

**Вопросы для экзамена по предмету**  
**Шаблоны программных платформ на языке Джава**  
**2021-2022 уч год**

[Сделано \(#b6d7a8\)](#)  
[Почти готово\(#ffe599\)](#)  
[Вопросы \(#ea9999\)](#)  
[Дубликат\(#cfe2f3\)](#)

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61									

Практические:

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

Ещё практические

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26				

## СОДЕРЖАНИЕ

<b>1. Понятие шаблона проектирования, составляющие шаблона. Приведите примеры употребления шаблонов в неверных контекстах</b>	<b>5</b>
<b>2. Классификация паттернов проектирования, приведите несколько примеров паттернов каждого класса</b>	<b>6</b>
<b>3. Функциональные интерфейсы. Понятие функционального интерфейса и использование в программах на языке Джава</b>	<b>7</b>
<b>4. Определение и использование функциональных интерфейсов. Аннотирование функциональных интерфейсов</b>	<b>8</b>
<b>5. Понятие и использование лямбда-выражений в языке Джава. Примеры</b>	<b>9</b>
<b>6. Паттерн “Стратегия. Пример использования функций для параметризации поведения объектов</b>	<b>13</b>
<b>7. Понятие чистой функции. Побочные эффекты функций. Чистота функций в ООП. Пример</b>	<b>17</b>
<b>8. Неизменяемый класс. Преимущества использования неизменяемых классов. Пример</b>	<b>18</b>
<b>9. Использование неизменяемых полей класса. Преимущества неизменяемых объектов</b>	<b>20</b>
<b>10. Функциональное программирование. Понятие инварианта класса. Примеры</b>	<b>21</b>
<b>11. Особенности многопоточной работы в Джава, использование final для полей данных для обеспечения потокобезопасности</b>	<b>22</b>
<b>12. Основное назначение паттерна “Строитель” (Builder) для разработки программ на языке Джава</b>	<b>24</b>
<b>13. Стандартные функциональные интерфейсы в Джава и их методы:</b>	<b>29</b>
<b>14. Потоки Stream API в Джава и их использование</b>	<b>30</b>
<b>15. Статический импорт и его использование для программирования Stream API</b>	<b>33</b>
<b>16. Назначение метода stream() интерфейса Collection. Примеры</b>	<b>34</b>
<b>17. Работа со Stream API. Нетерминальные операции потока Stream&lt;T&gt;: назначение и использование</b>	<b>35</b>

<b>18. Работа со Stream API. Терминальные операции, назначение и использование. Примеры</b>	<b>38</b>
<b>19. Интерфейс SplitIterator и его методы</b>	<b>39</b>
<b>20. Понятие многопоточности и написание многопоточных программ на языке Джава</b>	<b>40</b>
<b>21. Потоки (threads) в Джава. Создание потоков в Джава программах Подходы к созданию потоков: через наследование и реализацию интерфейса Runnable Thread API. Запуск потока с Runnable</b>	<b>40</b>
<b>22. Потоки в Джава. Методы Thread API</b>	<b>43</b>
<b>23. Синхронизация потоков. Понятие синхронизации. Блок синхронизации</b>	<b>43</b>
<b>24. Использование синхронизации потоков для пары процессов producer-reader(производители/потребители)</b>	<b>45</b>
<b>25. Реализация блока synchronized на основе мониторов</b>	<b>45</b>
<b>26. Метод synchronized (lock), особенности его использования для потоков</b>	
<b>46</b>	
<b>27. Синхронизация потоков . Правила happens-before (hb)</b>	<b>48</b>
<b>28. Синхронизация потоков. Модификатор полей volatile и его использование</b>	<b>53</b>
<b>29. Синхронизация потоков. Атомарные операции. Механизм Wait/notify</b>	
<b>56</b>	
<b>30. Синхронизация потоков с использованием классов пакета java.util.concurrent.locks</b>	<b>60</b>
<b>31. Запуск и прерывание потоков. Приостановка и прерывание выполнения нити</b>	<b>63</b>
<b>32. Обработка операции прерывания потока</b>	<b>63</b>
<b>33. Ожидание и присоединение запущенной нити основным потоком управления</b>	<b>66</b>
<b>34. Жизненный цикл потока на языке Джава. Состояние потока</b>	<b>70</b>
<b>35. Многопоточные примитивы и их использование</b>	<b>72</b>
<b>36. Интерфейс Executor в Джава и его использование. ExecutorService</b>	<b>72</b>

<b>37. Интерфейс Future в Джава. Основное назначение использования его в программах</b>	<b>75</b>
<b>38. Коллекции java.util.concurrent. Состав коллекции. Основные методы</b>	
77	
<b>39. Потокобезопасные коллекции пакета java.util.concurrent:</b>	<b>80</b>
<b>40. Реализация асинхронного выполнения в Джава</b>	<b>81</b>
<b>41. Паттерн “Одиночка” (Singleton) и его использование в Джава программах</b>	<b>82</b>
<b>42. Использование статических методов для создания экземпляра объекта вместо конструкторов</b>	<b>85</b>
<b>43. Паттерн “Строитель” и его использование большом количестве параметров конструктора</b>	<b>87</b>
<b>44. Понятие dependency injection (внедрение зависимости). Преимущества использования внедрения зависимостей при написании программ на языке Джава. Преимущества этого подхода перед паттерном Одиночка</b>	<b>88</b>
<b>45. Преимущество использования try-с-ресурсами по сравнению с использованием try-finally. Интерфейс интерфейса AutoCloseable</b>	<b>88</b>
<b>46. Паттерн Декоратор (Decorator) Преимущества его использования (например композиция по сравнению с наследованием)</b>	<b>90</b>
<b>47. Преимущества использования списков перед массивами</b>	<b>92</b>
<b>48. Основные системы сборки Gradle и Maven. Использование Gradle и основная терминология. Управление зависимостями в Gradle</b>	<b>93</b>
<b>49. Анатомия jar. Сканирование пакетов</b>	<b>96</b>
<b>50. Реализация REST API с помощью Spring Framework</b>	<b>96</b>
<b>51. Понятие Инверсии управления</b>	<b>98</b>
<b>52. Spring Boot и его использование</b>	<b>99</b>
В чем разница между Spring и Spring Boot?	99
<b>53. Работа с базами данных в джава приложениях. Встроенные СУБД для Джава приложений.</b>	<b>101</b>
<b>54. Реляционные СУБД для работы с джава приложениями. Пакет Пакет java.sql и его классы</b>	<b>101</b>
<b>55. Роль интерфейса JDBC для работы с джава приложениями</b>	<b>103</b>

<b>56. Основные компоненты JDBC API</b>	<b>103</b>
<b>57. JDBC URL и его использование</b>	<b>104</b>
<b>58. Работа JDBC драйвера</b>	<b>104</b>
<b>59. В чем заключается роль DI в Spring. Использования ServiceLoader</b>	<b>105</b>
<b>60. Интерфейс Connection. Пулы соединений</b>	<b>105</b>
<b>61. Принципы SOLID и их использование на Джава</b>	<b>106</b>

## **1. Понятие шаблона проектирования, составляющие шаблона. Приведите примеры употребления шаблонов в неверных контекстах**

*Шаблон проектирования* или *паттерн* (design pattern) в разработке ПО — повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

*Шаблон* – не законченный образец, не код; это пример решения задачи, который можно использовать в различных ситуациях.

Примеры неверного употребления:

- злоупотребление Singleton. Глобальные переменные не всегда хорошо
- усложнение программы за счет паттернов
- применение паттерна без оснований(Золотой молоток)
- Костыли для слабого языка программирования

Нужда в паттернах появляется тогда, когда люди выбирают для своего проекта язык программирования с недостаточным уровнем абстракции. В этом случае, паттерны — это костыль, который придаёт этому языку суперспособности.

Например, паттерн Стратегия в современных языках можно реализовать простой анонимной (лямбда) функцией.

Ещё паттерны могут показаться похожими на алгоритмы. Но у них есть разница. Алгоритм состоит из конкретных шагов, описывающих необходимые действия. Паттерны же лишь описывают подход, но не описывают шаги реализации.

## **2. Классификация паттернов проектирования, приведите несколько примеров паттернов каждого класса**

Паттерны бывают разные, т.к. решают разные проблемы. Обычно выделяют следующие категории:

- Порождающие  
Эти паттерны решают проблемы обеспечения гибкости создания объектов
- Структурные  
Эти паттерны решают проблемы эффективного построения связей между объектами
- Поведенческие  
Эти паттерны решают проблемы эффективного взаимодействия между объектами
- 

- *Порождающие* (Creational) - Абстрактная фабрика, строитель, фабричный метод

Беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей.

- *Структурные* (Structural) - Адаптер, мост, декоратор, фасад

Показывают различные способы построения связей между объектами.

- *Поведенческие* (Behavioral) - Цепочка обязанностей, команда, итератор

Заботятся об эффективной коммуникации между объектами

### **3. Функциональные интерфейсы. Понятие функционального интерфейса и использование в программах на языке Джава**

В Java функциями являются объекты, реализующие *функциональные интерфейсы* – интерфейсы с только одним абстрактным методом.

Список стандартных функциональных интерфейсов: вопрос 13.

```
@FunctionalInterface  
public interface Function<T, R> {  
    R apply(T arg);  
}
```

```
@FunctionalInterface  
public interface Converter<T, N> {  
  
    N convert(T t);  
  
    static <T> boolean isNotNull(T t){  
        return t != null;  
    }  
  
    default void writeToConsole(T t) {  
        System.out.println("Текущий объект - " + t.toString());  
    }  
  
    boolean equals(Object obj);  
}
```

## 4. Определение и использование функциональных интерфейсов. Аннотирование функциональных интерфейсов

Функциональные интерфейсы — это мощный механизм, предоставляемый Java 8, который дает нам возможность передавать функции в качестве параметров другим методам.

Аннотация `@FunctionalInterface` необязательная. Компилятор проверяет для интерфейсов, помеченных этой аннотацией, что в нем только один абстрактный метод.

Аннотация `@FunctionalInterface` используется, чтобы разработчики модуля А случайно не сломали зависящие от него модули. В случае ее наличия ошибки компиляции будут в модуле А.

Модуль А	Модуль В
Версия 1: <pre>interface UserDetails {     String getUserName(); }</pre>	UserDetails unknown = () -> "unknown";
Версия 2: <pre>interface UserDetails {     String getUserName();     String getUserAddress(); }</pre>	Ошибка компиляции: <pre>UserDetails unknown = () -&gt; "unknown";</pre> UserDetails больше не функциональный интерфейс! Нужно использовать анонимный класс.

```
@FunctionalInterface
public interface Converter<T, N> {

    N convert(T t);

    static <T> boolean isNotNull(T t){
        return t != null;
    }

    default void writeToConsole(T t) {
        System.out.println("Текущий объект - " + t.toString());
    }

    boolean equals(Object obj);
}
```

## 5. Понятие и использование лямбда-выражений в языке Java. Примеры

Лямбда-выражение или просто лямбда в Java — упрощённая запись анонимного класса, реализующего функциональный интерфейс.

Функциональный интерфейс в Java — интерфейс, в котором объявлен только один абстрактный метод. Однако, методов по умолчанию (default) такой интерфейс может содержать сколько угодно, что можно видеть на примере `java.util.function.Function`. Функциональный интерфейс может быть отмечен аннотацией `@FunctionalInterface`, но это не обязательное условие, так как JVM считает функциональным любой интерфейс с одним абстрактным методом.

Lambda-выражения в Java обычно имеют следующий синтаксис

(аргументы) -> (тело). Например:

(арг1, арг2...) -> { тело }

(тип1 арг1, тип2 арг2...) -> { тело }

Далее идет несколько примеров настоящих Lambda-выражений:

```
(int a, int b) -> { return a + b; }
```

```
() -> System.out.println("Hello World");
```

```
(String s) -> { System.out.println(s); }
```

```
() -> 42
```

```
() -> { return 3.1415 };
```

## Примеры Lambda-выражений

Поток Thread можно проинициализировать двумя способами:

```
// Старый способ:
```

```
new Thread(new Runnable() {
```

```
    @Override
```

```
    public void run() {
```

```
        System.out.println("Hello from thread");
```

```
}
```

```
}).start();  
  
// Новый способ:  
new Thread(  
  
() -> System.out.println("Hello from thread"))  
  
.start();
```

Управление событиями в Java 8 также можно осуществлять через Lambda-выражения. Далее представлены два способа добавления обработчика события `ActionListener` в компонент пользовательского интерфейса:

```
// Старый способ:  
  
button.addActionListener(new ActionListener() {  
  
    @Override  
  
    public void actionPerformed(ActionEvent e) {  
  
        System.out.println("Кнопка нажата. Старый  
способ!"); } } );  
  
// Новый способ:  
  
button.addActionListener( (e) -> {  
  
    System.out.println("Кнопка нажата. Lambda!");
```

```
} );
```

Простой пример вывода всех элементов заданного массива. Заметьте, что есть более одного способа использования lambda-выражения. Ниже мы создаем lambda-выражение обычным способом, используя синтаксис стрелки, а также мы используем оператор двойного двоеточия ( :: ), который в Java 8 конвертирует обычный метод в lambda-выражение:

```
// Старый способ:  
  
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6,  
7);  
for(Integer n: list) { System.out.println(n);}  
  
// Новый способ:  
  
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6,  
7);  
list.forEach(n -> System.out.println(n));
```

- Лямбда-выражение является блоком кода с параметрами.
- Используйте лямбда-выражение, когда хотите выполнить блок кода в более поздний момент времени.
- Лямбда-выражения могут быть преобразованы в функциональные интерфейсы.
- Лямбда-выражения имеют доступ к final переменным из охватывающей области видимости.

```
import java.util.concurrent.atomic.AtomicInteger;

/**
 * Output is:
 * 1
 * -7
 * 2
 * 0
 * 3
 * -1
 */
public class Main {
    static int d = 10;

    public static void main(String... $) {
        // static method reference
        a(Main::a);

        { // lambdas accessing different variables
            int a = 1;
            AtomicInteger b = new AtomicInteger(2);
            b.set(-2);
            d = 11; // effectively final variable

            a(c -> (int) c + a - b.get() - d);

            // compiler will throw an error 'cause 'a'
            // is local and it's being modified by a lambda
            // a(c -> { a = 2; return a; });

            // but this will work just fine 'cause 'd' is e.f.
            a(c -> { d = 2; return d; });

            // and this too as the variable itself isn't
            // being modified by a lambda
            a(c -> { b.set(0); return b.get(); });
        }

        // object method reference
        class B implements A {
            @Override public int a(long a)
            { return (int) a * 3; }
        }
        a(new B()::a);

        // anonymous class
        a(new A() {
            @Override public int a(long b)
            { return (int) -b; }
        });
    }

    static void a(A a)
    { System.out.println(a.a(1)); }

    static int a(long a) { return (int) a; }

    @FunctionalInterface interface A {
        int a(long a);
    }
}
```

```
    @Override boolean equals(Object a);  
}  
}
```

## 6. Паттерн “Стратегия. Пример использования функций для параметризации поведения объектов

Шаблон «Стратегия» относится к группе поведенческих шаблонов.

### Краткое определение шаблона «Стратегия»

Шаблон служит для переключения между семейством алгоритмов, когда объект меняет свое поведение, на основании изменения своего внутреннего состояния.

Картинка с Рефакторинг гуру для большего понимания?

### Практические примеры применения шаблона «Стратегия»

- Сортировка (sorting): мы хотим отсортировать эти числа, но мы не знаем, будем ли мы использовать BrickSort, BubbleSort или какую-либо другую сортировку. Например, у вас есть веб-сайт, на котором страница отображает элементы в зависимости от популярности. Однако «Популярным» может быть много вещей (большинство просмотров, большинство подписчиков, дата создания, большая активность, наименьшее количество комментариев). В случае, если руководство еще не знает точно, как сделать заказ, и может захотеть поэкспериментировать с различными заказами на более поздний срок, вы создаете интерфейс (IOrderAlgorithm или что-то еще) с методом

заказа и позволяет объекту Orderer делегировать порядок конкретной реализации интерфейса IOrderAlgorithm. Вы можете создать «CommentOrderer», «ActivityOrderer» и т. д. и просто отключить их при появлении новых требований.

- Игры (games): стратегии перемещения в пространстве игры — игрок ходит, либо бегает, но, возможно, в будущем он также сможет плавать, летать, телепортироваться, рыть под землей и др. Другой пример, когда в игре, например с различными персонажами, где каждый персонаж может иметь разные виды оружия, но в конкретный момент времени может использовать только одно из них. Так что контекстом здесь является персонаж «Король», «Командир», «Солдат» и оружие как стратегия где метод атаковать Attack() зависит от вида оружия. Так что, если конкретные классы оружия могут быть «Меч», «Топор», «Арбалет», «Лук и Стрелы» они все должны иметь метод Attack () .
- Хранение информации (storing information): стратегия сохранения информации: допустим, приложение имеет задачу сохранять информацию в базу данных. Но в дальнейшем может пригодиться возможность сохранения информации в файл или облачное хранилище. Именно из-за различия алгоритмов сохранения данных здесь применим паттерн Стратегия. Здесь контекстом будет объект, который сохраняет данные, а стратегией будет алгоритм сохранения информации.

В итоге: Паттерн «Стратегия» определяет семейство алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость. Он позволяет модифицировать алгоритмы независимо от их использования на стороне клиента.

Пример:

```
// Класс реализующий конкретную стратегию, должен реализовывать
// этот интерфейс
// Класс контекста использует этот интерфейс для вызова
// конкретной стратегии
interface Strategy {
    int execute(int a, int b);
}

// Реализуем алгоритм с использованием интерфейса стратегии
class ConcreteStrategyAdd implements Strategy {

    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategyAdd's
execute()");
        return a + b; // Do an addition with a and b
    }
}

class ConcreteStrategySubtract implements Strategy {

    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategySubtract's
execute()");
        return a - b; // Do a subtraction with a and b
    }
}

class ConcreteStrategyMultiply implements Strategy {

    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategyMultiply's
execute()");
        return a * b; // Do a multiplication with a and b
    }
}
```

```
// Класс контекста использующий интерфейс стратегии
class Context {

    private Strategy strategy;

    // Constructor
    public Context() {
    }

    // Set new strategy
    public void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }

    public int executeStrategy(int a, int b) {
        return strategy.execute(a, b);
    }
}

// Тестовое приложение
class StrategyExample {

    public static void main(String[] args) {

        Context context = new Context();

        context.setStrategy(new ConcreteStrategyAdd());
        int resultA = context.executeStrategy(3,4);
        context.setStrategy(new ConcreteStrategySubtract());
        int resultB = context.executeStrategy(3,4);

        context.setStrategy(new ConcreteStrategyMultiply());
        int resultC = context.executeStrategy(3,4);

        System.out.println("Result A : " + resultA );
        System.out.println("Result B : " + resultB );
    }
}
```

```
        System.out.println("Result C : " + resultC );  
    }  
}
```

## 7. Понятие чистой функции. Побочные эффекты функций.

### Чистота функций в ООП. Пример

*Чистая (pure) функция* – функция без побочных эффектов, результат которой зависит только от значений ее параметров.

