВЫЕ ПРИНЦИПЫ




ЛЮБОВЬ МАЯСОВА



**ЛЮБОВЬ МАЯСОВА**

Райффайзенбанк

 [@vlladanna](https://t.me/vlladanna)

 [@LyubovMayasova](https://vk.com/LyubovMayasova)



# План занятия

1. [Ключевые принципы](#)
2. [Абстракция](#)
3. [Инкапсуляция](#)
4. [Композиция](#)
5. [Наследование и полиморфизм](#)
6. [Итоги](#)



# КЛЮЧЕВЫЕ ПРИНЦИПЫ



# КЛЮЧЕВЫЕ ПРИНЦИПЫ

Ключевые принципы — это основные идеи, на которых строится определённый подход или деятельность.

Например, в тестировании\* у нас есть 7 ключевых принципов:

1. Тестирование показывает лишь наличие дефектов
2. Исчерпывающее тестирование невозможно
3. Раннее тестирование экономит время и деньги
4. Дефекты имеют особенность группироваться (эффект соседей)
5. Существует парадокс пестицида (устойчивость дефектов к тестам)
6. Тестирование должно зависеть от контекста
7. Отсутствие дефектов - заблуждение

Примечание\*: согласно ISTQB.



## КЛЮЧЕВЫЕ ПРИНЦИПЫ

Благодаря знанию этих принципов (а ещё желательно и причин, по которым каждый принцип существует) мы можем организовывать **свою работу**, а также понимать, как и почему **другие** организуют свою работу.



# КЛЮЧЕВЫЕ ПРИНЦИПЫ

В ООП тоже существуют свои принципы

1. абстракция\*
2. инкапсуляция
3. наследование и полиморфизм
4. композиция\*

Примечание\*: в разных источниках вы можете встретить разный список принципов (в большинстве будет только п.2-3), но мы будем рассматривать наиболее полную версию.



## КЛЮЧЕВЫЕ ПРИНЦИПЫ

Для отражения текущих тенденций мы добавим ещё один важный для нас принцип ООП — **Testability**. Он обозначает пригодность к тестированию. Testability позволяет понять, насколько удобно тестировать разработанные нами классы и методы.

**Важно:** мы добавили этот принцип «от себя» и на собеседовании вам не нужно говорить, что это ключевой принцип ООП. Это принцип современной разработки ПО в целом: если наша система (или её части) непригодны к тестированию, то нас ждут большие проблемы.





# КЛЮЧЕВЫЕ ПРИНЦИПЫ

Сегодня мы верхнеуровнево разберём ключевые принципы ООП.

Это поможет нам сформировать общую картину и понять, зачем это нужно.



# АБСТРАКЦИЯ



# АБСТРАКЦИЯ

**Абстракция** — ключевой принцип, позволяющий нам справляться со сложностью внешнего мира.

С помощью абстракции мы выделяем наиболее важные характеристики и информацию об объекте. Таким образом, мы отсекаем «всё ненужное», оставляя только **самое необходимое для решения задачи**.



# АБСТРАКЦИЯ

Легко сказать «отсекаем всё ненужное», но как это сделать?

Когда вы читаете «серьёзные» книжки, в них часто моделируются объекты реального мира. Иногда нам сложно понять, на основании чего выбираются те или иные характеристики таких объектов?

На самом деле всё немного проще (не будем усложнять), давайте смотреть на примере.

# ЯНДЕКС ГЛАВНАЯ СТРАНИЦА (БЛОК АФИША)

## Афиша



Вперёд  
мультфильм



Отель «Белград»  
комедия



Остров фантазий  
ужасы



Джентльмены  
боевик

Когда мы смотрим на эту картинку, то видим список объектов — фильмов. Они однотипные (состоят из одинаковых частей), поэтому можно написать класс, который бы описывал объекты этого типа.

*Вопрос к аудитории: какие поля должны быть в этом объекте?*



# АНАЛИЗ И ДИЗАЙН

Как мы это сделали? Посмотрели на афиши и заметили, что все они состоят из одинаковых частей:

1. Изображение
2. Название
3. Жанр

То есть, мы описали в виде полей все характеристики\*, которые увидели.