В императивных языках некоторые функции в процессе выполнения своих вычислений могут модифицировать значения глобальных переменных, осуществлять операции ввода-вывода, реагировать на исключительные ситуации, вызывая их обработчики. Такие функции называются **функциями с побочными эффектами**. Другим видом побочных эффектов является модификация переданных в функцию параметров (переменных), когда в процессе вычисления выходного значения функции изменяется и значение входного параметра.

Описывать функции без побочных эффектов позволяет практически любой язык программирования. Однако некоторые языки поощряют или даже требуют от некоторых видов функций использования побочных эффектов. Например, во многих объектно-ориентированных языках в функцию-член класса передаётся скрытый параметр — указатель на экземпляр класса, от имени которого вызывается соответствующая функция (например, в C++ этот параметр называется `this`, а в Object Pascal — `self`), который эта функция неявно модифицирует. Тем не менее, в языке C++ можно указать для метода класса модификатор `const`, тем самым сообщив компилятору о том, что метод не модифицирует данные класса.

Свойства чистой функции?

```

class Point {
    int x;
    int y;
    void move(int dx) {
        this.x += dx; // побочный эффект
    }
    int getX() { return x; } // не является чистой, т.к. разные вызовы
                                // могут вернуть разные значения
}

```

```

class Point {
    private int x;
    private int y;
    Point(int x, int y) { this.x = x; this.y = y; }
    Point move(int dx) {
        return new Point(this.x + dx, y); // нет побочных эффектов
    }
    int getX() { return x; }
}

```

И move, и getX – чистые функции (в данном случае this мы считаем неявным параметром функции).

[online.mirea.ru](http://online.mirea.ru)

## 8. Неизменяемый класс. Преимущества использования неизменяемых классов. Пример

В ООП чистота функций (методов) достигается с помощью использования неизменяемых (immutable) классов. Все методы неизменяемого класса являются чистыми. После создания объекта неизменяемого класса его состояние больше не меняется. При операциях с такими объектами создаются новые объекты, а не меняются существующие. Для того, чтобы гарантировать неизменяемость свойств объекта, следует использовать модификатор final для полей. Также неизменяемый класс как правило закрыт для наследования, т.к. иначе класс-наследник может добавить изменяемые поля.

Для того чтобы сделать класс неизменяемым, необходимо выполнить следующие условия:

1. Не предоставляйте сеттеры или методы, которые изменяют поля или объекты, ссылающиеся на поля. Сеттеры подразумевают изменение состояния объекта а это то, чего мы хотим тут избежать.
2. Сделайте все поля final и private. Поля, обозначенные private, будут недоступными снаружи класса, а обозначение их final гарантирует, что вы не измените их даже случайно.
3. Не разрешайте субклассам переопределять методы. Самый простой способ это сделать – объявить класс как final. Финализированные классы в Java не могут быть переопределены.
4. Всегда помните, что ваши экземпляры переменных могут быть либо изменяемыми, либо неизменяемыми. Определите их и возвращайте новые объекты со скопированным содержимым для всех изменяемых объектов (ссылочные типы). Неизменяемые переменные (примитивные типы) могут быть безопасно возвращены без дополнительных усилий.

```
import java.util.Map;
public final class MutableClass {
    private String field;
    private Map<String, String> fieldMap;
    public MutableClass(String field, Map<String, String> fieldMap) {
        this.field = field;
        this.fieldMap = fieldMap;
    }
    public String getField() {
        return field;
    }
    public Map<String, String> getFieldMap() {
        return fieldMap;
    }
}
```

## 9. Использование неизменяемых полей класса.

### Преимущества неизменяемых объектов

Плюсы неизменяемых объектов:

1. Проще понять работу класса
  1. Отсутствует “действие на расстоянии”
  2. Результат работы программы не зависит от истории
  3. Проще проверять инварианты класса
2. Проще тестировать класс
3. Проще использовать в многопоточной среде
4. Можно использовать в качестве ключей Map

Поскольку внутреннее состояние неизменяемого объекта остается постоянным во времени, мы можем безопасно разделить его между несколькими потоками .

Мы также можем свободно использовать его, и ни один из объектов, ссылающихся на него, не заметит никакой разницы, мы можем сказать, что неизменяемые объекты не имеют побочных эффектов .

## 10. Функциональное программирование. Понятие инварианта класса. Примеры

Стиль программирования, основные черты:

1. Предпочтение чистых функций (отсутствие побочных эффектов)

2. Функции как объекты первого класса (*first-class citizen*):

1. Могут храниться в переменных

2. Могут передаваться как параметры

3. Могут возвращаться как результат

Функциональное программирование — парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании).

*Функциональное программирование - это программирование с использованием математических функций. Для преобразования методов в математические функции нам нужно сделать их сигнатуры честными в том смысле, что они должны полностью отражать все возможные входные данные и результаты, и нам также необходимо убедиться, что метод работает только с теми значениями, которые мы передаем, и ничего больше.*

*Практики, которые помогают преобразовать методы в математические функции:*

- *Иммутабельность.*
- *Избегать исключения для управления потоком программы.*
- *Избавляться от примитивной одержимости.*
- *Делать null явными.*

*Инвариант класса* – условие, которое должно выполняться на протяжении всего срока жизни объекта. В случае изменяемого объекта инвариант нужно проверять в каждом методе, изменяющем состояние (mutator method). Это можно забыть сделать, и инвариант будет нарушен. В неизменяемом классе инвариант достаточно проверять в конструкторе.

```
// Инвариант: a < b
class OrderedPair {
    private int a;
    private int b;

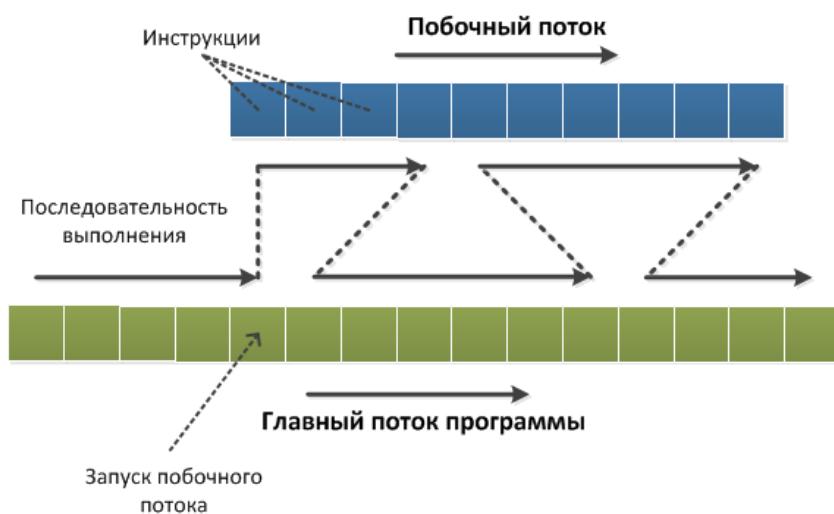
    OrderedPair(int a, int b) {
        check(a < b);
        this.a = a;
        this.b = b;
    }
}
```

## 11. Особенности многопоточной работы в Джава, использование final для полей данных для обеспечения потокобезопасности

Один поток – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса.

Следует отдельно обговорить фразу «параллельно с другими потоками». Известно, что на одно ядро процессора, в каждый момент времени, приходится одна единица исполнения. То есть одноядерный процессор может обрабатывать команды только последовательно, по одной за раз (в упрощенном случае). Однако запуск нескольких параллельных потоков возможен и в системах с одноядерными процессорами. В этом случае система будет периодически переключаться между потоками, поочередно давая выполняться то одному, то другому потоку. Такая схема называется псевдо-параллелизмом. Система запоминает состояние (контекст) каждого потока, перед тем как переключиться на другой поток, и восстанавливает его по возвращению к выполнению потока. В контекст потока входят такие параметры, как стек, набор значений регистров процессора, адрес исполняемой команды и прочее...

Проще говоря, при псевдопараллельном выполнении потоков процессор мечется между выполнением нескольких потоков, выполняя по очереди часть каждого из них.



```
public class Program          //Класс с методом main().
{
    public static void main(String[] args)
    {
        //Создание потока
        Thread myThready = new Thread(new Runnable()
        {
            public void run() //Этот метод будет выполняться в побочном потоке
            {
                System.out.println("Привет из побочного потока!");
            }
        });
        myThready.start(); //Запуск потока

        System.out.println("Главный поток завершён...");
    }
}
```

В Java процесс завершается тогда, когда завершается последний его поток. Даже если метод `main()` уже завершился, но еще выполняются порожденные им потоки, система будет ждать их завершения.

Однако это правило не относится к особому виду потоков – демонам. Если завершился последний обычный поток процесса, и остались только потоки-демоны, то они будут принудительно завершены и выполнение процесса закончится. Чаще всего потоки-демоны используются для выполнения фоновых задач, обслуживающих процесс в течение его жизни.

Объявить поток демоном достаточно просто — нужно перед запуском потока вызвать его метод `setDaemon(true);`

Проверить, является ли поток демоном, можно вызвав его метод `boolean isDaemon();`

```

//Троекратное изменение действия инкременатора
//с интервалом в i*2 секунд
for(int i = 1; i <= 3; i++)
{
    try{
        Thread.sleep(i*2*1000); //Ожидание в течении i*2 сек.
    }catch(InterruptedException e){}
}

mInc.changeAction();      //Переключение действия
}

mInc.finish(); //Инициация завершения побочного потока
}

```

В языке Java модификатор **final** для полей имеет особый смысл при многопоточной работе: к полю с этим модификатором могут безопасно обращаться одновременно несколько потоков, и они гарантированно будут видеть одинаковое значение.

Для не-final полей без особой синхронизации потоков может быть так, что каждый поток видит свое значение поля.

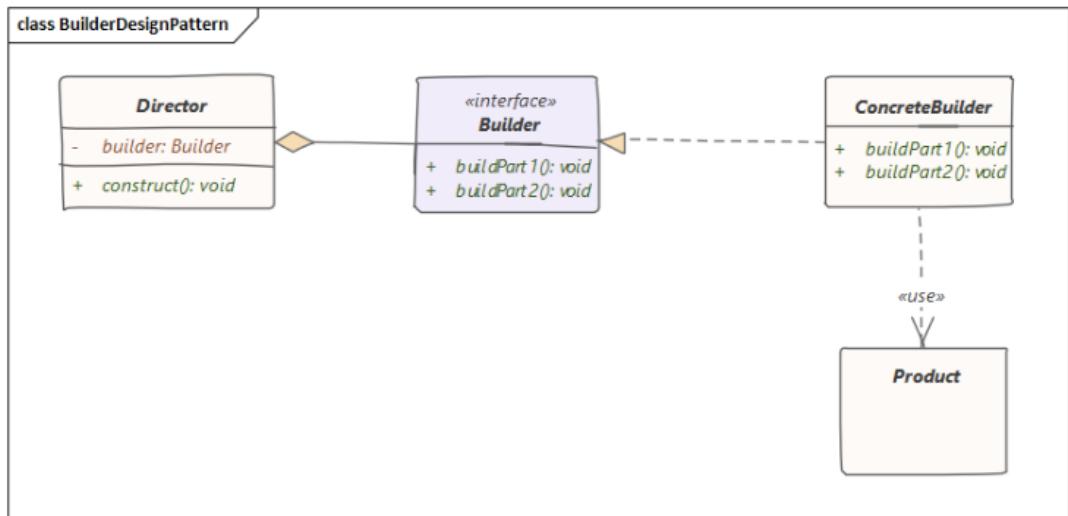
## 12. Основное назначение паттерна “Строитель” (Builder) для разработки программ на языке Джава

**Строитель** (англ. **Builder**) — порождающий шаблон проектирования предоставляет способ создания составного объекта. Отделяет конструирование сложного объекта от его представления так, что в результате одного и того же процесса конструирования могут получаться разные представления. позволяет изменять внутреннее представление продукта; изолирует код, реализующий конструирование и представление; дает более тонкий контроль над процессом конструирования.

Паттерн *Builder* позволяет использовать изменяемый объект для задания свойств, а затем создания на их основе неизменяемого объекта. Как

правило изменяемый Builder существует в рамках одного метода и снаружи не виден, так что это не нарушает функциональной чистоты.

Картинка с Рефакторинг гуру для большего понимания?



Итак, паттерн проектирования Builder можно разбить на следующие важные компоненты:

- **Product** (продукт) - Класс, который определяет сложный объект, который мы пытаемся шаг за шагом сконструировать, используя простые объекты.
- **Builder** (строитель) - абстрактный класс/интерфейс, который определяет все этапы, необходимые для производства сложного объекта-продукта. Как правило, здесь объявляются (абстрактно) все этапы (**buildPart**), а их реализация относится к классам конкретных строителей (**ConcreteBuilder**).
- **ConcreteBuilder** (конкретный строитель) - класс-строитель, который предоставляет фактический код для создания объекта-продукта. У нас может быть несколько разных **ConcreteBuilder**-классов, каждый из которых реализует различную разновидность или способ создания объекта-продукта.
- **Director** (распорядитель) - супервизионный класс, под контролем которого строитель выполняет скоординированные этапы для создания объекта-продукта. Распорядитель обычно получает на вход строителя с этапами на выполнение в четком порядке для построения объекта-продукта.

## Пример со сборкой автомобилей с использованием паттерна проектирования Builder

**Шаг 1:** Создайте класс **Car** (автомобиль), который в нашем примере является продуктом:

```
package org.trishinfotech.builder;

public class Car {

    private String chassis;
    private String body;
    private String paint;
    private String interior;

    public Car() {
        super();
    }

    public Car(String chassis, String body, String paint, String interior) {
        this();
        this.chassis = chassis;
        this.body = body;
        this.paint = paint;
        this.interior = interior;
    }

    public String getChassis() {
        return chassis;
    }
}
```

**Шаг 2:** Создайте абстрактный класс/интерфейс **CarBuilder**, в котором определите все необходимые шаги для создания автомобиля.

```
package org.trishinfotech.builder;

public interface CarBuilder {

    // Этап 1
    public CarBuilder fixChassis();

    // Этап 2
    public CarBuilder fixBody();

    // Этап 3
    public CarBuilder paint();

    // Этап 4

    public CarBuilder fixInterior();
}
```

**Шаг 3:** Теперь пора написать `ConcreteBuilder`. Как я уже упоминал, у нас могут быть разные варианты `ConcreteBuilder`, и каждый из них выполняет сборку по-своему, чтобы предоставить нам различные представления сложного объекта `Car`.

Итак, ниже приведен код `ClassicCarBuilder`, который собирает старые модели автомобилей.

```
package org.trishinfotech.builder;

public class ClassicCarBuilder implements CarBuilder {

    private String chassis;
    private String body;
    private String paint;
    private String interior;

    public ClassicCarBuilder() {
        super();
    }

    @Override
    public CarBuilder fixChassis() {
        System.out.println("Assembling chassis of the classical model");
        this.chassis = "Classic Chassis";
        return this;
    }

    @Override
    public CarBuilder fixBody() {
        System.out.println("Assembling body of the classical model");
        this.body = "Classic Body";
        return this;
    }

    @Override
    public CarBuilder paint() {
        System.out.println("Painting body of the classical model");
        this.paint = "Classic White Paint";
        return this;
    }

    @Override
    public Car build() {
        Car car = new Car(chassis, body, paint, interior);
        if (car.doQualityCheck()) {
            return car;
        } else {
            System.out.println("Car assembly is incomplete. Can't deliver!");
        }
        return null;
    }
}
```

**Шаг 4:** Теперь мы напишем класс-распорядитель `AutomotiveEngineer`, под руководством которого строитель будет собирать автомобиль (объект `Car`) шаг за шагом в четко определенном порядке.

```
package org.trishinfotech.builder;

public class AutomotiveEngineer {

    private CarBuilder builder;

    public AutomotiveEngineer(CarBuilder builder) {
        super();
        this.builder = builder;
        if (this.builder == null) {
            throw new IllegalArgumentException("Automotive Engineer can't work without Car");
        }
    }

    public Car manufactureCar() {
        return builder.fixChassis().fixBody().paint().fixInterior().build();
    }

}
```

Мы видим, что метод `manufactureCar` вызывает этапы сборки автомобиля в правильном порядке.

Теперь пришло время написать класс `Main` для выполнения и тестирования нашего кода.

```
package org.trishinfotech.builder;

public class Main {

    public static void main(String[] args) {
        CarBuilder builder = new SportsCarBuilder();
        AutomotiveEngineer engineer = new AutomotiveEngineer(builder);
        Car car = engineer.manufactureCar();
        if (car != null) {
            System.out.println("Below car delivered: ");
            System.out.println("=====");
            System.out.println(car);
            System.out.println("=====");
        }
    }
}
```

## **13. Стандартные функциональные интерфейсы в Джава и их методы:**

Стандартные функциональные интерфейсы и их методы:

1. Function: R apply(T arg) - представляет функцию перехода от объекта типа T к объекту типа R

2. Supplier: T get() - не принимает никаких аргументов, но должен возвращать объект типа T

3. Consumer: void accept(T arg) - выполняет некоторое действие над объектом типа T, при этом ничего не возвращая

4. Predicate: boolean test(T arg) - проверяет соблюдение некоторого условия. Если оно соблюдается, то возвращается значение true. В качестве параметра лямбда-выражение принимает объект типа T

5. BiFunction: R apply(T arg1, U arg2)

6. BiConsumer: void accept(T arg1, U arg2)

7. BiPredicate: boolean test(T arg1, U arg2)

Также есть варианты этих интерфейсов для примитивных типов int, long и double.

В своих программах имеет смысл определять свои функциональные интерфейсы, а не пользоваться предопределенными, потому что:

1. Название интерфейса и метода лучше документирует его назначение
2. Можно использовать конкретные типы вместо типовых параметров, что облегчает чтение
3. В среде разработки будет проще найти реализации этого интерфейса
4. Стандартные интерфейсы не предусматривают использование checked exceptions

## 14. Потоки Stream API в Джава и их использование

Можно взять кусок из номера 11 - *Нельзя Thread - поток и Stream - поток. Но Thread это поток выполнения, а Stream это поток данных.*

Stream API — это новый способ работать со структурами данных в функциональном стиле. Stream (поток) API (описание способов, которыми одна компьютерная программа может взаимодействовать с другой программой) — это по своей сути поток данных. Сам термин "поток" довольно размыт в программировании в целом и в Java в частности.

Поток (Stream) – это представление последовательности элементов, над которым можно производить операции. Операции делятся на две категории:

1. Нетерминальные(конвейерные) – их результат тоже является потоком, и к нему в свою очередь можно тоже применить операцию;

### 2.1 Краткое описание конвейерных методов работы со стримами

Метод stream	Описание	Пример
<b>filter</b>	Отфильтровывает записи, возвращает только записи, соответствующие условию	collection.stream().filter(``a1``::equals).count()
<b>skip</b>	Позволяет пропустить N первых элементов	collection.stream().skip(collection.size() - 1).findFirst().orElse(``1``)
<b>distinct</b>	Возвращает стрим без дубликатов (для метода equals)	collection.stream().distinct().collect(Collectors.toList())
<b>map</b>	Преобразует каждый элемент стрима	collection.stream().map((s) -> s + ``_1``).collect(Collectors.toList())

2. Терминальные – их результат уже не является потоком, и эти операции завершают работу с потоком:

<b>findAny</b>	Возвращает любой подходящий элемент из стрима (возвращает Optional)	collection.stream().findAny().orElse(«1»)
<b>collect</b>	Представление результатов в виде коллекций и других структур данных	collection.stream().filter((s) -> s.contains(«1»)).collect(Collectors.toList())
<b>count</b>	Возвращает количество элементов в стриме	collection.stream().filter(«a1»::equals).count()

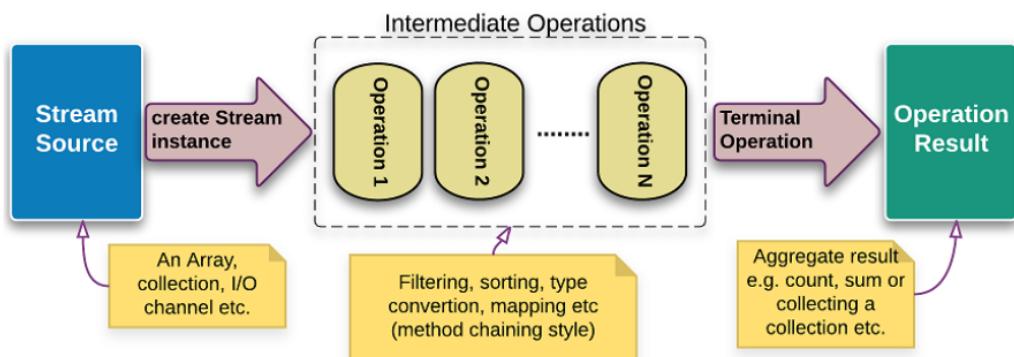
### 3.1 Примеры использования filter, findFirst, findAny, skip, limit и count

**Условие:** дана коллекция строк Arrays.asList(«a1», «a2», «a3», «a1»), давайте посмотрим как её можно обрабатывать используя методы filter, findFirst, findAny, skip и count:

Задача	Код примера	Результат
Вернуть количество вхождений объекта «a1»	collection.stream().filter(«a1»::equals).count()	2
Вернуть первый элемент коллекции или 0, если коллекция пуста	collection.stream().findFirst().orElse(«0»)	a1
Вернуть последний элемент коллекции или «empty», если коллекция пуста	collection.stream().skip(collection.size() — 1).findAny().orElse(«empty»)	a1
Найти элемент в коллекции равный «a3» или кинуть ошибку	collection.stream().filter(«a3»::equals).findFirst().get()	a3
Вернуть третий элемент коллекции по порядку	collection.stream().skip(2).findFirst().get()	a3
Вернуть два элемента начиная со второго	collection.stream().skip(1).limit(2).toArray()	[a2, a3]
Выбрать все элементы по шаблону	collection.stream().filter((s) -> s.contains(«1»)).collect(Collectors.toList())	[a1, a1]

Задача	Код примера	Результат
Выбрать мужчин-военнообязанных (от 18 до 27 лет)	peoples.stream().filter((p) -> p.getAge() >= 18 && p.getAge() < 27 && p.getSex() == Sex.MAN).collect(Collectors.toList())	[{"name='Петя', age=23, sex=MAN}]
Найти средний возраст среди мужчин	peoples.stream().filter((p) -> p.getSex() == Sex.MAN).mapToInt(People::getAge).average().getAsDouble()	36.0
Найти кол-во потенциально работоспособных людей в выборке (т.е. от 18 лет и учитывая что женщины выходят в 55 лет, а мужчина в 60)	peoples.stream().filter((p) -> p.getAge() >= 18).filter((p) -> (p.getSex() == Sex.WOMEN && p.getAge() < 55)    (p.getSex() == Sex.MAN && p.getAge() < 60)).count()	2

## Java Streams



Можно почитать здесь, про методы взаимодействия и т.д.

<https://habr.com/ru/company/luxoft/blog/270383/>

## 15. Статический импорт и его использование для программирования Stream API

Статический импорт выглядит как

```
import static java.lang.Math.sqrt;
```

Оператор import, предваряемый ключевым словом static, можно применять для импорта статических членов класса или интерфейса. Благодаря статическому импорту появляется возможность ссылаться на статические члены непосредственно по именам, не уточняя их именем класса.

То есть после такого импорта sqrt нам больше не нужно писать Math.sqrt(16). Пишем просто sqrt(16).

Статический импорт - синтаксический сахар, т.к. Stream API использует большое количество статических методов, таких как Stream.of(), Stream.generate(), Stream.empty(), Stream.generate(), Stream.concat(), Stream.iterate(), то использование всех этих методов в коде с явным указанием класса может перегружать читаемость. Поэтому можно использовать статический импорт и получать понятный код.

```
import static java.util.stream.Collectors.toSet;
import static java.util.stream.Stream.of;
import static java.util.stream.Stream.concat;
import static java.util.stream.Stream.empty;

public class Test {

    public static void main(String[] args) {
        Set<Integer> set =
            concat(
                concat(
                    of(1, 2, 3, 4),
                    of(5, 6, 7, 8)),
                    empty()
                )
            .collect(toSet());
    }
}

// another way of importing things
// - especially for the lazy ones
import static java.util.stream.Stream.*;
```

## 16. Назначение метода stream() интерфейса Collection.

### Примеры

<b>Способ создания стрима</b>	<b>Шаблон создания</b>	<b>Пример</b>
1. Классический: Создание стрима из коллекции	collection.stream()	<pre>Collection&lt;String&gt; collection = Arrays.asList("a1", "a2", "a3");  Stream&lt;String&gt; streamFromCollection = collection.stream();</pre>

`default Stream<E> stream()`

Returns a sequential `Stream` with this collection as its source.

This method should be overridden when the `spliterator()` method cannot return a spliterator that is `IMMUTABLE`, `CONCURRENT`, or *late-binding*. (See `spliterator()` for details.)

```
public interface Stream<T>
extends BaseStream<T, Stream<T>>
```

A sequence of elements supporting sequential and parallel aggregate operations. The following example illustrates an aggregate operation using `Stream` and `IntStream`:

```
int sum = widgets.stream()
    .filter(w -> w.getColor() == RED)
    .mapToInt(w -> w.getWeight())
    .sum();
```

## 17. Работа со Stream API. Нетерминальные операции потока Stream<T>: назначение и использование

1. Нетерминальные(конвейерные) – их результат тоже является потоком, и к нему в свою очередь можно тоже применить операцию;

## 2.1 Краткое описание конвейерных методов работы со стримами

Метод stream	Описание	Пример
<b>filter</b>	Отфильтровывает записи, возвращает только записи, соответствующие условию	collection.stream().filter(«a1»::equals).count()
<b>skip</b>	Позволяет пропустить N первых элементов	collection.stream().skip(collection.size() — 1).findFirst().orElse(«1»)
<b>distinct</b>	Возвращает стрим без дубликатов (для метода equals)	collection.stream().distinct().collect(Collectors.toList())
<b>map</b>	Преобразует каждый элемент стрима	collection.stream().map(s -> s + "_1").collect(Collectors.toList())

<b>map</b>	Преобразует каждый элемент стрима	collection.stream().map((s) -> s + "_1").collect(Collectors.toList())
<b>peek</b>	Возвращает тот же стрим, но применяет функцию к каждому элементу стрима	collection.stream().map(String::toUpperCase).peek((e) -> System.out.print(", " + e)).collect(Collectors.toList())
<b>limit</b>	Позволяет ограничить выборку определенным количеством первых элементов	collection.stream().limit(2).collect(Collectors.toList())
<b>sorted</b>	Позволяет сортировать значения либо в натуральном порядке, либо задавая Comparator	collection.stream().sorted().collect(Collectors.toList())
<b>mapToInt</b> , <b>mapToDouble</b> , <b>mapToLong</b>	Аналог map, но возвращает числовой стрим (то есть стрим из числовых примитивов)	collection.stream().mapToInt((s) -> Integer.parseInt(s)).toArray()

Задача	Код примера	Результат
Выбрать мужчин-военнообязанных (от 18 до 27 лет)	peoples.stream().filter((p)-> p.getAge() >= 18 && p.getAge() < 27 && p.getSex() == Sex.MAN).collect(Collectors.toList())	[{"name='Петя', age=23, sex=MAN}]
Найти средний возраст среди мужчин	peoples.stream().filter((p) -> p.getSex() == Sex.MAN).mapToInt(People::getAge).average().getAsDouble()	36.0
Найти кол-во потенциально работоспособных людей в выборке (т.е. от 18 лет и учитывая что женщины выходят в 55 лет, а мужчина в 60)	peoples.stream().filter((p) -> p.getAge() >= 18).filter((p) -> (p.getSex() == Sex.WOMEN && p.getAge() < 55)    (p.getSex() == Sex.MAN && p.getAge() < 60)).count()	2

## 18. Работа со Stream API. Терминальные операции, назначение и использование. Примеры

2. Терминальные – их результат уже не является потоком, и эти операции завершают работу с потоком:

<b>findAny</b>	Возвращает любой подходящий элемент из стрима (возвращает Optional)	collection.stream().findAny().orElse(«1»)
<b>collect</b>	Представление результатов в виде коллекций и других структур данных	collection.stream().filter((s) -> s.contains(«1»)).collect(Collectors.toList())
<b>count</b>	Возвращает количество элементов в стриме	collection.stream().filter(«a1»::equals).count()

Задача	Код примера	Результат
Выбрать мужчин-военнообязанных (от 18 до 27 лет)	peoples.stream().filter((p)-> p.getAge() >= 18 && p.getAge() < 27 && p.getSex() == Sex.MAN).collect(Collectors.toList())	[{"name='Петя', age=23, sex=MAN}]
Найти средний возраст среди мужчин	peoples.stream().filter((p) -> p.getSex() == Sex.MAN).mapToInt(People::getAge).average().getAsDouble()	36.0
Найти кол-во потенциально работоспособных людей в выборке (т.е. от 18 лет и учитывая что женщины выходят в 55 лет, а мужчина в 60)	peoples.stream().filter((p) -> p.getAge() >= 18).filter((p) -> (p.getSex() == Sex.WOMEN && p.getAge() < 55)    (p.getSex() == Sex.MAN && p.getAge() < 60)).count()	2

## 19. Интерфейс SplitIterator и его методы

Сплиттератор — это интерфейс, который содержит 8 методов, причём четыре из них уже имеют реализацию по умолчанию. Используется для параллельного программирования, последовательная и параллельная обработка данных. Оставшиеся методы — это `tryAdvance`, `trySplit`, `estimateSize` и `characteristics`.

*SplitIterator*. Это интерфейс, доступный в пакете `java.util`.

Реальный класс `Stream` работает похоже, но использует не `Iterator`, а `SplitIterator`:

```
public interface SplitIterator<T> {
    boolean tryAdvance(Consumer<T> action);
    SplitIterator<T> trySplit();
    long estimateSize();
}
```

Методы `SplitIterator`:

1. tryAdvance: полный аналог методов hasNext+next из Iterator; объединены в один, так как так его проще реализовывать (см. пример)
2. Метод estimateSize: может использоваться для оптимизации; например, в нашем MyStream мы могли бы его использовать для начального размера списка в методе `toList`
3. Метод trySplit: используется для возможности параллельного обхода коллекции

<https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html>

**API Note:**

Spliterators, like Iterators, are for traversing the elements of a source. The Spliterator API was designed to support efficient parallel traversal in addition to sequential traversal, by supporting decomposition as well as single-element iteration. In addition, the protocol for accessing elements via a Spliterator is designed to impose smaller per-element overhead than Iterator, and to avoid the inherent race involved in having separate methods for `hasNext()` and `next()`.

## **20. Понятие многопоточности и написание многопоточных программ на языке Джава**

Смотреть пункт 11

Самый краткий и понятный сайт:  
<https://javarush.ru/groups/posts/2047-threadom-java-ne-isportishjh--chastjh-i---potoki>

## **21. Потоки (threads) в Джава. Создание потоков в Джава программах Подходы к созданию потоков: через наследование и реализацию интерфейса Runnable Thread API. Запуск потока с Runnable**

<https://javarush.ru/groups/posts/2047-threadom-java-ne-isportishjh--chastjh-i---potoki> - тут всё есть

## Отличие Runnable от Thread?

- Thread thread = new Thread();
- thread.start();

```
public class MyThread extends Thread {  
    public void run(){  
        System.out.println("MyThread running");  
    }  
}
```

```
MyThread myThread = new MyThread();  
myThread.start();
```

```
Thread thread = new Thread(){  
    public void run(){  
        System.out.println("Thread Running"); } }  
thread.start();
```

- public interface Runnable() { public void run(); }

### Класс Java реализует Runnable

```
public class MyRunnable implements Runnable {  
    public void run(){  
        System.out.println("MyRunnable running");  
    } }
```

### Анонимная реализация Runnable

```
Runnable myRunnable = new Runnable(){  
    public void run(){  
        System.out.println("Runnable running");  
    } }
```

## Способ 1

Создать объект класса Thread, передав ему в конструкторе нечто, реализующее интерфейс Runnable. Этот интерфейс содержит метод run(), который будет выполняться в новом потоке. Поток закончит выполнение, когда завершится его метод run().

Выглядит это так:

```
class SomeThing          //Нечто, реализующее интерфейс Runnable
implements Runnable     //(содержащее метод run())
{
    public void run()      //Этот метод будет выполняться в побочном потоке
    {
        System.out.println("Привет из побочного потока!");
    }
}

public class Program       //Класс с методом main()
{
    static SomeThing mThing; //mThing - объект класса, реализующего интерфейс Runnable

    public static void main(String[] args)
    {
        mThing = new SomeThing();

        Thread myThready = new Thread(mThing); //Создание потока "myThready"
        myThready.start();                  //Запуск потока

        System.out.println("Главный поток завершён...");
    }
}
```

## Способ 2

Создать потомка класса Thread и переопределить его метод run():

```
class AffableThread extends Thread
{
    @Override
    public void run() //Этот метод будет выполнен в побочном потоке
    {
        System.out.println("Привет из побочного потока!");
    }
}

public class Program
{
    static AffableThread mSecondThread;

    public static void main(String[] args)
    {
        mSecondThread = new AffableThread(); //Создание потока
        mSecondThread.start(); //Запуск потока

        System.out.println("Главный поток завершён...");
    }
}
```

В приведённом выше примере в методе main() создается и запускается еще один поток. Важно отметить, что после вызова метода mSecondThread.start() главный поток продолжает своё выполнение, не дожидаясь пока порожденный им поток завершится. И те инструкции, которые идут после вызова метода start(), будут выполнены параллельно с инструкциями потока mSecondThread.

## 22. Потоки в Джава. Методы Thread API

### Thread API

```
public class Thread {  
    void start(); // запуск потока  
    static Thread currentThread(); // поток, который вызвал этот метод  
    String getName(); // имя потока (можно задать через setName)  
    void setName(String name);  
    void join() throws InterruptedException;  
    void setDaemon(boolean on);  
    static void sleep(long millis) throws InterruptedException;  
    State getState();  
}
```

Подробности

## 23. Синхронизация потоков. Понятие синхронизации. Блок синхронизации

Синхронизация заключается в согласовании скоростей выполнения процессов или потоков путем приостановки одного потока до наступления некоторого события, а затем активизация этого потока при наступлении этого события. Например, поток - получатель должен обращаться за данными только после того, как данные помещены в буфер потоком – отправителем.

Другой поток можно подождать с помощью sleep, interrupt, join(пока другой не закончит)

Потоки прекрасно работают, если они не используют mutable shared state:

- shared – два или более потока обращаются к одним и тем же данным в памяти

- mutable – данные являются изменяемыми

Т.е. все хорошо, если:

- каждый поток работает со своими изменяемыми данными (данные не являются shared)

- потоки работают с общими неизменяемыми данными (данные не являются mutable)

Блок синхронизации:

```
synchronized (lock) { // lock может быть любым объектом // код блока }
```

1. Блок синхронизации для одного и того же lock может выполнять одновременно только один поток

2. Если поток 2 выполняет блок синхронизации с lock после того, как поток 1 выполнил блок синхронизации с lock, то поток 2 “увидит” все изменения, внесенные потоком 1.

## Синхронизация потоков

Пример:

```
synchronized (lock) {
```

```
    counter++;
```

```
}
```

Поток 1	Поток 2
	counter = 0
counter = counter + 1	(параллельное выполнение невозможно в блоке synchronized)
counter = 0 + 1	
	counter = 1
(параллельное выполнение невозможно в блоке synchronized)	counter = counter + 1
	counter = 1 + 1
	counter = 2

Visibility: Поток 2 гарантированно увидит изменения, внесенные потоком 1

[online.mirea.ru](http://online.mirea.ru)

Смысл прост. Если один поток зашел внутрь блока кода, который помечен словом synchronized, он моментально захватывает мьютекс 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.

## 24. Использование синхронизации потоков для пары процессов producer/reader(производители/потребители)

```
private static int result = 0;  
private static boolean ready = false;
```

[Исходный код](#)

```
public static void main(String[] args) {  
    Object lock = new Object();  
    Runnable producer = () -> {  
        sleep(100);  
        synchronized (lock) {  
            result = 42; ready = true;  
        }  
    };  
    Runnable reader = () -> {  
        while (true) {  
            synchronized (lock) {  
                if (ready) {  
                    System.out.println(result);  
                    break;  
                }  
            }  
        }  
    };  
    new Thread(reader).start(); new Thread(producer).start();  
}
```

Важно, что и у producer, и у reader используется один и тот же объект lock в блоке synchronized. При использовании разных объектов никаких гарантий синхронизации нет.