Примечание\*: во избежание путаницы мы будем говорить характеристики, а не свойства (так как вы помните, что такое свойства, не так ли?).

# АНАЛИЗ И ДИЗАЙН

В виде класса это могло бы выглядеть так:

```
public class Movie {  
    private String imageUrl;  
    private String name;  
    private String genre;  
  
    // + getters/setters*  
}
```

Примечание\*: здесь и далее, если используются автоматические сгенерированные геттеры и сеттеры (без логики), мы не будем приводить полностью их код, а просто будем оставлять соответствующие комментарии.



# АНАЛИЗ И ДИЗАЙН

На самом деле, мы с вами провели целых две операции (пусть даже неявно):

1. **операцию анализа** — разложили картинку на составляющие, выделили объекты и их характеристики
2. **операцию дизайна\*** — из составляющих собрали описание объектов в виде класса и набора полей

Примечание\*: вы можете встретить также названия «проектирование», «синтез» и т.д. Термины сейчас очень размыты, главное, чтобы вы понимали, о чём идёт речь.



# АБСТРАКЦИЯ

Если бы перед нами стояла задача создать только систему, способную отображать этот блок (см. скриншот), мы бы на этом и остановились, так как других свойств мы не используем.

## Афиша



Вперёд  
мультфильм



Отель «Белград»  
комедия



Остров фантазий  
ужасы



Джентльмены  
боевик

*Вопрос к аудитории: согласны ли вы с этим утверждением?*



## ID (идентификатор)

Мы дадим вам совет, который сэкономит вам время: привыкните, что у каждого объекта, представляющего собой абстракцию данных, должно быть поле `id`.

**`id` — это уникальный идентификатор**, позволяющий найти этот объект среди других объектов.

Во всех системах, хранящих данные, есть идентификаторы: артикул, инвентарный номер и т.д. Их можно называть по-разному, но они должны быть.

---

## ID (идентификатор)

id не обязательно должен быть числом, хотя так делают достаточно часто.

Например, в качестве id для кино можно выбрать строку из ссылки (если кликнуть на тизер):

- вперёд — vpered-2020
- отель Белград — otel-belgrad
- остров фантазий — ostrov-fantazii-2020

---

# id

```
public class Movie {  
    private String id;  
    private String imageUrl;  
    private String name;  
    private String genre;  
  
    // + getters/setters*  
}
```

Ещё стоит добавить url, на который нужно переходить при клике по тизеру.

# АНАЛИЗ

Если мы прокрутим чуть дальше, то увидим, что наш анализ не полон:



Один вдох  
драма



Бладшот  
боевик



Пиноккио  
фэнтези



Кукла 2: Брамс  
ужасы



Вопросы к аудитории:

1. Как мы можем хранить эту информацию?
2. Как мы могли это предусмотреть заранее?

# КАК ХРАНИТЬ ИНФОРМАЦИЮ

Есть много вариантов, как хранить информацию:

1. Хранить **private boolean premiereTomorrow**;<sup>\*</sup>
2. Хранить дату премьеры<sup>\*\*</sup> и создать метод, определяющий, наступит завтра день премьеры или нет
3. Другие варианты

Но любой из них не является единственно верным.

Примечание<sup>\*</sup>: обратите внимание, что для **boolean** полей геттер генерируется как `isPremiereTomorrow`.

Примечание<sup>\*\*</sup>: время и даты - один из основных источников ошибок, об этом мы ещё будем говорить.



## КАК МЫ МОГЛИ ЭТО ПРЕДУСМОТРЕТЬ ЗАРАНЕЕ?

Ответ **для текущей ситуации**: выполнить анализ более тщательно, пролистав все доступные варианты.

Старайтесь не делать поспешных выводов только на основании просмотра первых доступных вариантов.

Вы знаете, что самое интересное — **вблизи границ** (первые, последние), и нужно не забыть кого-то из середины.