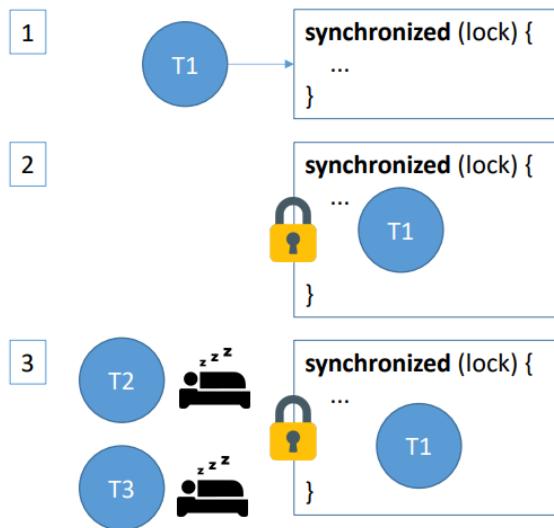
Гарантируется, что значение поля ready внутри блока synchronized будет прочитано то, которое было записано потоком producer

[online.mirea.ru](#)

## 25. Реализация блока synchronized на основе мониторов

Блок synchronized реализован на основе мониторов:

- Монитор может быть в двух состояниях: свободен (released) и захвачен (acquired)
  - При входе в блок synchronized происходит попытка захвата монитора:
    - Если монитор свободен, то он становится захвачен
    - Если монитор уже захвачен другим потоком, то текущий поток останавливается и ждет, пока другой поток не освободит монитор
    - Если монитор захвачен текущим потоком, то он остается захвачен
    - При выходе из блока synchronized монитор освобождается
    - Если при этом другие потоки ждали освобождения этого монитора, то выбирается один из этих потоков, который захватывает монитор и входит в свой блок synchronized.



Монитор принадлежит объекту lock, а не конкретному блоку synchronized. Если два или более блока synchronized используют один и тот же lock, то они используют один монитор для блокировки.

Потоки T2 и T3 находятся в состоянии BLOCKED

```
public class HelloWorld{
    public static void main(String []args){
        Object object = new Object();
        synchronized(object) {
            System.out.println("Hello World");
        }
    }
}
```

## 26. Метод synchronized (lock), особенности его использования для потоков

**Synchronized** (с англ. "синхронизированный") - это ключевое слово, которое позволяет заблокировать доступ к методу или части кода, если его уже использует другой поток.

**Lock** - это монитор. Монитор - это специальный объект, который следит за "состоянием" метода или объекта. Он смотрит, "занят" он или "свободен" в данный момент.

## Синхронизация потоков

Блок синхронизации:

```
synchronized (lock) { // lock может быть любым объектом  
    // код блока  
}
```

1. Блок синхронизации для одного и того же lock может выполнять одновременно только один поток
2. Если поток 2 выполняет блок синхронизации с lock после того, как поток 1 выполнил блок синхронизации с lock, то поток 2 “увидит” все изменения, внесенные потоком 1.

## Синхронизация потоков

Пример:

```
synchronized (lock) {  
    counter++;
```

	Поток 1	Поток 2	
	counter = 0		
	counter = counter + 1	(параллельное выполнение невозможна в блоке synchronized)	
	counter = 0 + 1		
	counter = 1		
	(параллельное выполнение невозможна в блоке synchronized)	counter = counter + 1	
		counter = 1 + 1	
	counter = 2		

**Visibility:** Поток 2 гарантированно увидит изменения, внесенные потоком 1

## Синхронизация потоков

Блок synchronized реализован на основе мониторов:

- Монитор может быть в двух состояниях: свободен (released) и захвачен (acquired)
- При входе в блок synchronized происходит попытка захвата монитора:
  - Если монитор свободен, то он становится захвачен
  - Если монитор уже захвачен другим потоком, то текущий поток останавливается и ждет, пока другой поток не освободит монитор
  - Если монитор захвачен текущим потоком, то он остается захвачен
- При выходе из блока synchronized монитор освобождается
  - Если при этом другие потоки ждали освобождения этого монитора, то выбирается один из этих потоков, который захватывает монитор и входит в свой блок synchronized.

Блок synchronized можно написать перед названием метода, если нужно сделать потокобезопасным весь метод, а не только его часть. Два примера ниже идентичны

```
1  public void boo() {  
2      synchronized (this) {  
3          int i = 5;  
4      }  
5  }  
6  public synchronized void foo() {  
7      int i = 5;  
8  }
```

### Имплементация класса Lock

Добавим новое статическое поле Lock:

```
private static final Lock lock = new ReentrantLock();
```

И изменим код метода increment:

```
lock.lock();  
buf++;  
lock.unlock();
```

При запуске программы она покажет 10000. Блокировка работает.

Использование Lock может понадобиться для более точечной блокировки.

Также можно использовать отдельно ReadLock и WriteLock для того, чтобы не блокировать Thread чтения, если нет записи.

## 27. Синхронизация потоков . Правила happens-before (hb)

Правила happens-before (hb):

1. В рамках одного потока любая операция happens-before любой операцией следующей за ней в исходном коде
2. Выход из synchronized блока happens-before входа в synchronized блок на том же мониторе
3. Запись volatile поля happens-before чтение того же самого volatile поля

4. Завершение метода run экземпляра класса Thread happens-before выхода из метода join()

5. Вызов метода start() экземпляра класса Thread happens-before начало метода run() экземпляра того же треда

Связь happens-before транзитивна, т.е. если X happens-before Y, а Y happens-before Z, то X happens-before Z. Если X happens-before Y, то все изменения, внесенные до операции X, будут видны в коде, следующем за операцией Y. Семантика многопоточной работы в Java (Java Memory Model) описывается именно в терминах happens-before.

Упрощенно можно считать, что при входе в synchronized все переменныечитываются из памяти, а при выходе – записываются в память. На нижнем уровне visibility гарантируется:

- генерацией компилятором кода, записывающего в память содержимого регистров
- инструкцией процессора вида “барьер памяти” (в Intel – MFENCE), которая гарантирует порядок записи данных процессором в память

Но даже при отсутствии отношения happens-before Java гарантирует, что при чтении поля мы всегда прочитаем значение, записанное одним из потоков, а не мусор. При этом не гарантируется “свежесть” значения или порядок, в котором мы видим изменения значения. Если сначала поток 1 установил значение поля x=1, а потом поток 2 установил значение x=2, то поток, который не использует синхронизацию для доступа к полю x, может увидеть любую последовательность значений: только 1; только 2; сначала 1, потом 2; сначала 2, потом 1.

```
private static int result = 0;
private static volatile boolean ready = false;
```

[Исходный код](#)

```
public static void main(String[] args) {
    Runnable producer = () -> {
        sleep(100);
        result = 42;
        ready = true;
    };
    Runnable reader = () -> {
        while (!ready);
        System.out.println(result);
    };
    new Thread(reader).start();
    new Thread(producer).start();
}
```

- “result=42” hb “ready=true” (п.1)
- “ready=true” hb “while (!ready)” (п.3; гарантирует, что условие цикла видит изменение поля)
- “while (!ready)” hb “println(result)” (п.1)

Следовательно,  
“result=42” hb “println(result)” (из транзитивности; гарантирует, что в reader мы увидим присвоенное в producer значение result)

## Синхронизация потоков

Пример отсутствия happens-before:

Пример 1

```
// Поток 1:
int result = 0;
volatile boolean ready = false;

result = 42;
ready = true;
```

```
// Поток 2:
while (true) {
    // Не гарантируется visibility
    // для result, так как нет
    // чтения volatile ready
    System.out.println(result);
}
```

Пример 2

```
// Поток 1:
int result = 0;
boolean ready = false; // не volatile
// Компилятор вправе переставить:
result = 42;
ready = true;
```

```
// Поток 2:
while (!ready);
// Возможные значения:
// зацикливание, 42, 0
System.out.println(result);
```

[Следующий ...](#)

Другой источник

## Правила «happens-before»

Последнее, чего мы коснемся сегодня, это принципы «happens before».

Как ты уже знаешь, в Java основную часть работы по выделению времени и ресурсов потокам для выполнения их задач выполняет планировщик потоков.

Также ты не раз видел, как потоки выполняются в произвольном порядке, и чаще всего предсказать его невозможно.

Да и вообще, после «последовательного» программирования, которым мы занимались до этого, многопоточность выглядит randomной штукой. Как ты уже убедился, ход работы многопоточной программы можно контролировать при помощи целого набора методов.

Но в дополнение к этому в многопоточности Java существует еще один «островок стабильности» — 4 правила под названием «happens-before».

Дословно с английского это переводится как «происходит перед», или «происходит раньше, чем». Понять смысл этих правил достаточно просто.

Представь, что у нас есть два потока — А и В. Каждый из этих потоков может выполнять операции 1 и 2.

И когда в каждом из правил мы говорим «*A happens-before B*», это означает, что все изменения, выполненные потоком А до момента операции 1 и изменения, которые повлекла эта операция, видны потоку В в момент выполнения операции 2 и после выполнения этой операции.

Каждое из этих правил дает гарантию, что при написании многопоточной программы одни события в 100% случаев будут происходить раньше, чем другие, и что поток В в момент выполнения операции 2 всегда будет в курсе изменений, которые поток А сделал во время операции 1.

Давай рассмотрим их.

## **Правило 1.**

Освобождение мьютекса *happens before* происходит раньше захвата этого же монитора другим потоком.

Ну, тут вроде все понятно. Если мьютекс объекта или класса захвачен одним потоком, например, потоком А, другой поток (поток В) не может в это же время его захватить. Нужно подождать, пока мьютекс не освободится.

## **Правило 2.**

Метод Thread.start() *happens before* Thread.run().

Тоже ничего сложного. Ты уже знаешь: чтобы начал выполняться код внутри метода run(), необходимо вызвать у потока метод start(). Именно его, а не сам метод run()!

Это правило гарантирует, что установленные до запуска Thread.start() значения всех переменных будут видны внутри начавшего выполнение метода run().

## **Правило 3.**

Завершение метода run() *happens before* выход из метода join().

Вернемся к нашим двум потокам — А и В.

Мы вызываем метод join() таким образом, чтобы поток В обязательно дождался завершения А, прежде чем выполнять свою работу.

Это означает, что метод run() объекта А обязательно отработает до самого конца. И все изменения в данных, которые произойдут в методе run() потока А стопроцентно будут видны в потоке В, когда он дождется завершения А и начнет работу сам.

## **Правило 4.**

Запись в volatile переменную *happens-before* чтение из той же переменной.

При использовании ключевого слова volatile мы, фактически, всегда будем получать актуальное значение. Даже в случае с long и double, о проблемах с которыми говорилось ранее.

Как ты уже понял, изменения, сделанные в одних потоках, далеко не всегда видны другим потокам. Но, конечно, очень часто встречаются ситуации, когда подобное поведение программы нас не устраивает.

Допустим, в потоке А мы присвоили значение переменной:

```
int z;
```

```
....
```

```
z= 555;
```

Если наш поток В должен вывести значение переменной z на консоль, он запросто может вывести 0, потому что не знает о присвоенном ей значении.

Так вот, Правило 4 гарантирует нам: если объявить переменную z как volatile, изменения ее значений в одном потоке всегда будут видны в другом потоке.

Если мы добавим в предыдущий код слово volatile...

```
volatile int z;
```

```
....
```

```
z= 555;
```

...ситуация, при которой поток В выведет в консоль 0, исключена. Запись в volatile-переменные происходит раньше, чем чтение из них.

## **28. Синхронизация потоков. Модификатор полей volatile и его использование**

Модификатор полей volatile: private static volatile boolean ready = false; Правила happens-before гарантируют, что при чтении volatile поля мы читаем последнее записанное значение. Упрощенно можно представить, что любое обращение к volatile полю завернуто в блок synchronized. Но это не гарантирует атомарности операций! “counter++” все равно подвержено

проблеме параллельности даже при наличии volatile. Но для примера producer/reader можно использовать volatile:

```
private static int result = 0;  
private static volatile boolean ready = false;
```

[Исходный код](#)

```
public static void main(String[] args) {  
    Runnable producer = () -> {  
        sleep(100);  
        result = 42;  
        ready = true;  
    };  
    Runnable reader = () -> {  
        while (!ready);  
        System.out.println(result);  
    };  
    new Thread(reader).start();  
    new Thread(producer).start();  
}
```

- “result=42” hb “ready=true” (п.1)
  - “ready=true” hb “while (!ready)” (п.3; гарантирует, что условие цикла видит изменение поля)
  - “while (!ready)” hb “println(result)” (п.1)
- Следовательно, “result=42” hb “println(result)” (из транзитивности; гарантирует, что в reader мы увидим присвоенное в producer значение result)

## Синхронизация потоков

Пример отсутствия happens-before:

Пример 1

```
// Поток 1:  
int result = 0;  
volatile boolean ready = false;  
  
result = 42;  
ready = true;
```

```
// Поток 2:  
while (true) {  
    // Не гарантируется visibility  
    // для result, так как нет  
    // чтения volatile ready  
    System.out.println(result);  
}
```

Пример 2

```
// Поток 1:  
int result = 0;  
boolean ready = false; // не volatile  
// Компилятор вправе переставить:  
result = 42;  
ready = true;
```

```
// Поток 2:  
while (!ready);  
// Возможные значения:  
// зацикливание, 42, 0  
System.out.println(result);
```

Другой источник Модификатор volatile используют, когда нужно:

- обеспечить видимость данных – убедиться, что при обращении к переменной любой поток получит её последнее записанное значение;

- исключить кэширование значений переменной и хранить их только в основной памяти.

Как только один поток записал что-то в volatile-переменную, значение идёт прямо в общую память и тут же доступно остальным потокам:

```
class CarSharingBase
{
    static volatile int your_car_ID = 3222233;
}
```

Но учтите, что модификатор volatile никак не ограничивает одновременный доступ к данным. А значит, в работу одного потока с полем может вмешаться другой поток. Вот что будет, если два потока одновременно получат доступ к операции увеличения на единицу (`i++`):

```
int i = 0;
```

Поток 1: читает переменную (0)

Поток 1: прибавляет единицу

Поток 2: читает переменную (0)

Поток 1: записывает значение (1)

Поток 2: прибавляет единицу

Поток 2: записывает значение (1)

Если бы два потока не мешали друг другу, а работали последовательно, мы получили бы на выходе значение «2», но вместо этого видим единицу. Чтобы такого не происходило, нужно обеспечить атомарность операции. Атомарными называют операции, которые могут быть выполнены только полностью. Если они не выполняются полностью, они не выполняются вообще, но прервать их невозможно.

В примере с увеличением на единицу мы видим сразу три действия: чтение, сложение, запись. Чтение и запись — операции атомарные, но между ними могут вклиниваться действия другого потока. Поэтому составная операция инкремента (`i++`) полностью атомарной не является.

Обратите внимание: с volatile-переменной возможны как атомарные, так и неатомарные операции. Ключевое слово volatile позволяет сделать так, чтобы все потоки читали одно и то же из основной памяти, но не более того.

Простейший способ гарантировать атомарность — выстроить потоки в очередь за ресурсами с помощью механизма synchronized. Представьте, что на электронный счёт одновременно переводят деньги два клиента. Уж лучше попросить одного из них немного подождать, чем допустить ошибки в денежных расчетах.

## 29. Синхронизация потоков. Атомарные операции. Механизм Wait/notify

<https://metanit.com/java/tutorial/8.5.php> - если лень читать ниже

Атомарная операция — это операция, которая выполняется полностью или не выполняется совсем, частичное выполнение невозможно.

Атомики - это классы, которые выполняют операции изменения своего значения атомарно, т.о. они поддерживают lock-free thread-safe использование переменных. Достигается это с помощью алгоритма compare-and-swap (CAS) и работает быстрее, чем аналогичные реализации с блокировками.

Как можно определить атомарность?

Атомарность операции чаще всего принято обозначать через ее признак неделимости: операция может либо применяться полностью, либо не применяться вообще. Хорошим примером будет запись значений в массив:

```
public class Curiouslyity {  
  
    public volatile int[] array;  
  
    public void nonAtomic() { array = new int[1]; array[0] = 1; }  
  
    public void probablyAtomic() { array = new int[] { 1 }; } }
```

При использовании метода nonAtomic существует вероятность того, что какой-то поток обратится к array[0] в тот момент, когда array[0] не проинициализирован, и получит неожиданное значение. При использовании probablyAtomic (при том условии, что массив сначала заполняется, а уже потом присваивается - **я сейчас не могу гарантировать, что в java это именно так, но представим, что это правило действует в рамках примера**) такого быть не должно: array всегда содержит либо null, либо проинициализированный массив, но в array[0] не может содержаться что-то,

кроме 1. Эта операция неделима, и она не может применяться наполовину, как это было с nonAtomic - только либо полностью, либо никак, и весь остальной код может спокойно ожидать, что в array будет либо null, либо значения, не прибегая к дополнительным проверкам.

Кроме того, под атомарностью операции зачастую подразумевают видимость ее результата всем участникам системы, к которой это относится (в данном случае - потокам); это логично, но, на мой взгляд, не является обязательным признаком атомарности.

### Почему это важно?

Атомарность зачастую проистекает из бизнес-требований приложений: банковские транзакции должны применяться целиком, билеты на концерты заказываться сразу в том количестве, в котором были указаны, и т.д. Конкретно в том контексте, который разбирается (многопоточность в java), задачи более примитивны, но произрастают из тех же требований: например, если пишется веб-приложение, то разбирающий HTTP-запросы сервер должен иметь очередь входящих запросов с атомарным добавлением, иначе есть риск потери входящих запросов, а, следовательно, и деградация качества сервиса. Атомарные операции предоставляют *гарантии* (неделимости), и к ним нужно прибегать, когда эти гарантии необходимы.

Кроме того, атомарные операции *линеаризуемы* - грубо говоря, их выполнение можно разложить в одну линейную историю, в то время как просто операции могут производить граф историй, что в ряде случаев неприемлемо.