# КАК МЫ МОГЛИ ЭТО ПРЕДУСМОТРЕТЬ ЗАРАНЕЕ?

Ответ в целом: никак.

Мы можем попытаться предугадать возможные варианты (построить гипотезы) на основании:

- своего опыта (если мы являемся экспертами этой области)
- анализа конкурентов
- общения с заказчиками, пользователями и другими экспертами

Конечно же, если есть ТЗ, то нужно использовать информацию из него (но не факт, что составитель ТЗ тоже предусмотрел всё).



---

# КАК МЫ МОГЛИ ЭТО ПРЕДУСМОТРЕТЬ ЗАРАНЕЕ?

Ключевое: привыкните к тому, что предусмотреть заранее всё **невозможно**. И не расстраивайтесь, если вы чего-то не предусмотрели.

Системы развиваются, требования меняются. Поэтому, привыкните к тому, что вам придётся периодически выполнять **редизайн** (добавлять/удалять/изменять поля, методы, классы).



# ВЫВОДЫ

Из ответа на вопрос следуют выводы:

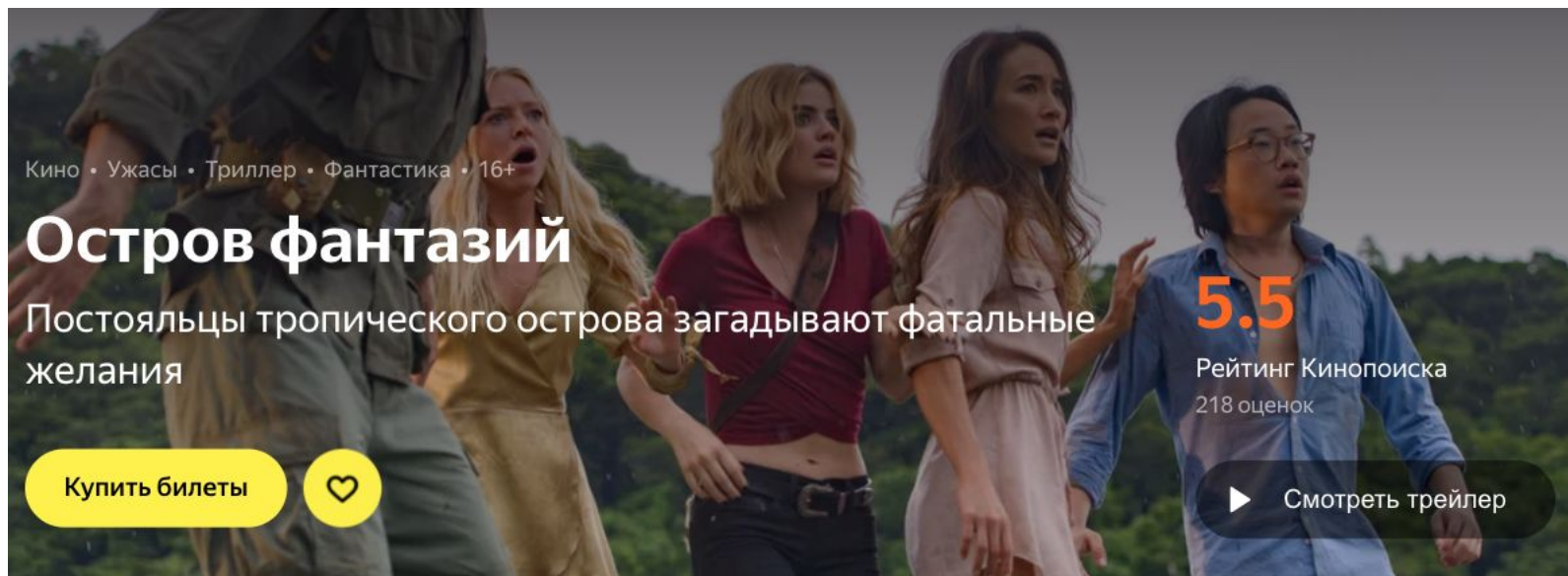
1. Нужно наращивать экспертизу в конкретной области
2. Нужно быть готовым к изменениям и уметь производить их безопасно

Если с экспертизой и моральной готовностью понятно, то что значит «проводить их безопасно»?

Нужно проверить, что после наших изменений всё работает «как нужно»: тесты (или автотесты) и **testability**.

# РЕДИЗАЙН

Мы разобрались с тем, как показывать «ленту». Но если мы перейдём по ссылке, то увидим:





# РЕДИЗАЙН

Что мы заметили? Полей, достаточных для отображения ленты, уже недостаточно для отображения странички фильма.

Поэтому, придётся сделать редизайн нашего класса и добавить в него необходимые поля.

# ВОПРОСЫ

Вопрос: что с методами? Как мы сделаем метод «купить билеты»?

Купить билеты



Кроме того, как объекты решают, какой из них будет отображаться в ленте и в каком порядке?



Один вдох  
драма



Бладшот  
боевик



Пиноккио  
фэнтези



Кукла 2: Брамс  
ужасы



# СЛОЖНОСТЬ

Если объекты научатся «сами себя продавать» и выстраиваться в нужном порядке, то это круто (как умные авто, которые коллективно паркуются, помогая друг другу), но **сложно**.

Задача программиста — бороться со сложностью, а не создавать её.  
Как «**всё упростить**», чтобы нам самим было проще?

---

# РЕАЛЬНЫЙ МИР

Представим, что мы идём за футболкой в магазин одежды. Там стопочками лежат футболки или висят на вешалках:



Изображение с сайта: [pixabay.com](https://pixabay.com)



# РЕАЛЬНЫЙ МИР

*Вопросы к аудитории:*

- *как футболки решают, кому лежать в стопочке или висеть на вешалке?*
- *как футболки решают, в каком порядке это делать?*
- *как футболки «продают себя»?*



---

# РЕАЛЬНЫЙ МИР

Правильный ответ: **никак\***.

**Это решает менеджер.** Он складывает футболки в нужном порядке, вешает на вешалку, а некоторые даже одевает на манекен! Футболка не продаёт сама себя\*, это тоже делает менеджер.

Футболка отвечает только за хранение своих данных — размер, цвет, цена и т.д.

Примечание\*: хотя, возможно, скоро и начнут 

# МЕНЕДЖЕР

А что если нам поступить так же? Договориться, что у нас будут **объекты, которые содержат только данные** (+ get/set к ним), и **объекты, которые управляют ими.**

Тогда наша работа делится на:

- объекты с данными (классы, описывающие их называются *data-classes*)
- объекты-менеджеры

# MovieManager

```
public class MovieManager {  
    private Movie[] movies;  
  
    public Movie[] getMoviesForFeed() {  
        // TODO: add logic  
        return null;  
    }  
}
```

Обратите внимание: никаких getter'ов и setter'ов здесь нет.



# МЕНЕДЖЕР

Таким образом, мы приходим к следующему:

- если объекты data-классов содержат только приватные поля + get/set к ним и не содержат логики\*, то тесты к ним не нужны
- объектам data-классов не нужно знать об окружающем мире, за это теперь будут отвечать менеджеры
- но для объектов-менеджеров тесты нужны, поскольку именно в них теперь будет сосредоточена вся бизнес-логика

Примечание\*: если содержат, то тесты к ним, конечно, нужны.



# МЕНЕДЖЕР

Менеджер на данном этапе будет отвечать за:

- хранение списка (в нашем случае массива) объектов
- добавление объекта в список
- удаление объекта из списка
- поиск объекта в списке
- другие операции (сортировка и т.д.)



## А КАК ЖЕ КОНДИЦИОНЕР?

Кондиционер — это два в одном: и данные, и менеджер для этих данных.

Важно: мы применяли с вами разделение там, где один менеджер управляет многими объектами.