Для некоторых случаев удобно использовать классы AtomicInteger  
counter = new AtomicInteger(0); // И его аналоги AtomicLong, AtomicBoolean  
Это аналоги volatile полей, но кроме того они добавляют методы, которые выполняются атомарно:

```
int newValue = counter.addAndGet(delta); // Аналог "counter += delta"  
int oldValue = counter.getAndAdd(delta); // Аналог "counter += delta"  
int newValue = counter.incrementAndGet(); // Аналог "++counter"  
int oldValue = counter.getAndIncrement(); // Аналог "counter++"
```

В частности, вместо “volatile boolean ready” можно использовать “AtomicBoolean ready”

Иногда при взаимодействии потоков встает вопрос о извещении одних потоков о действиях других. Например, действия одного потока зависят от результата действий другого потока, и надо как-то известить один поток, что второй поток произвел некую работу. И для подобных ситуаций у класса **Object** определено ряд методов:

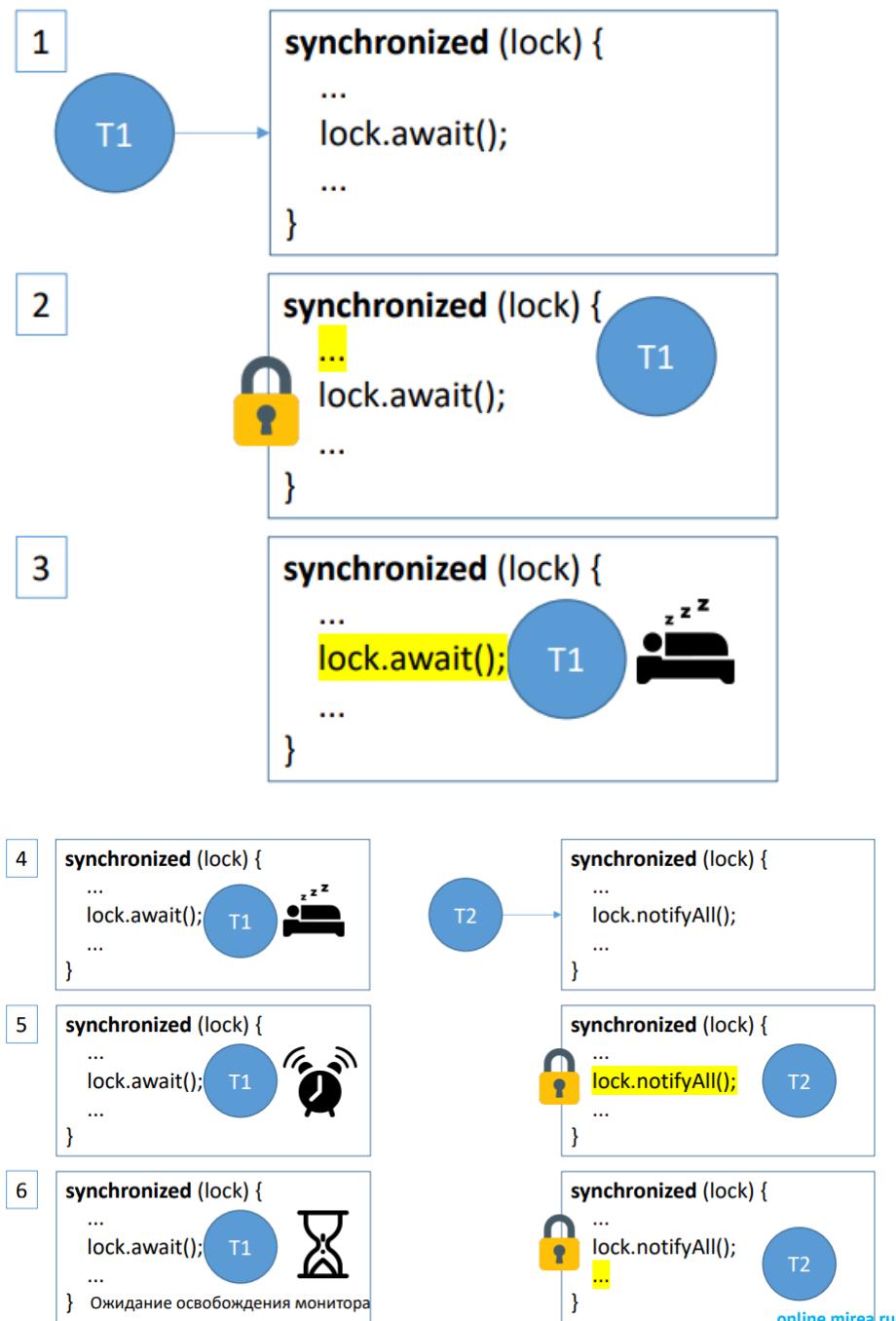
- **wait()**: освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод **notify()**
- **notify()**: продолжает работу потока, у которого ранее был вызван метод **wait()**
- **notifyAll()**: возобновляет работу всех потоков, у которых ранее был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

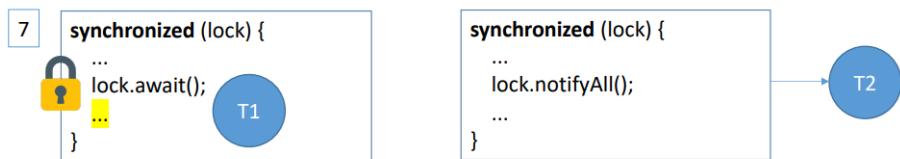
Все эти методы вызываются только из синхронизированного контекста - синхронизированного блока или метода.

```
// Класс Магазин, хранящий произведенные товары
class Store{
    private int product=0;
    public synchronized void get() {
        while (product<1) {
            try {
                wait();
            }
            catch (InterruptedException e) {
            }
        }
        product--;
        System.out.println("Покупатель купил 1 товар");
        System.out.println("Товаров на складе: " + product);
        notify();
    }
}
```

Метод **lock.wait()** освобождает монитор **lock** и переводит текущий поток в режим ожидания монитора (состояние WAITING). Так как монитор освобожден, другие потоки могут выполнять блоки “**synchronized (lock)**”, пока этот поток находится в режиме ожидания (“спит”).

Метод `lock.notifyAll()` будет все ожидающие этот монитор потоки. Все эти потоки начинают пытаться захватить монитор (это получится не сразу после вызова `notifyAll`, а только когда вызвавший `notifyAll` поток выйдет из блока `synchronized`). После этого все спавшие потоки по очереди захватят монитор и выполнят блок `synchronized`.





Метод `lock.notify()`, в отличие от `lock.notifyAll()`, будит только один поток. В нашем случае (и в подавляющем большинстве случаев) нужно использовать `notifyAll`, так как результата producer могут ждать несколько потоков reader, и их все нужно уведомить о доступности результата.

---

```

public static void main(String []args) throws InterruptedException {
    Object lock = new Object();
    // task будет ждать, пока его не оповестят через lock
    Runnable task = () -> {
        synchronized(lock) {
            try {
                lock.wait();
            } catch(InterruptedException e) {
                System.out.println("interrupted");
            }
        }
        // После оповещения нас мы будем ждать, пока сможем взять лок
        System.out.println("thread");
    };
    Thread taskThread = new Thread(task);
    taskThread.start();
    // Ждём и после этого забираем себе лок, оповещаем и отдаём лок
    Thread.currentThread().sleep(3000);
    System.out.println("main");
    synchronized(lock) {
        lock.notify();
    }
}

import java.lang.Thread;
import java.util.concurrent.atomic.AtomicIntegerArray;

/*
 * Gonna produce the following output every time:
 * a 0
 * b 0123
 * c 0123456
 * a 0
 * b 0123
 * c 0123456
 * a 0
 * b 0123
 * c 0123456
 */
public class A {
    private int d = 0;
    private AtomicIntegerArray e = new AtomicIntegerArray(3);

    static <a> void pr(final a b) { System.out.print(b); }
}

```

```

        static void fr(final int a) { for (
            int b = 0, c = a + a * 2;
            b <= c;
            pr(b++)
        ) ; }

        void wt(final int a) { try {
            while (d != a)
                wait();
        } catch(final Exception $) {} }

        void cl(final int i) {
            var $ = 0;
            while (($ = e.get(i)) < 3) {
                switch (i) {
                    case 0 -> d(0, 'a', 1);
                    case 1 -> d(1, 'b', 2);
                    case 2 -> d(2, 'c', 0);
                }
                e.set(i, $ + 1);
            }
        }

        public static void main(final String... $$) {
            final var $ = new A();

            final var a = new Thread(() -> $.cl(0));
            final var b = new Thread(() -> $.cl(1));
            final var c = new Thread(() -> $.cl(2));

            a.start();
            b.start();
            c.start();
        }

        synchronized void d(final int i, final char j, final int k) {
            wt(i);
            pr("\n" + j + ' ');
            fr(i);
            d = k;
            notifyAll(); // won't work with 'notify()'
        }
    }

import java.lang.Thread;

/*
 * Making atomics from non-atomics
 */
public class A {
    volatile String a = "";

    static <a> void $(a b) { System.out.println(b); }

    // The output (#) will be: one or several random letters.
    // Reads the value of the variable, appends new string to it
    // and then assigns the new value.
    void nonAtomic(String b) { a += b; }

    // The output (#) will be: random sequence of 4 letters every time.
    // Thanks to synchronized keyword this method
    // executes as an atomic operation.
    // But there's no synchronization between the threads so the
    // sequence will be random.
    synchronized void atomic(String b) { a += b; }

    public static void main(String ...$) throws Exception {
        var e = new A();
        Thread a, b, c, d;
    }
}

```

```

if (false) {
    a = new Thread(() -> e.nonAtomic("a"));
    b = new Thread(() -> e.nonAtomic("b"));
    c = new Thread(() -> e.nonAtomic("c"));
    d = new Thread(() -> e.nonAtomic("d"));
} else {
    a = new Thread(() -> e.atomic("a"));
    b = new Thread(() -> e.atomic("b"));
    c = new Thread(() -> e.atomic("c"));
    d = new Thread(() -> e.atomic("d"));
}

a.start(); b.start(); c.start(); d.start();
a.join(); b.join(); c.join(); d.join();

$(e.a); $(e.a.length()); // (#)

e.a = "";
e.nonAtomic("a");
e.nonAtomic("b");
e.nonAtomic("c");
e.nonAtomic("d");
$(e.a); $(e.a.length()); // abcd \n 4
}
}

```

## 30. Синхронизация потоков с использованием классов пакета `java.util.concurrent.locks`

### Синхронизация потоков

В пакете `java.util.concurrent.locks` есть классы для расширенной поддержки синхронизации.

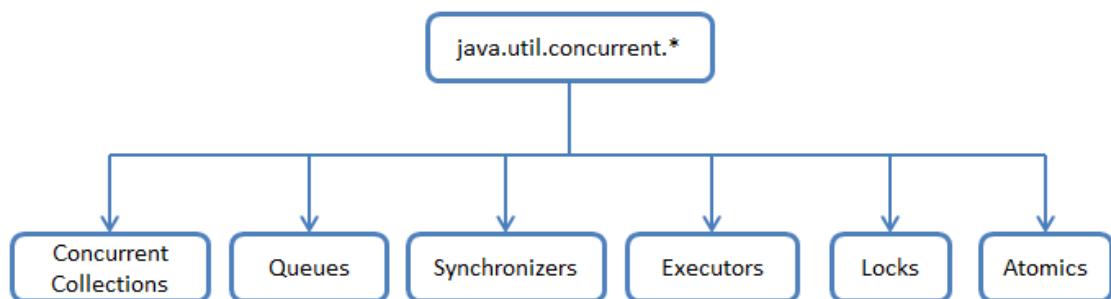
Класс `ReentrantLock` является аналогом блока `synchronized` с некоторыми дополнительными функциями. В новых версиях Java рекомендуется использовать его вместо `synchronized`.

Класс `ReentrantReadWriteLock` позволяет блокировать потоки только в случае, когда идет изменение данных, а в случае чтения данных блокировки потоков не происходит.

## Пример использования ReentrantLock

```
14     public static void main(String[] args) {
15         ReentrantLock lock = new ReentrantLock();
16         Condition isReady = lock.newCondition();
17         Runnable producer = () -> {
18             try {
19                 Thread.sleep(100);
20                 lock.lock();
21                 try {
22                     result = 42;
23                     ready = true;
24                     isReady.signalAll();
25                 } finally {
26                     lock.unlock();
27                 }
28             } catch (InterruptedException ex) {
29                 // завершение потока
30             }
31         };
32         Runnable reader = () -> {
33             try {
34                 lock.lock();
35                 try {
36                     while (!ready) {
37                         isReady.await();
38                     }
39                     System.out.println(result);
40                 } finally {
41                     lock.unlock();
42                 }
43             } catch (InterruptedException ex) {
44                 // завершение потока
45             }
46         };
47         new Thread(reader).start();
48         new Thread(producer).start();
49     }
50 }
```

Наверное у многих возникало чувство некоторого хаоса при беглом взгляде на `java.util.concurrent`. В одном пакете намешаны разные классы с совершенно разным функционалом, что несколько затрудняет понимание что к чему относится и как это работает. Поэтому, можно схематично поделить классы и интерфейсы по функциональному признаку, а затем пробежаться по реализации конкретных частей.



Concurrent Collections — набор коллекций, более эффективно работающие в многопоточной среде нежели стандартные универсальные коллекции из java.util пакета. Вместо базового враппера Collections.synchronizedList с блокированием доступа ко всей коллекции используются блокировки по сегментам данных или же оптимизируется работа для параллельного чтения данных по [wait-free](#) алгоритмам.

Queues — неблокирующие и блокирующие очереди с поддержкой многопоточности. Неблокирующие очереди заточены на скорость и работу без блокирования потоков. Блокирующие очереди используются, когда нужно «притормозить» потоки «Producer» или «Consumer», если не выполнены какие-либо условия, например, очередь пуста или перепонена, или же нет свободного «Consumer»'а.

Synchronizers — вспомогательные утилиты для синхронизации потоков. Представляют собой мощное оружие в «параллельных» вычислениях.

Executors — содержит в себе отличные фрейморки для создания пулов потоков, планирования работы асинхронных задач с получением результатов.

Locks — представляет собой альтернативные и более гибкие механизмы синхронизации потоков по сравнению с базовыми synchronized, wait, notify, notifyAll.

Atomics — классы с поддержкой атомарных операций над примитивами и ссылками.

Лучше глянуть предыдущий номер, там можно взять некоторые ответы

## **31. Запуск и прерывание потоков. Приостановка и прерывание выполнения нити**

Во-первых, должны соблюдаться правила happens-before. Смотреть 27 вопрос.

<https://metanit.com/java/tutorial/8.4.php> - ссылка на источник

Распространенный способ завершения потока представляет опрос логической переменной. И если она равна, например, false, то поток завершает бесконечный

цикл и заканчивает свое выполнение. Пишем метод, далее, когда необходимо сбрасываем флаг, вызываем исключение - поток завершён

Или через метод `interrupt()`, используя исключение

## 32. Обработка операции прерывания потока

<https://urvanov.ru/2016/05/27/java-8-многопоточность/> - ссылочка неплохая

Прерывание (`interrupt`) — это сигнал для потока, что он должен прекратить делать то, что он делает сейчас, и делать что-то другое. Что должен делать поток в ответ на прерывание, решает программист, но обычно поток завершается.

Поток отправляет прерывание вызывая метод `public void interrupt()` класса `Thread`. Для того чтобы механизм прерывания работал корректно, прерываемый поток должен поддерживать возможность прерывания своей работы.

Как поток должен поддерживать прерывание своей работы? Это зависит от того, что он сейчас делает. Если поток часто вызывает методы, которые могут бросить `InterruptedException`, то он просто вызывает `return` при перехвате подобного исключения. Пример:

Java

```
1  for (int i = 0; i < importantInfo.length; i++) {  
2      // Пауза 4 секунды  
3      try {  
4          Thread.sleep(4000);  
5      } catch (InterruptedException e) {  
6          // Ожидание было прервано! Больше не нужно сообщений.  
7          return;  
8      }  
9      // Пишем сообщение  
10     System.out.println(importantInfo[i]);  
11 }
```

Многие методы, которые бросают `InterruptedException`, например методы `sleep`, останавливают своё выполнение и возвращают управление в вызвавший их код при получении прерывания (`interrupt`).

Что если поток выполняется длительное время без вызова методов, которые бросают исключение `InterruptedException`? Тогда он может периодически вызывать метод `Thread.interrupted()`, который возвращает `true`, если получен сигнал о прерывании. Например:

## Java

```
1  for (int i = 0; i < inputs.length; i++) {  
2      heavyCrunch(inputs[i]);  
3      if (Thread.interrupted()) {  
4          // Мы были прерваны: no more crunching.  
5          return;  
6      }  
7  }
```

В этом примере код просто проверяет на наличие сигнала о прерывании, и выходит из потока, если сигнал есть. В более сложных приложениях имеет смысл бросить исключение `InterruptedException`:

### Java

```
1 if (Thread.interrupted()) {  
2     throw new InterruptedException();  
3 }
```

Это позволяет располагать код обработки прерывания потока в одной клаузе `catch`.

Механизм прерывания реализован с помощью внутреннего флага, известного как статус прерывания (*interrupt status*). Вызов `Thread.interrupt()` устанавливает этот флаг. Когда поток проверяет наличие прерывания вызовов `Thread.interrupted()`, то флаг статуса прерывания сбрасывается. Нестатический метод `isInterrupted()`, который используется одним потоком для проверки статуса прерывания другого потока, не меняет флаг статуса прерывания.

По соглашению любой метод, который прерывает свою выполнение бросая исключение `InterruptedException`, очищает флаг статуса прерывания, когда он бросает это исключение. Однако есть вероятность, что флаг статуса прерывания будет сразу же установлен ещё раз, если другой поток вызовет `interrupt()`.

### 33. Ожидание и присоединение запущенной нити основным потоком управления

Как заставить поток ждать другие потоки (присоединяться)? t1.join();

```
public class ThreadJoinExample implements Runnable {  
    public void run() {  
        for (int i = 1; i <= 10; i++) {  
            System.out.println("This is message #" + i);  
            try {  
                Thread.sleep(2000);  
            } catch (InterruptedException ex) {  
                System.out.println("I'm about to stop");  
                return;  
            }  
        }  
    }  
}  
  
public static void main(String[] args) {  
    Thread t1 = new Thread(new ThreadJoinExample());  
    t1.start();  
    try {  
        t1.join();  
    } catch (InterruptedException ex) {  
        // do nothing  
    }  
    System.out.println("I'm " + Thread.currentThread().getName());  
}
```

#### Соединение

Метод `join` позволяет одному потоку ждать завершения другого потока.

Если `t` является экземпляром класса `Thread`, чей поток в данный момент продолжает выполняться, то

```
t.join();
```

приведёт к приостановке выполнения текущего потока до тех пор, пока поток `t` не завершит свою работу. Метод `join()` имеет варианты с параметрами:

#### Java

```
1  public final void join(long millis)  
2          throws InterruptedException
```

## Java

```
1 public final void join(long millis,  
2                         int nanos)  
3                     throws InterruptedException
```

Они позволяют задать время в миллисекундах и дополнительно количество наносекунд, в течение которых ждать завершения выполнения потока. Однако, как и с методами `sleep`, методы `join` зависят от возможностей операционной системы, поэтому вы не должны полагаться на то, что `join` будет ждать точно указанное время.

Как и методы `sleep`, методы `join` отвечают на сигнал прерывания, останавливая процесс ожидания и бросая исключение `InterruptedException`.

## Простой пример

Пример состоит из двух потоков. Первый поток является главным потоком приложения, который имеет каждая программа на Java. Главный поток создаёт новый поток и ждёт его завершения. Если второй поток выполняется слишком долго, то главный поток прерывает его.