В случае кондиционера у вас один объект, который сам отвечает за управление собой. Поэтому, отдельно выделять менеджера (в большинстве случаев\*) не нужно.

Примечание\*: мы говорим «в большинстве случаев», потому что не может быть идеальных правил на все случаи жизни.



# ИНКАПСУЛЯЦИЯ



# ИНКАПСУЛЯЦИЯ

**Инкапсуляция** — объединение данных и методов доступа к ним для предотвращения прямого доступа.

Мы уже обсуждали на примере кондиционера, что приватные поля + методы доступа (get/set) должны предотвратить установку температуры выше максимальной/ниже минимальной.





# ИНКАПСУЛЯЦИЯ

В задании с менеджером фильмов инкапсуляция позволяет нам оставить методы доступа для использования другими объектами.

Методы доступа, которые мы предоставляем другим объектам, называют ещё интерфейсом\* нашего объекта (API, которое мы обсуждали на лекции по JUnit).

Благодаря тому, что мы предоставляем только методы доступа, мы можем менять внутреннюю реализацию (так как она скрыта от остальных), не нарушая нашего с ними взаимодействия.

Примечание\*: не путайте с ключевым словом `interface` в Java.

---

# ИНКАПСУЛЯЦИЯ

На примере Афиши: менеджеру главной страницы Яндекса всё равно как менеджер Яндекс Афиши выбирает фильмы для показа ленты, главное, чтобы менеджер отдавал фильмы в нужном формате.

Таким образом, обеспечивается гибкость систем:

- в обычные дни показываем стандартную ленту
- в дни фестивалей можем показывать специализированную

При этом менеджер главной страницы Яндекса будет просто запрашивать у менеджера Афиши данные для ленты.

# МЕНЕДЖЕРЫ

```
public class MainPageManager {  
    private MovieManager movieManager;  
  
    /**  
     * Main Page generation  
     */  
    public String generatePage() {  
        Movie[] movies = movieManager.getMoviesForFeed();  
        // TODO: add logic  
        return null;  
    }  
}
```

Таким образом, правила все те же: все поля — приватные, методы доступа к ним — публичные.



# КОМПОЗИЦИЯ



# КОМПОЗИЦИЯ

**Композиция** — это подход, при котором мы строим объекты, собирая их (как конструктор) из других объектов.

Например, смартфон (это объект) состоит из:

- экран (и это тоже объект)
- корпус (и это объект)
- камера (и это объект)
- батарея (и это объект)
- процессор (и это объект) и т.д.

Таким образом, если собрать эти составляющие объекты в один, то получится смартфон.





# КОМПОЗИЦИЯ

Такое композиционное устройство позволяет делегировать ответственность за разработку разным людям или подразделениям а также заменять их.

Например, если у вашего смартфона сломался экран, то скорее всего, ваш старый экран просто «отклеят» и «приклеят» новый (причём это именно так и выглядит физически).



# АНАЛИЗ

*Вопрос к аудитории: давайте подумаем, композицией каких объектов является банкомат?*



Видеокамера, которая снимает лицо

Компьютер и система безопасности

Принтер для чеков

Считывающее устройство для карт

Кассеты с купюрами из банка

Кассета, куда попадают внесенные купюры

Механизм, который забирает купюры из кассет и выдает их в руки





## АНАЛИЗ

Возвращаясь к нашей задаче про фильмы и Яндекс: получается, что мы можем собирать главную страницу, делегируя разным менеджерам ответственность за предоставление собственных блоков (см. следующий слайд).

Найти

Видео

Картинки

Маркет

Карты

ещё

Погода



+4, днём +9

Почта



Войти

USD MOEX 71,40 +2,83   EUR MOEX 80,59 +3,08   Нефть 37,77 -16,55%   ещё ▾

## Афиша



Вперёд  
мультфильм



Отель «Белград»  
комедия



Остров фантазий  
ужасы



Джентльмены  
боевик



Один вдох  
драма


↩ Ближайшие кинотеатры



# КОМПОЗИЦИЯ

**Q:** Тогда получается, что главный менеджер должен содержать десяток других менеджеров? Но каждый менеджер же просто генерирует кусочек страницы, нельзя ли как-то унифицировать?

**A:** Можно.



# НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

# НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

Мы не будем давать сейчас определения этих терминов (им будет посвящена отдельная лекция) и объясним общую идею на примере.

Представим, что мы хотим «унифицировать» поведение всех менеджеров страницы:

```
public class MovieManager {  
    private Movie[] movies;  
  
    public String generateBlock() {  
        // TODO: add logic  
        return null;  
    }  
}
```

# НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

```
public class NewsManager {  
    private News[] news;  
  
    public String generateBlock() {  
        // TODO: add logic  
        return null;  
    }  
}
```

```
public class WeatherManager {  
    private int currentTemperature;  
    private int morningTemperature;  
    private int eveningTemperature;  
  
    public String generateBlock() {  
        // TODO: add logic  
        return null;  
    }  
}
```



# УНИФИКАЦИЯ

Мы не можем просто описать всех менеджеров одним классом, так как у них разный набор полей.

Но при этом их всех объединяет то, что они менеджеры, и у них есть метод `generateBlock`, который генерирует свой «кусочек» страницы.



---

# В РЕАЛЬНОЙ ЖИЗНИ

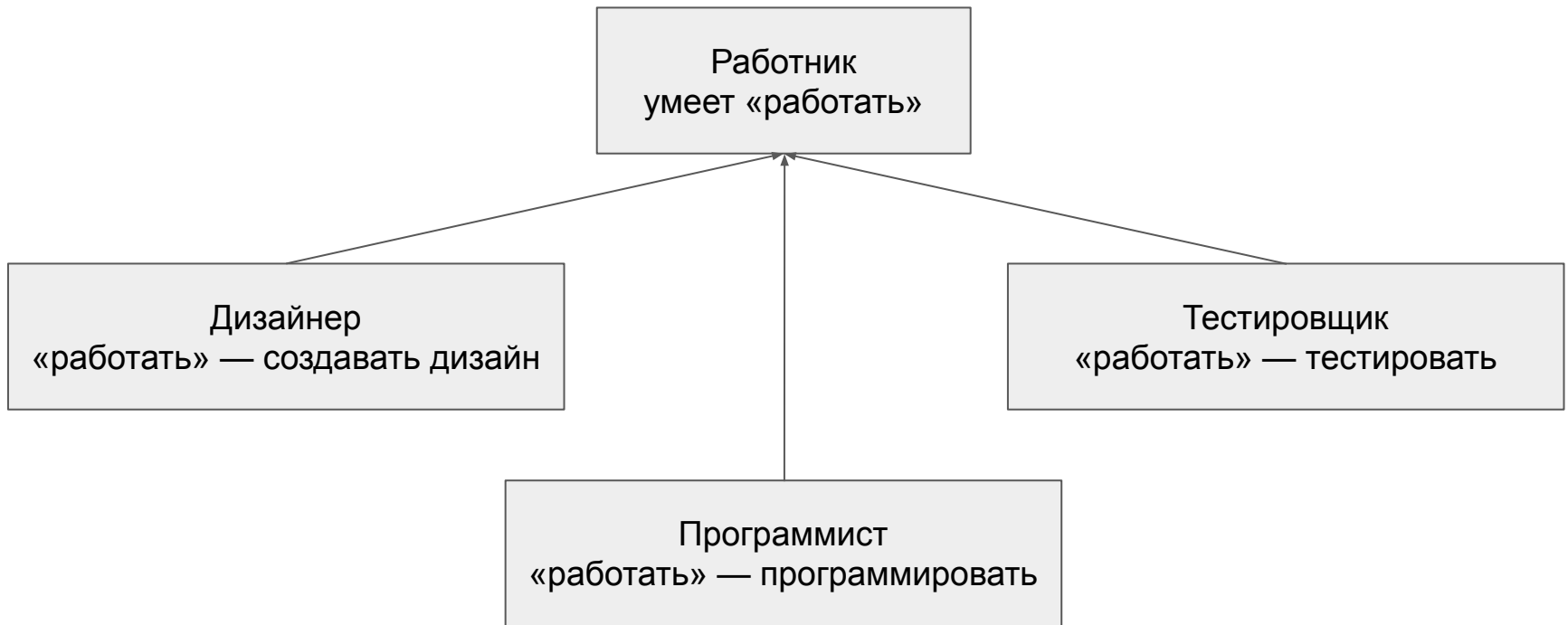
В реальной жизни менеджер приходит в IT-команду и говорит:  
«Работаем!».