## Java

```
1 public class SimpleThreads {  
2  
3     // Выводим сообщение  
4     // с именем текущего потока в начале.  
5     static void threadMessage(String message) {  
6         String threadName =  
7             Thread.currentThread().getName();  
8         System.out.format("%s: %s%n",  
9             threadName,  
10            message);  
11    }  
12  
13    private static class MessageLoop  
14        implements Runnable {  
15            public void run() {  
16                String importantInfo[] = {  
17                    "Mares eat oats",  
18                    "Does eat oats",  
19                    "Little lambs eat ivy",  
20                    "A kid will eat ivy too"  
21                };  
22                try {  
23                    for (int i = 0;  
24                        i < importantInfo.length;  
25                        i++) {  
26                        // Ждём 4 секунды  
27                        Thread.sleep(4000);  
28                        // Пишем сообщение  
29                        threadMessage(importantInfo[i]);  
30                    }  
31                } catch (InterruptedException e) {  
32                    e.printStackTrace();  
33                }  
34            }  
35        }  
36    }  
37}
```

```
30         }
31     } catch (InterruptedException e) {
32         threadMessage("I wasn't done!");
33     }
34 }
35 }

36
37 public static void main(String args[])
38 throws InterruptedException {
39
40     // Задержка в миллисекундах
41     // перед тем как мы прерываем MessageLoop
42     // (по умолчанию один час).
43     long patience = 1000 * 60 * 60;
44
45     // Если есть аргумент командной строки,
46     // то он указывает ожидание в секундах.
47     if (args.length > 0) {
48         try {
49             patience = Long.parseLong(args[0]) * 1000;
50         } catch (NumberFormatException e) {
51             System.err.println("Argument must be an integer.");
52             System.exit(1);
53         }
54     }
55
56     threadMessage("Starting MessageLoop thread");
57     long startTime = System.currentTimeMillis();
58     Thread t = new Thread(new MessageLoop());
```

```

59         t.start();

60

61     threadMessage("Waiting for MessageLoop thread to finish");

62     // ждём пока MessageLoop

63     // существует

64     while (t.isAlive()) {

65         threadMessage("Still waiting...");

66         // Ждём максимум 1 секунду

67         // завершения потока MessageLoop

68         t.join(1000);

69         if (((System.currentTimeMillis() - startTime) > patience)

70             && t.isAlive()) {

71             threadMessage("Tired of waiting!");

72             t.interrupt();

73             // должно быть недолго теперь.

74             // -- Ждём до конца

75             t.join();

76         }

77     }

78     threadMessage("Finally!");

79 }

80 }

```

## 34. Жизненный цикл потока на языке Джава. Состояние потока

**Жизненный цикл потока.** При выполнении программы объект Thread может находиться в одном из четырех основных **состояний**: «новый», «рабочеспособный», «нерабочеспособный» и «пассивный». При создании **потока** он получает **состояние «новый»** (NEW) и не выполняется. Для перевода **потока**

из состояния «новый» в «рабочоспособный» (RUNNABLE) следует выполнить метод `start()`, вызывающий метод `run()`.

- **New** — поток находится в состоянии `New`, когда создается экземпляр объекта класса `Thread`, но метод `start` не вызывается.
- **Runnable** — когда для объекта `Thread` был вызван метод `start`. В этом состоянии поток либо ожидает, что планировщик заберет его для выполнения, либо уже запущен. Назовем состояние, когда поток уже выбран для выполнения, «работающим» (`running`).
- **Non-Runnable(Blocked, Timed-Waiting)** — когда поток жив, то есть объект класса `Thread` существует, но не может быть выбран планировщиком для выполнения. Он временно неактивен.
- **Terminated** — когда поток завершает выполнение своего метода `run`, он переходит в состояние `terminated` (завершен). На этом этапе задача потока завершается.

Ниже дано схематическое представление жизненного цикла потока в Java:

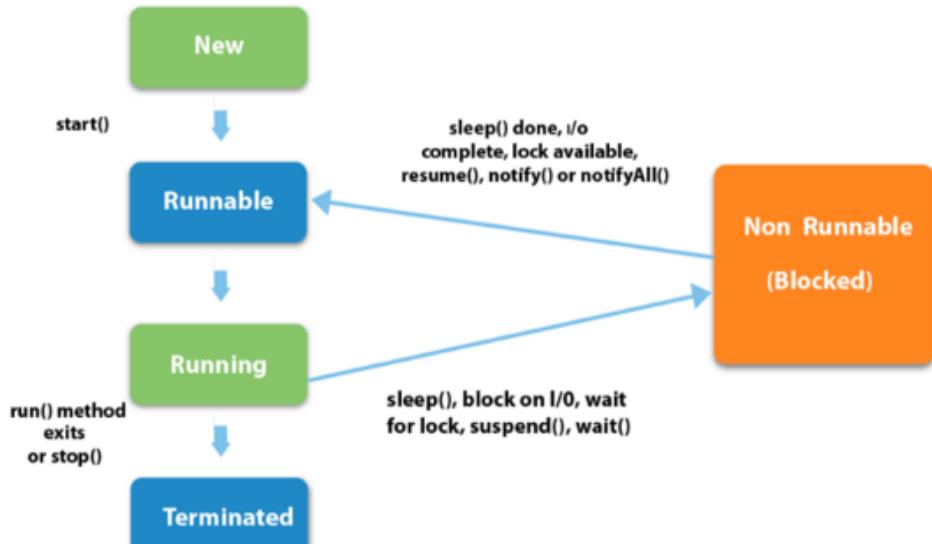


Рис. 1 — Жизненный цикл Java-потока

**Планировщик** — это программное обеспечение, которое используется для отслеживания задач компьютера. Он отвечает за назначение задач ресурсам, которые могут совершать работу. Мы не будем углубляться в логику, которую реализует планировщик. На данный момент достаточно знать, что планировщик имеет контроль над тем, какая задача должна быть назначена какому аппаратному ресурсу и когда, исходя из доступности ресурса и состояния задачи.

## **35. Многопоточные примитивы и их использование**

Может быть имелись в виду AtomicInteger и прочие?

Многопоточные примитивы

- Класс Thread с методами start/join
- Блоки synchronized
- Методы wait/notify/notifyAll
- Поля с модификатором volatile
- Классы java.util.concurrent.atomic.\*
- Классы java.util.concurrent.locks.\*
- Метод interrupt(), InterruptedException и interrupted state

Все это низкоуровневые механизмы работы с потоками.

## **36. Интерфейс Executor в Джава и его использование.**

### **ExecutorService**

Потоки потребляют довольно много ресурсов компьютера:

- Переключение между потоками занимает заметное время (context switch)
  - Каждый поток требует отдельного стека вызовов (размер по умолчанию – 1 Мб)
  - Потоки конкурируют между собой за процессор
  - Запуск потока занимает довольно много времени Операционные системы плохо справляются с ситуациями ~10000 потоков и более.

Одно из решений: пулы потоков (thread pools); позволяют ограничить количество одновременно запущенных потоков, не ограничивая количество задач.

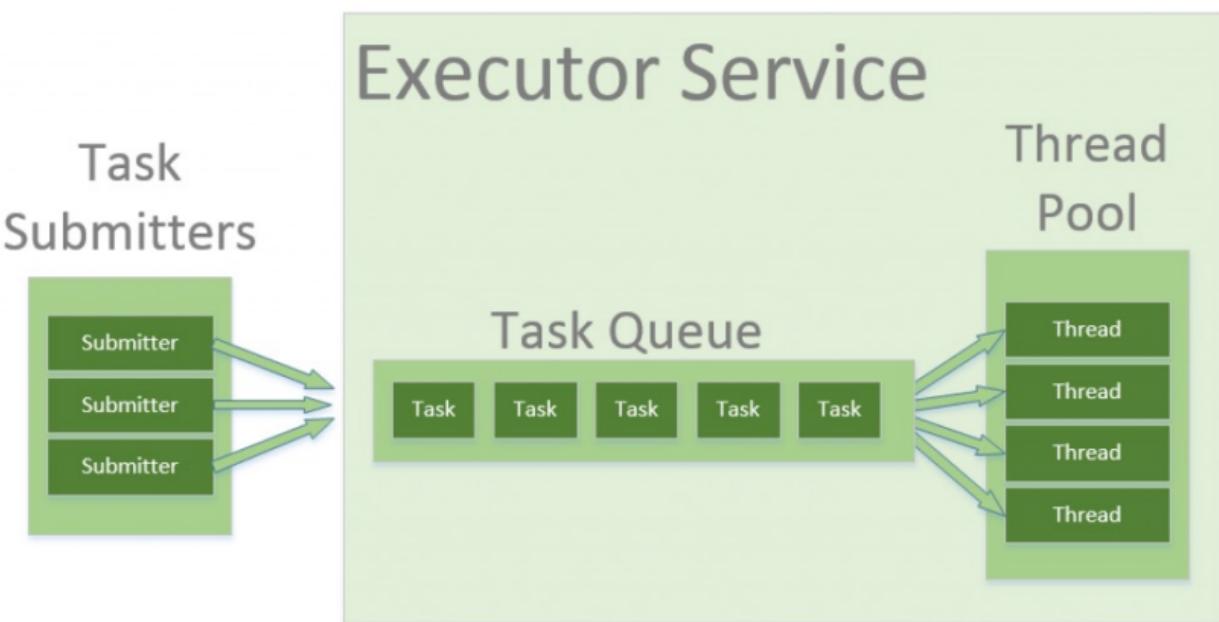
Runnable

```
code = () -> { ... // Код задачи };
```

```
Executor executor = ...; executor.execute(code);
```

Интерфейс Executor – абстракция для запуска задач. Он может запускать потоки по необходимости. Интерфейс `java.util.concurrent.Executor` — это простой интерфейс для поддержки запуска новых задач

Стандартные реализации интерфейса Executor:  
`Executors.newSingleThreadExecutor()`      `Executors.newFixedThreadPool(int nThreads)`      `Executors.newCachedThreadPool()` Все они на самом деле возвращают объект класса `ThreadPoolExecutor` – реализацию пула потоков. Он состоит из набора потоков и очереди задач. “Под капотом” у него то же класс `Thread` с методом `start` и примитивы синхронизации.



Ещё раз вспомним. У нас есть `Executor` для `execute` (т.е. выполнения) некой задачи в потоке, когда реализация создания потока скрыта от нас. У нас есть `ExecutorService` — особый `Executor`, который имеет набор возможностей по управлению ходом выполнения. И у нас есть фабрика `Executors`, которая позволяет создавать `ExecutorService`. Давайте теперь это проделаем сами:

```
public static void main(String[] args) throws  
ExecutionException, InterruptedException {  
    Callable<String> task = () ->  
    Thread.currentThread().getName();  
    ExecutorService service =  
    Executors.newFixedThreadPool(2);  
    for (int i = 0; i < 5; i++) {  
        Future result = service.submit(task);  
        System.out.println(result.get());  
    }  
    service.shutdown();  
}
```

Как мы видим, мы указали фиксированный пул потоков (Fixed Thread Pool) размером 2. После чего мы поочередно отправляем в пул задачи. Каждая задача возвращает строку (String), содержащую имя потока (`currentThread().getName()`). Важно в самом конце выполнить `shutdown` для `ExecutorService`, потому что в противном случае наша программа не завершится.

В фабрике `Executors` есть и другие фабричные методы. Например, мы можем создать пул всего из одного потока — `newSingleThreadExecutor` или пул с кэшированием `newCachedThreadPool`, когда потоки будут убираться из пула, если они простоявают 1 минуту.

На самом деле, за этими `ExecutorService` прячется блокирующая очередь, в которую помещаются задачи и из которой эти задачи выполняются. Подробнее про блокирующие очереди можно посмотреть в видео "[Блокирующая очередь - Collections #5 - Advanced Java](#)". А так же можно прочитать обзор "[Блокирующие очереди пакета](#)

[concurrent](#)" и ответ на вопрос "[When to prefer LinkedBlockingQueue over ArrayBlockingQueue?](#)". Супер упрощённо — BlockingQueue (блокирующая очередь) блокирует поток, в двух случаях:

- ПОТОК ПЫТАЕТСЯ ПОЛУЧИТЬ ЭЛЕМЕНТЫ ИЗ ПУСТОЙ ОЧЕРЕДИ
- ПОТОК ПЫТАЕТСЯ ПОЛОЖИТЬ ЭЛЕМЕНТЫ В ПОЛНУЮ ОЧЕРЕДЬ

```
import java.lang.Thread;
import java.util.ArrayDeque;

/** Task */
@FunctionalInterface interface C { void a(); }

/*
 * Background single thread task executor,
 * illustrates basic principles
 */
class B extends Thread {
    private ArrayDeque<C> aa = new ArrayDeque<>();
    private volatile boolean bb = true;

    @Override public void run() { while (bb) {
        C a = aa.poll();
        if (a == null) continue;
        a.a(); // each task can be launched
                // within its own thread if
                // there's such need
    } }

    void a(C a) { aa.add(a); }

    /*
     * Stops the execution without
     * interrupting running tasks
     */
    void b() throws Exception {
        bb = false;
        super.join();
    }
}

public class Main {

    /**
     * Output will be:
     * a
     * b
     * c
     * every time
     */
    public static void main(String...$) {
        var a = new B();
        a.start();

        a.a() -> System.out.println('a'));
        a.a() -> System.out.println('b'));
        a.a() -> System.out.println('c'));

        try { Thread.sleep(1000); a.b(); } 
        catch (Exception $$) {}
    }
}
```

## **37. Интерфейс Future в Джава. Основное назначение использования его в программах**

Не всегда нам достаточно просто запустить задачу и забыть про нее; часто нужно узнать результат ее выполнения. Для этого используются интерфейсы:

```
public interface ExecutorService extends Executor { Future<T> submit(Runnable task); Future<T> submit(Callable<T> task); ... } public interface Callable<T> { T call() throws Exception; }
```

Future представляет собой будущий результат выполнения задачи:

```
ExecutorService es = Executors.newFixedThreadPool(10);  
Future<String> f1 = es.submit(() -> { Thread.sleep(1000); return "Hello future"; }); //  
Задача начинает выполняться параллельно // и завершится через 1 секунду  
System.out.println(f1.isDone()); // скорее всего false
```

Метод `isDone()` возвращает `true`, если задача успешно выполнена и ее результат можно получить:

```
String result1 = f1.get(); // "Hello future"
```

Если задача еще не выполнена, метод `get` блокирует выполнение текущего потока до тех пор, пока не произойдет одно из:

- задача выполнится успешно и метод `get` вернет ее результат
- задача выполнится с исключением и метод `get` выбросит `ExecutionException` (исходное исключение – `ex.getCause()`)
- задача будет отменена (см. далее) и метод `get` выбросит `CancellationException`
- текущий поток будет прерван и метод `get` выбросит `InterruptedException`

Если задача становится неактуальной, ее можно отменить:

```
boolean cancelled = f1.cancel(true); // или f1.cancel(false)
```

При этом:

1. Если задача уже завершена или уже отменена, то ничего не происходит (cancel возвращает false)
2. Если задача еще не начала выполняться (стоит в очереди), то она просто убирается из очереди
3. Если задача уже начала выполняться (т.е. ей выделен поток из пула), то:

1. Если параметр mayInterruptIfRunning=false, то задача все равно выполняется до конца, отмена игнорируется
2. Если параметр mayInterruptIfRunning=true, то для потока выполняется метод interrupt(). Если задача проверяет состояние прерывания потока, она будет прекращена.

Пример:

```
ExecutorService es = Executors.newFixedThreadPool(2);
Future<Integer> f1 = es.submit(() -> sum(array, 0, array.length / 2));
Future<Integer> f2 = es.submit(() -> sum(array, array.length / 2, array.length));
int totalSum = f1.get() + f2.get();
```

Две задачи суммирования выполняются параллельно, затем мы получаем сумму результатов двух задач.

Futures – повышение уровня абстракции поверх примитивов многопоточной работы. Но нужно помнить, что при обращении разных задач к одному и тому же shared mutable state возникают все те же проблемы многопоточности.

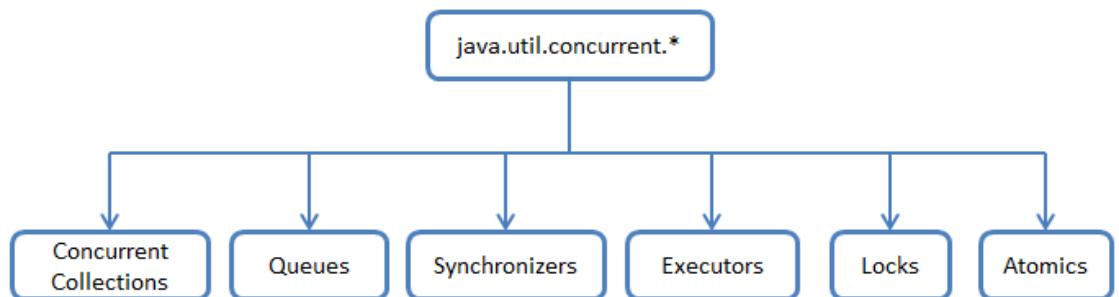
**Future** хранит результат асинхронного вычисления.

Класс-оболочка **FutureTask** представляет собой удобный механизм для превращения Callable одновременно в Future и Runnable, реализуя оба интерфейса.

## 38. Коллекции java.util.concurrent. Состав коллекции.

### Основные методы

Наверное у многих возникало чувство некоторого хаоса при беглом взгляде на `java.util.concurrent`. В одном пакете намешаны разные классы с совершенно разным функционалом, что несколько затрудняет понимание что к чему относится и как это работает. Поэтому, можно схематично поделить классы и интерфейсы по функциональному признаку, а затем пробежаться по реализации конкретных частей.



`Concurrent Collections` — набор коллекций, более эффективно работающих в многопоточной среде нежели стандартные универсальные коллекции из `java.util` пакета. Вместо базового враппера `Collections.synchronizedList` с блокированием доступа ко всей коллекции используются блокировки по сегментам данных или же оптимизируется работа для параллельного чтения данных по `wait-free` алгоритмам.

`Queues` — неблокирующие и блокирующие очереди с поддержкой многопоточности. Неблокирующие очереди заточены на скорость и работу без блокирования потоков. Блокирующие очереди используются, когда нужно «притормозить» потоки «Producer» или «Consumer», если не выполнены какие-либо условия, например, очередь пуста или перепонена, или же нет свободного «Consumer»'а.

`Synchronizers` — вспомогательные утилиты для синхронизации потоков. Представляют собой мощное оружие в «параллельных» вычислениях.

`Executors` — содержит в себе отличные фреймворки для создания пулов потоков, планирования работы асинхронных задач с получением результатов.

Locks — представляет собой альтернативные и более гибкие механизмы синхронизации потоков по сравнению с базовыми synchronized, wait, notify, notifyAll.

Atomics — классы с поддержкой атомарных операций над примитивами и ссылками.

## Concurrent Collections

	Исключение	Возвращает особое значение
Добавление элемента в хвост очереди	<b>add(elem)</b> Выбрасывает IllegalStateException, если в очередь больше нельзя добавить элементы	<b>offer(elem)</b> Возвращает false, если в очередь больше нельзя добавить элементы
Удаление элемента из головы очереди	<b>remove()</b> Выбрасывает NoSuchElementException, если очередь пуста	<b>poll()</b> Возвращает null, если очередь пуста
Проверка головы очереди	<b>element()</b> Выбрасывает NoSuchElementException, если очередь пуста	<b>peek()</b> Возвращает null, если очередь пуста

В очередь нельзя добавить элементы (add/offer неуспешны), если она ограничена по размеру

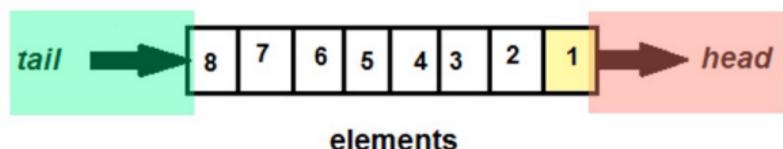
**public interface Queue<E> extends Collection<E> {**

**boolean add(E elem);**  
**boolean offer(E elem);**

**E remove();**  
**E poll();**

**E element();**  
**E peek();**

}



Классы, реализующие интерфейс BlockingQueue: LinkedBlockingQueue: на основе двусвязного списка, размер может быть неограничен ArrayBlockingQueue: на основе массива, размер фиксирован SynchronousQueue: очередь “нулевой длины”; добавление элемента в хвост

блокируется, пока другой поток не заберет этот элемент из головы. Все эти классы являются потокобезопасными, т.к. спроектированы специально для использования разными потоками одновременно.

### Пример producer/reader с использованием очередей:

```
BlockingQueue<Integer> queue = new LinkedBlockingQueue<>(10);
```

```
Runnable producer = () -> {
    try {
        for (int i = 0; i < 1000; i++) {
            queue.put(i);
        }
    } catch (InterruptedException ex) {
        return;
    }
};
```

```
Runnable reader = () -> {
    try {
        while (true) {
            int value = queue.take();
            System.out.println(value);
        }
    } catch (InterruptedException ex) {
        return;
    }
};
```

producer будет клать элементы в очередь; если в очереди накопится 10 элементов, а reader не будет успевать их обрабатывать, метод put будет блокироваться до освобождения места в очереди.

Другие интерфейсы, связанные с очередями:

TransferQueue – блокирующая очередь с подтверждением доставки

BlockingDeque – блокирующая двусторонняя очередь

Другие классы, связанные с очередями:

ConcurrentLinkedQueue – неблокирующая очередь

ConcurrentLinkedDeque – неблокирующая двусторонняя очередь

DelayQueue – блокирующая очередь с задержкой

LinkedBlockingDeque – блокирующая двусторонняя очередь

LinkedTransferQueue – блокирующая очередь с подтверждением доставки

PriorityBlockingQueue – блокирующая отсортированная очередь

Наиболее полезным является класс ConcurrentHashMap – он позволяет делать кэши, которые можно использовать из нескольких потоков:

```
private final  
ConcurrentMap primes = new ConcurrentHashMap<>(); // Этот метод  
потокобезопасен:  
public int getNthPrime(int n) {  
    return  
primes.computeIfAbsent(n, k -> { // вычисление n-го простого числа });}
```

Класс ConcurrentHashMap оптимизирован для использования многими потоками: если несколько потоков только читают данные из него, то они не блокируются; только при одновременной записи в ConcurrentHashMap возможна блокировка потоков, но и при этом если запись идет в разные buckets, то они не блокируются друг с другом.

## **39. Потокобезопасные коллекции пакета java.util.concurrent:**

Смотреть предыдущий вопрос

TransferQueue – блокирующая очередь с подтверждением доставки

BlockingDeque – блокирующая двусторонняя очередь

Другие классы, связанные с очередями:

ConcurrentLinkedQueue – неблокирующая очередь

ConcurrentLinkedDeque – неблокирующая двусторонняя очередь

DelayQueue – блокирующая очередь с задержкой

LinkedBlockingDeque – блокирующая двусторонняя очередь

LinkedTransferQueue – блокирующая очередь с подтверждением

Классы, реализующие интерфейс BlockingQueue: LinkedBlockingQueue: на основе двусвязного списка, размер может быть неограничен  
ArrayBlockingQueue: на основе массива, размер фиксирован  
SynchronousQueue: очередь “нулевой длины”; добавление элемента в хвост блокируется, пока другой поток не заберет этот элемент из головы Все эти классы являются потокобезопасными, т.к. спроектированы специально для использования разными потоками одновременно.

## **40. Реализация асинхронного выполнения в Джава**

Отсюда рождается идея дробить обработку запроса на более мелкие части, чтобы на время ожидания ввода/вывода отдать поток другим ожидающим запросам. Вместо: String text = readFromFile(); processText(text); пишем: readFromFile(text -> { processText(text); }); где метод readFromFile,

пока файл не будет прочитан, отдает поток другим запросам; `processText` может быть вызван уже не в том потоке, что изначальный вызов `readFromFile`

В Java это реализовано через `CompletableFuture` (аналог `Promise` в JavaScript). К сожалению, асинхронное программирование очень плохо ложится на традиционные инструменты разработки и идиомы императивного программирования. Асинхронный код трудно писать, читать, тестировать и отлаживать.

В будущих версиях Java планируется добавление виртуальных потоков (Project Loom). В отличие от текущей реализации, где поток Java = потоку операционной системы, виртуальные потоки будут управляться JVM. Такие потоки используют меньше памяти и намного быстрее запускаются. При этом при использовании виртуальным потоком блокирующих операций он отдает “физический поток” другим виртуальным потокам. Поэтому простой подход “поток на каждый запрос” в модели виртуальных потоков будет работать и для большого количества запросов, и не нужно будет уродовать программу асинхронным подходом.

## 41. Паттерн “Одиночка” (Singleton) и его использование в Java программах

```
public static class A {  
    public static final A INSTANCE = new A();  
    private A() {}  
}
```

OR

```
public static class B {  
    private static B b = null;  
    private B() {}  
  
    public static B getInstance()  
    { return b == null ? b = new B() : b; }  
}
```

Опишите паттерн singleton, приведите пример реализации

(41) и (3)  
Паттерн Singleton (единичка) - повторяющий паттерн, который гарантирует, что класс имеет только один экземпляр и представляет к нему посыпанную почку доступа.

Несообщаемость паттерна Singleton: часто требуется не сообщать о себе, чтобы некоторые объекты могли существовать только в единственной экземпляре. Например:

- Один механизм записи информации в журнал (логер)
- Доступ к СУБД
- Собирщик мусора

По сути, нам нужно создать единственный объект класса и хранить его в статической переменной и запретить создание второго объекта этого класса.

Что можно использовать вместо Singleton:

(1) Глобальные переменные, но они не защищены от записи, поэтому любой код может подменять их значение без ведома.

К глобальной переменной можно организовать способ доступа. Если присвоение многопоточное, и если менять значение глобальной переменной из разных потоков, то могут возникнуть невозможные негативные эффекты.  
Возможна ситуация, когда разные потоки передают данные друг у друга.

(2) Обычный конструктор класса, но он всегда возвращает новый объект

(3) Статические методы (откапывают от создания объектов и создают класс, у которого все методы статические). Часто это оказывается неудобно. Как создать такой класс, чтобы он был Singleton?

(4) Приватный конструктор. Ограничивает возможность создания объектов класса за пределами самого класса. Использование приватного конструктора - защищает класса от временного копирования. Здесь, по сути, делаем один единственный экземпляр класса.

② Публичный статический метод, который возвращает экземпляр класса.  
Данный метод часто называют `get Instance()`. Он является публичной  
методой доступа к экземпляру класса. Из-за чего можно его не вызывать,  
он всегда будет отдавать один и тот же объект. Здесь, по сути, мы делаем  
механический единственный метод, только через который можно получать ссылку на этот  
единственный экземпляр класса.

Реализации:

(1-ая (простая) реализация:

```
public class Singleton {  
    private static final Singleton INSTANCE = new Singleton();  
    private Singleton() {}  
    public static Singleton getInstance() {  
        return INSTANCE;  
    }  
}
```

}

(+):

1. Простота и прозрачность кода 2. Томокодежность 3. Вызывается синхронизацией

(-):

1. Не ленивая реализация. В любой момент времени один экземпляр класса, даже если он не нужен, то есть в любой момент существует память.

(2-ая (ленивая) реализация:

```
public class Singleton {  
    private static Singleton INSTANCE;  
    private Singleton() {}  
    private static Singleton getInstance() {  
        if (INSTANCE == null) {  
            INSTANCE = new Singleton();  
        }  
        return INSTANCE;  
    }  
}
```

(+):

1. Простота и прозрачность кода  
2. Ленивая реализация  
3. Экземпляр создается только если он нужен

(-):

1. Не томокодежно  
(потребуется синхронизированием  
иначе кода, который перенесет  
новый объект)

## **42. Использование статических методов для создания экземпляра объекта вместо конструкторов**

Используется, когда нужно много разных способов создания объекта, и не нужно плодить лишние конструкторы.

Преимущества:

- Статическому методу можно дать осмысленной имя;
- Следуя из первого, возможно наличие методов с одинаковыми параметрами но разной логикой;
- Из статического метода можно вернуть null если что то пошло не так. Конструктор же всегда вернет что-то, или же выбросит исключение
- Можно вернуть тип, отличный от объявленного (например, вернуть класс-наследник)
- Необходим для синглтона

Статические и нестатические методы отличаются (главным образом) только наличием одной вещи - контекстом. Если код суммирует два аргумента (пресловутый Math) или преобразовывает массив в список (Arrays.asList()), то ему не нужен для этого внешний контекст (состояние), и такой код, по логике, должен быть статическим. В случаях, когда вызов метода меняет состояние, и без экземпляра класса не обойтись (например, добавление новой секции к отчету), код, конечно, не должен быть статическим, и метод ReportSectionAppender.append(Report report, ReportSection section) без дополнительных условий появляться не должен.

Однако лично у меня есть сильные предубеждения против статических вызовов, из-за которых лично я этой логике не следую. Статические вызовы облегчают написание кода, но у них есть два минуса, которые (лично для меня) перевешивают плюсы. Во-первых, статические вызовы скрывают зависимости и вносят небольшую долю глобального контекста в код - в конструктор класса может передаваться три объекта, а внутри использоваться еще с десяток статических классов, что обнаружится только в тот момент, когда на рефакторинг будет дан зеленый свет, "потому что этот класс почти не затрагивает другие". И, во-вторых, это довольно жесткий хардкод - статический вызов сам по себе нельзя ни подменить другой реализацией, ни обернуть в прокси, не изменив сам код. В случае того же самого рефакторинга это может встать

серьезным боком, если архитектура не была продумана идеально изначально (это как раз мой случай).

Что по поводу "лишь бы не создавать экземпляр класса" - это обычные опасения, от которых стоит избавляться. Когда вы пишете программу, ваша первая задача - реализовать поставленные к ней требования, оптимизацией - если она вообще потребуется - можно будет заняться позже. Делайте так, как удобно.

Резюмируя: писать бесконтекстные методы статикой и выносить их в utility-классы - это обычная практика, принятая коммьюнити. Тем не менее, у этого подхода есть минусы, которые заставляют некоторых ее не использовать или использовать по минимуму.

При использовании static нужно быть внимательным к тому где и как вы его используете.

Так как static, фактически, означает, что этот объект *singleton*, т.е. он (класс) и его поля существуют в единственном экземпляре, и используя его в других классах вы используете один и тот же объект, а изменяя его состояние, к примеру, в классе N, вы изменяете его и для остальных классов.

Зачастую их используют как util-ные классы, к примеру когда вам нужно в каком то месте в коде произвести подсчет, или конвертацию какого то значения. Хороший пример [java.lang.Math](#), класс который используется для математических операции.

Либо как объект который содержит в себе значения которые могут быть нужны и являются общими для остальных классов, к примеру это могут быть какие то константы которые могут использовать остальные классы из этого приложения. И чтобы каждый раз не создавать объект который содержит всегда одинаковую информацию - создают его один для всех.

Мое мнение:

1. для приватных методов класса, если можно сделать метод статически - обязательно делайте его статическим, это покажет всем что метод не связан с контекстом класса и его можно вынести в другой класс или легко рефакторить.
2. для публичным методов класса - используйте static очень ограниченно:
  - во-первых, лучше если static методы будут находятся в отдельных Utils классах. Не стоит смешивать в одном классе статические и обычные методы
  - во-вторых, статические публичные методы лучше использовать только для методов, которые делают простые операции одним единственным способом. Например, такие как взятие синуса, округление и т.п.

- в-третьих, никогда не используйте статические методы если возможно потребуется их расширение на разные сущности или разные виды реализации. Например, если вы делаете методы parseXML(url), parseTxt(url), parseHTML(url), которые делают похожие вещи но с разными источниками намного лучше создать интерфейс, имеющий метод parse(url) и три класса XmlPraser, TxtPraser и HTMLParser, которые реализуют этот метод. Это упростит понимание и позволит потом легко и просто добавить четвертый и пятый источник.
3. Чтобы избавится от необходимости "создавать экземпляр класса" советую посмотреть на dependency injection фреймворки, такие как Spring или guice. Они позволяют легко и незаметно "создавать экземпляры класса", например сравните:

код с использованием DI

```
public class myClass { @Inject Parser parser; public void myMethod() { parser.parser(url); }}
```

код с использованием статических методов

```
public class myClass { public void myMethod() { Parser.parser(url); }}
```

Не такая большая разница, не так ли? При этом, при использовании DI вы сохраните гибкость, возможность переопределения при Unit тестировании, легкость расширения и т.п.

```
public static User createWithDefaultCountry(String name, String email)
{
    return new User(name, email, "Argentina");
}
```

Статические методы используются вместо конструкторов, чтобы не рефакторить код конструктора, когда нужно значение по умолчанию и тд.

## 43. Паттерн “Строитель” и его использование большом количестве параметров конструктора

Напишу про паттерн “Строитель” - Давай, одобряю, но выше он уже был)(Ссылка <https://habr.com/ru/post/86252/>)

Крч, он упрощает ситуацию, вместо трёхкилометрового кода, будет трехкилометровый паттерн, но зато вы можете понтануться им.

Ну вот еще ссылка, но там более сложный случай рассматривается (<https://habr.com/ru/company/otus/blog/552412/>)

## Паттерн проектирования Builder

- Паттерн проектирования Builder разработан для обеспечения гибкого решения различных задач создания объектов в объектно-ориентированном программировании.
- Паттерн проектирования Builder позволяет отделить построение сложного объекта от его представления.
- Паттерн Builder создает сложные объекты, используя простые объекты и поэтапный подход.
- Паттерн предоставляет один из лучших способов создания сложных объектов.
- Это один из [паттернов проектирования банды четырех \(GoF\)](#), которые описывают, как решать периодически возникающие задачи проектирования в объектно-ориентированном программном обеспечении.
- Этот паттерн полезен для создания разных иммутабельных объектов с помощью одного и того же процесса построения объекта.

Паттерн Builder — это паттерн проектирования, который позволяет поэтапно создавать сложные объекты с помощью четко определенной последовательности действий. Строительство контролируется объектом-распорядителем (director), которому нужно знать только тип создаваемого объекта.

Вместо непосредственного создания желаемого объекта, клиент вызывает конструктор (или статическую фабрику) со всеми необходимыми параметрами и получает объект строителя. Затем клиент вызывает *сеттер-подобные* методы у объекта строителя для установки каждого дополнительного параметра. Наконец, клиент вызывает метод `build()` для генерации объекта, который будет являться неизменным(immutable). Строитель является статическим внутренним классом в классе, который он строит.

```
// паттерн Builder
public class NutritionFacts {
    private final int servingSize;
    private final int servings;
```

```
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // Обязательные параметры
        private final int servingSize;
        private final int servings;
        // Дополнительные параметры – инициализируются
        // значениями по умолчанию
        private int calories = 0;
        private int fat = 0;
        private int carbohydrate = 0;
        private int sodium = 0;

        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings = servings;
        }

        public Builder calories(int val) {
            calories = val;
            return this;
        }

        public Builder fat(int val) {
            fat = val;
            return this;
        }

        public Builder carbohydrate(int val) {
            carbohydrate = val;
            return this;
        }
    }
}
```

```
public Builder sodium(int val) {
    sodium = val;
    return this;
}

public NutritionFacts build() {
    return new NutritionFacts(this);
}

private NutritionFacts(Builder builder) {
    servingSize = builder.servingSize;
    servings = builder.servings;
    calories = builder.calories;
    fat = builder.fat;
    sodium = builder.sodium;
    carbohydrate = builder.carbohydrate;
}
}
```

#### **44. Понятие dependency injection (внедрение зависимости).**

**Преимущества использования внедрения зависимостей при написании программ на языке Джава. Преимущества этого подхода перед паттерном Одиночка**

Внедрение зависимостей — это стиль настройки объекта, при котором поля объекта задаются внешней сущностью. Другими словами, объекты настраиваются внешними объектами. DI — это альтернатива самонастройке объектов

Главный плюс в том, что мы можем рассматривать приложение, как набор сервисов или модулей. Сам по себе код получается чуть компактней и не нужно завязываться на имя класса.

При развитии программ очень часто оказывается, что некоторая сущность, которая ранее присутствовала в 1 экземпляре, требуется в нескольких экземплярах.

Поэтому использованию Singleton предпочтите dependency injection

## **45. Преимущество использования try-с-ресурсами по сравнению с использованием try-finally. Интерфейс интерфейса AutoCloseable**

При завершении работы с потоком его надо закрыть с помощью метода close(), который определен в интерфейсе Closeable.

При закрытии потока освобождаются все выделенные для него ресурсы, например, файл. В случае, если поток окажется не закрыт, может происходить утечка памяти.

Есть два способа закрытия файла. Первый традиционный заключается в использовании блока try..catch..finally.

Поскольку при открытии или считывании файла может произойти ошибка ввода-вывода, то код считывания помещается в блок try. И чтобы быть уверенным, что поток в любом случае закроется, даже если при работе с ним возникнет ошибка, вызов метода close() помещается в блок finally. И, так как метод close() также в случае ошибки может генерировать исключение IOException, то его вызов также помещается во вложенный блок try..catch

Начиная с Java 7 можно использовать еще один способ, который автоматически вызывает метод close. Этот способ заключается в использовании конструкции try-with-resources (try-с-ресурсами). Данная конструкция работает с объектами, которые реализуют интерфейс AutoCloseable. Так как все классы потоков реализуют интерфейс Closeable,

который в свою очередь наследуется от AutoCloseable, то их также можно использовать в данной конструкции

Синтаксис конструкции следующий: try(название\_класса имя\_переменной = конструктор\_класса). Данная конструкция также не исключает использования блоков catch.

После окончания работы в блоке try у ресурса (в данном случае у объекта FileInputStream) автоматически вызывается метод close().