При этом:

- для дизайнеров работать — это создавать дизайн
- для программистов — программировать
- для тестировщиков — тестировать
- РМ — собирать совещания, требовать отчёты, «всё исправить» 🐉\*  
и т.д.

Примечание\*: это, конечно, шутка. правильные РМ делают полезную и важную работу.

# В РЕАЛЬНОЙ ЖИЗНИ



При этом все они являются «работниками» (или сотрудниками).

# ПОЛИМОРФИЗМ И НАСЛЕДОВАНИЕ

Полиморфизм и наследование позволяют нам осуществить этот трюк в Java: мы можем называть всех менеджеров, кроме главного, `BlockManager`. У всех менеджеров должен быть метод `generateBlock`, но при этом этот метод будет у каждого свой.

Тогда задача главного менеджера страницы предельно проста:

1. собрать всех менеджеров
2. сказать каждому: «генерируй блок»
3. включить сгенерированный блок в страницу

Ему не нужно знать, что один генерирует фильмы, а другой — новости.

# МЕНЕДЖЕРЫ

```
public class MainPageManager {  
    private BlockManager[] managers;  
  
    /**  
     * Main Page generation  
     */  
    public String generate() {  
        for (BlockManager manager : managers) {  
            String block = manager.generateBlock();  
        }  
        // TODO: add logic  
        return null;  
    }  
}
```

с сюда можно класть любых менеджеров

но метод у каждого свой (со своей логикой)

# ПОЛИМОРФИЗМ И НАСЛЕДОВАНИЕ

Это позволит нам быстро и безопасно внедрять функции наподобие  
ВОТ ЭТОЙ:

## Блоки на странице



### Афиша

Расписание развлечений в вашем городе



### Коллекции

Идеи для вдохновения



### Пользователи обсуждают

Самое обсуждаемое на сервисах Яндекса



### Игры

Каталог браузерных игр



### Дзен

Публикации на основе ваших интересов



### Интересное в СМИ

Персональная подборка материалов СМИ



### ТВ онлайн

Прямой эфир телеканалов в Яндекс.Эфире

Отменить

Сохранить



# ИТОГИ



## ИТОГИ

Сегодня мы снова говорили об ООП и внесли для себя ключевое упрощение: мы можем при необходимости делить классы по назначению (менеджеры и data-классы).



# ИТОГИ

Мы практиковались в проектировании на примере анализа реальной системы.

**Самое важное** для вас как для тестировщика, программиста и автоматизатора — обязательно **нужно смотреть на:**

- реальные системы, существующие в объектах поля
- связи между объектами
- механику взаимодействия

Это самый быстрый способ наработки навыка анализа и проектирования систем.





## ИТОГИ

Мы верхнеуровнево рассмотрели ключевые принципы ООП на примерах. Дальше мы их будем внедрять в практику.

Важно, чтобы вы понимали, что эти принципы нужны для обеспечения:

- простоты
- гибкости
- безопасности

При правильном применении они также обеспечат и тестируемость.



# ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаём в чате Slack!
- Задачи можно сдавать по частям.
- Зачёт по домашней работе проставляется после того, как приняты **все задачи**.



Задавайте вопросы и напишите отзыв о лекции!

**ЛЮБОВЬ МАЯСОВА**



[@vlladanna](https://t.me/vlladanna)



[@LyubovMayasova](https://t.me/LyubovMayasova)