Если нам надо использовать несколько потоков, которые после выполнения надо закрыть, то мы можем указать объекты потоков через точку с запятой:

```
try(FileInputStream fin=new  
FileInputStream("C://SomeDir//Hello.txt");  
FileOutputStream fos = new  
FileOutputStream("C://SomeDir//Hello2.txt"))  
{  
    //.....  
}
```

## **46. Паттерн Декоратор (Decorator) Преимущества его использовании (например композиция по сравнению с наследованием)**

Главное отличие декоратора от наследования: декоратор позволяет добавлять функционал к отдельному объекту в рантайме. Наследование добавляет логику только на этапе компиляции и для всех объектов этого типа.

<https://javarush.ru/groups/posts/3833-pattern-proektirovaniya-dekorator-s-primerami>

**Декоратор (англ. Decorator)** — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту. Шаблон Декоратор предоставляет гибкую альтернативу практике создания подклассов с целью расширения функциональности.

**Адаптор** - частный случай декоратора. Отличие скорее в том, что декоратор может быть масштабнее (содержать в себе много сущностей, в то время как адаптор, как правило, реализует интерфейс для определенного типа). Прокси - противоположность первых двух, т.к. ее задача - быть прозрачной на стыке взаимодействия типов и не менять интерфейс.

```
public interface InterfaceComponent {
    void doOperation();
}

class MainComponent implements InterfaceComponent {

    @Override
    public void doOperation() {
        System.out.print("World!");
    }
}

abstract class Decorator implements InterfaceComponent {
    protected InterfaceComponent component;

    public Decorator (InterfaceComponent c) {
        component = c;
    }

    @Override
    public void doOperation() {
        component.doOperation();
    }

    public void newOperation() {
        System.out.println("Do Nothing");
    }
}

class DecoratorSpace extends Decorator {

    public DecoratorSpace(InterfaceComponent c) {
        super(c);
    }

    @Override
    public void doOperation() {
        System.out.print(" ");
        super.doOperation();
    }

    @Override
    public void newOperation() {
        System.out.println("New space operation");
    }
}
```

```

@Override
public void newOperation() {
    System.out.println("New space operation");
}
}

class DecoratorComma extends Decorator {

    public DecoratorComma(InterfaceComponent c) {
        super(c);
    }

    @Override
    public void doOperation() {
        System.out.print(",");
        super.doOperation();
    }

    @Override
    public void newOperation() {
        System.out.println("New comma operation");
    }
}

class DecoratorHello extends Decorator {

    public DecoratorHello(InterfaceComponent c) {
        super(c);
    }

    @Override
    public void doOperation() {
        System.out.print("Hello");
        super.doOperation();
    }

    @Override
    public void newOperation() {
        System.out.println("New hello operation");
    }
}

class Main {

    public static void main (String... s) {
        Decorator c = new DecoratorHello(new DecoratorComma(new DecoratorSpace(new MainComponent())));
        c.doOperation(); // Результатом выполнения программы "Hello, World!"
        c.newOperation(); // New hello operation
    }
}

```

## 47. Преимущества использования списков перед массивами

Массив выигрывает в скорости, однако его предпочтительнее использовать в том случае, когда заранее известно число элементов.

Списки более медленные, однако работать с ними удобнее и проще, дженерики можно применять. Сами по себе являются динамическими. Кроме того возможно изменение подтипа за счет задания через интерфейс, что обеспечивает гибкость (`List<String> list = new ArrayList<String>();`)

## 48. Основные системы сборки Gradle и Maven.

### Использование Gradle и основная терминология. Управление зависимостями в Gradle

Основные используемые системы сборки – Maven и Gradle.

Gradle создан в 2007 году, текущая версия – 7.0.

Gradle использует для конфигурации проекта полноценный язык программирования; изначально Groovy, с версии 3.0 также поддерживается Kotlin.

### Gradle – терминология

build		Набор библиотек и приложений. Большие проекты, как правило, состоят из нескольких подпроектов
project	(под)проект	Ваша библиотека или приложение. Кроме исходного кода, для проекта должно быть описание (build.gradle)
task	задача	Задача, которую можно выполнить над проектом. Например, <code>compileJava</code> – скомпилировать исходный код на Java
plugin	плагин	Модуль Gradle, предназначенный для выполнения класса задач. Например, <code>application</code> – плагин для сборки приложений Java. Плагин добавляет описания задач, которые он умеет выполнять.

#### Gradle tasks

Как правило, задачи (tasks) создаются плагинами (рекомендуемый способ). Но можно их создавать и вручную:

```
tasks.register("hello") {  
    doLast {  
        println("Hello task")  
    }  
}
```

создает задачу hello, которую можно выполнить: gradlew hello

Фазы выполнения

Файлы `settings.gradle` и `build.gradle` содержат обычный исполняемый код на языке Groovy (для Kotlin эти файлы должны называться `settings.gradle.kts` и `build.gradle.kts`).

### Фазы выполнения

При запуске Gradle эти файлы выполняются. Но выполнение идет в три фазы:

- фаза инициализации: выполнение `settings.gradle`
- фаза конфигурации: при выполнении `build.gradle` создаются и конфигурируются задачи
- фаза выполнения: запускается указанная при запуске Gradle задача и те задачи, от которых она зависит

### Конфигурация проектов

- параметры `group` и `version` описывают группу и версию, под которой проект будет публиковаться Если вы планируете использование вашего проекта другими разработчиками, эти параметры нужно указывать
- параметр `description` – человекочитаемое описание проекта
- `sourceCompatibility` указывает, какую версию Java использует проект
- `options.encoding` указывает кодировку исходного кода

### Управление зависимостями

Кроме запуска компиляции, Gradle умеет автоматически скачивать нужные для сборки приложения библиотеки. Для этого нужно указать:

- репозиторий, из которого качать библиотеки  
(обычно это `mavenCentral: https://repo1.maven.org/maven2`)
- “адрес” библиотеки в виде тройки
  - `group` – группа, в которую входит библиотека
  - `name` – имя библиотеки
  - `version` – версия библиотеки

Проекты Maven в первую очередь определяются файлами объектной модели проекта (POM), написанными в XML. Эти файлы POM.xml содержат зависимости проекта, плагины, свойства и данные конфигурации. Maven использует декларативный подход и имеет предопределенный жизненный цикл.

## Maven или Gradle: различия

Некоторые из ключевых различий между Maven и Gradle:

- Язык сценария сборки: сценарий сборки Gradle по своей сути более универсален и эффективен, чем Maven. Это связано с тем, что Gradle основан на языке программирования (Groovy), а Maven — на языке разметки (XML). Предостережение здесь в том, что скрипт сборки Gradle уязвим для ошибок, поскольку он основан на языке программирования.
- Производительность: Gradle реализует такие стратегии, как кэш сборки и инкрементные компиляции, чтобы обеспечить высокую производительность. Gradle утверждает, что он работает до семи раз быстрее, чем Maven для инкрементных изменений, и в три раза быстрее, когда выходные данные задачи кэшируются. Однако отнеситесь к этому с недоверием. Есть разработчики, которые считают Maven более быстрым из двух.
- Гибкость и простота настройки: скрипт сборки Gradle на основе Groovy предлагает большую гибкость, чем Maven XML. Например, вы можете написать настройки плагина прямо в скрипт сборки Gradle. Gradle также более эффективен, если вы хотите настроить артефакты сборки и структуру проекта. Хотя Maven также обладает широкими возможностями настройки, его конфигурация на основе XML требует нескольких дополнительных шагов для настройки вашей сборки.

- Плагины: Maven существует дольше, чем Gradle. По этой причине доступно больше плагинов Maven, и больше крупных поставщиков поддерживают плагины Maven, чем плагины Gradle.
- Управление зависимостями: два инструмента сборки используют разные подходы для разрешения конфликтов зависимостей. Maven следует порядку объявления, а Gradle ссылается на дерево зависимостей.

## 49. Анатомия jar. Сканирование пакетов

### Анатомия jar

- Jar это **zip** архив специального вида!
- Внутри jar находятся **байт код программы, ресурсы** а также сохранена переменная **classpath** для данной программы.
- **Напоминание!** Classpath это переменная в которой указаны пути до всех классов программы и самое главное майн класс.

### Сканирование пакетов

- Одной из ключевых особенностей JVM является **динамическая загрузка классов**. Достигается это благодаря сущности **загрузчика классов**.
- Идея состоит в том что мы можем двигаться по классам переменной classpath и загружать их при помощи classloader. Загружая классы являющие **подтипов** данного что реализует библиотека **reflections**.

## 50. Реализация REST API с помощью Spring Framework

Задача веб-сервера – выдать данные для отображения в браузере (обычно в формате JSON). Эти данные обрабатываются отдельным frontend приложением.

REST – стиль взаимодействия клиента с сервером. Обычно он подразумевает запросы и ответы в формате JSON, где адрес запроса содержит информацию о том, что хочет сделать клиент:

- GET /api/books – получить список всех книг
- GET /api/books/11 – получить информацию о книге с id=11
- POST /api/books – создание записи о книге; данные о создаваемой книге передаются в теле запроса
- PUT /api/books/15 – обновление записи о книге с id=15; новые данные о книге передаются в теле запроса

Итого задача серверной части веб-приложения при обработке запроса:

1. Понять, какой запрос пришел (например, запрос /api/books для получения списка всех книг)
2. Обратиться к базе данных (или другому источнику данных) для получения запрошенных данных в виде обычных Java-объектов (в нашем примере списка книг)
3. Сериализовать Java-объекты в формат JSON и отправить в качестве ответа на запрос

Архитектуры реализуем с помощью MVC:

1. Слой представления отвечает за взаимодействие с клиентом: сериализацию/десериализацию JSON, маршрутизацию (routing) – определение, какое действие было запрошено (/api/books или /api/books/{id})
2. Слой бизнес-логики отвечает за логику работы приложения (в простейшем случае просто перенаправление запроса к слою хранения данных)
3. Слой хранения данных отвечает за взаимодействие с базой данных: получение данных из БД и сохранение данных в БД (в случае запроса /api/books – получение из БД списка книг)

Поверх веб-сервера часто используется фреймворк, упрощающий рутинную работу: Например, Spring/Spring Boot

Spring Boot – расширение Spring Framework, предоставляющее конфигурации для решения стандартных задач, таких как веб-приложения. Включает в себя:

- предустановленный набор библиотек
- встроенный веб-сервер
- средства мониторинга приложения
- логика задается аннотациями

Как тестировать REST API?

- встроенный в IntelliJ IDEA клиент HTTP
- curl – утилита командной строки
- Postman

## 51. Понятие Инверсии управления

Спринг помнишь? Ты сам не кодируешь логику по которои управление из точки входа в прогу переходит например в метод маршрутизации запросов, за тебя это делает фреймворк. Это и есть твоя инверсия - когда потоком управления проги заведует фреймворк, а ты лишь реагируешь на внутренние вызовы (например как тот же вызов метода маршрутизации запроса, в котором ты сам пишешь логику). Я это так понял

Инверсия управления - предоставление callback-а в качестве реакции на какие либо события извне, вместо того, чтобы реализовывать логику обработки события на месте.

Инверсия управления (англ. *Inversion of Control*, IoC) — важный принцип объектно-ориентированного программирования, используемый для уменьшения зацепления (связанности) в компьютерных программах. Также архитектурное решение интеграции, упрощающее расширение возможностей системы, при котором поток управления программы контролируется фреймворком.

Одной из реализаций инверсии управления в применении к управлению зависимостями является [внедрение зависимостей](#) (англ. *dependency injection*)<sup>[2][3]</sup>. Внедрение зависимости используется во многих [фреймворках](#), которые называются IoC-контейнерами.

Если сравнить с более низкоуровневыми технологиями, IoC-контейнер — это [компоновщик](#), который собирает не [объектные файлы](#), а объекты [ООП](#) ([экземпляры класса](#)) во [время исполнения](#) программы. Очевидно, для реализации подобной идеи было необходимо создать

не только сам компоновщик, но и [фабрику](#), производящую объекты. Аналогом такого компоновщика (естественно, более функциональным) является [компилятор](#), одной из функций которого является создание объектных файлов. В идеи компоновки программы во время исполнения нет ничего нового. Предоставление программисту инструментов внедрения зависимостей дало значительно большую гибкость в разработке и удобство в тестировании кода<sup>[4]</sup>.

The **Inversion-of-Control (IoC)** pattern, is about providing *any kind* of callback (which controls reaction), instead of acting ourself directly (in other words, inversion and/or redirecting control to external handler/controller). The **Dependency-Injection (DI)** pattern is a more specific version of IoC pattern, and is all about removing dependencies from your code.

Every DI implementation can be considered IoC, but one should not call it IoC, because implementing Dependency-Injection is harder than callback (Don't lower your product's worth by using general term "IoC" instead).

For DI example, say your application has a text-editor component, and you want to provide spell checking. Your standard code would look something like this:

```
public class TextEditor { private SpellChecker checker; public TextEditor() { this.checker = new SpellChecker(); } }
```

What we've done here creates a dependency between the TextEditor and the SpellChecker. In an IoC scenario we would instead do something like this:

```
public class TextEditor { private locSpellChecker checker; public TextEditor(locSpellChecker checker) { this.checker = checker; } }
```

In the first code example we are instantiating SpellChecker (this.checker = new SpellChecker());, which means the TextEditor class directly depends on the SpellChecker class.

In the second code example we are creating an abstraction by having the SpellChecker dependency class in TextEditor's constructor signature (not initializing dependency in class). This allows us to call the dependency then pass it to the TextEditor class like so:

```
SpellChecker sc = new SpellChecker(); // dependency TextEditor textEditor = new TextEditor(sc);
```

Now the client creating the TextEditor class has control over which SpellChecker implementation to use because we're injecting the dependency into the TextEditor signature.

```
class TodoService {  
    TodoRepository todoRepository;  
    TodoService(TodoRepository todoRepository) {
```

```
        this.todoRepository = todoRepository;
    }
}

class Application {
    public static void main(String[] args) {
        TodoRepository todoRepository = new TodoRepository();
        TodoService todoService = new TodoService(todoRepository);
    }
}
```

## 52. Spring Boot и его использование

### В чем разница между Spring и Spring Boot?

Как отмечено выше, Spring — это платформа приложений на основе Java с открытым кодом, которая охватывает множество небольших проектов. К таким [проектам Spring](#), помимо прочего, относятся Spring Data, Spring Cloud и Spring Security. Чтобы понять разницу между Spring Boot и Spring, важно знать, что, несмотря на различия в основных возможностях, они входят в семейство Spring.

Иными словами, Spring - Платформа веб-приложений с открытым кодом на Java, а Spring Boot - расширение/модуль, созданный на платформе Spring.

Spring boot Предоставляет возможность создавать автономные приложения Spring, которые можно запускать сразу же без заметок, конфигурации XML и написания больших объемов дополнительного кода.

Используйте Spring Boot, когда захотите:

- Простота использования
- Подход на основе рекомендаций\*.
- Для быстрого создания качественных приложений и сокращения времени разработки.
- Отказ от необходимости писать шаблонный код и настраивать XML.
- Разработка API REST.

Spring Boot удобен для написания микросервисов.

Spring Boot Берет на себя все рутинные действия по созданию Spring-приложений и ускоряет вашу работу настолько, насколько это возможно.

### **53. Работа с базами данных в джава приложениях.**

#### **Встроенные СУБД для Джава приложений.**

JDBC в своей основе имеет концепцию драйверов. Driver позволяет получать соединение (getconnection) с БД. Для реализации поставленной задачи задействуют специальные URL-адреса. Драйверы заключаются динамически (тогда, когда используемая утилита функционирует). Алгоритм «активации» будет следующим:

1. Происходит загрузка софта.
2. Драйвер инициализируется и загружается.
3. Осуществляется самостоятельная регистрация drivers.
4. Вызов производится «автоматом». Это происходит тогда, когда используемое приложение требует URL с протоколом, за который отвечают драйверы.

JDBC использует экземпляры классов java.sql. После того, как это было сделано, происходит передача тех или иных команд для корректировки информации. JDBC посредством драйверов взаимодействует с СУБД и выводит тот или иной результат.

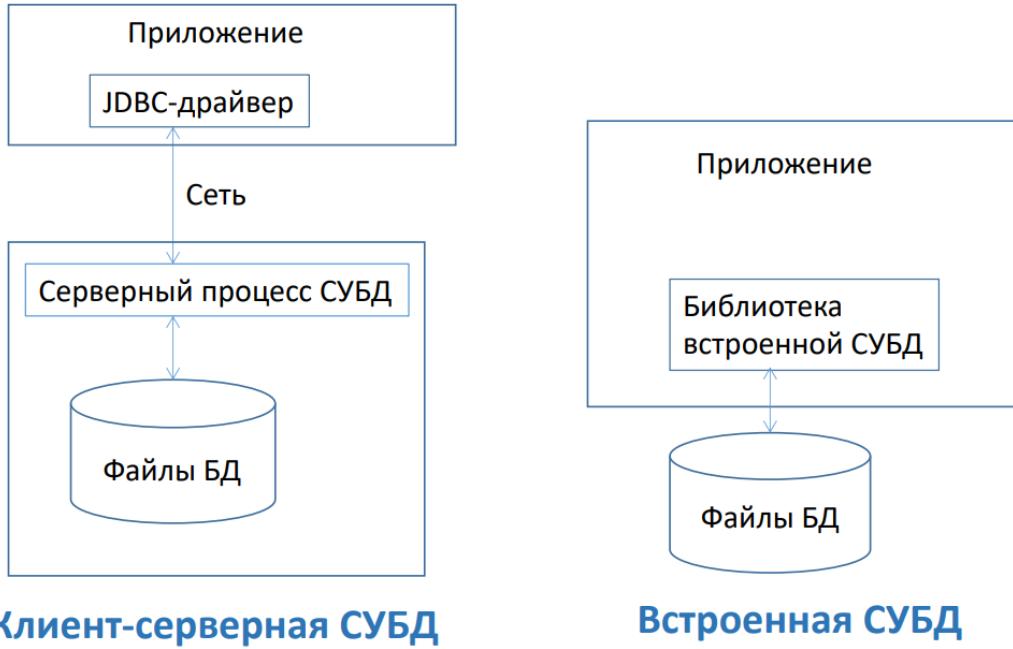
JDBC работает с «электронными хранилищами информации» через специальные запросы. О них необходимо знать каждому потенциальному разработчику до того, как будет рассмотрен образец применения БД на практике в приложении.

Загрузка драйвера в память производится так: `Class.forName ("com.mysql.jdbc.Driver");`. Далее происходит установка соединения с имеющимся хранилищем. Для этого используют команду `Connection cn = DriverManager.getConnection("jdbc:mysql://localhost/my_db", "login", "password");`. Для URL применяется следующий шаблон:

```
jdbc: <название_драйвера>[://хост[: порт/]] <название подключаемой базы  
данных>
```

После установления connections происходит import java information. Система получает запрос и создает специальный объект для его последующей передачи. Завершающий этап работы JDBC – это закрытие всех имеющихся соединений.

# Реляционные базы данных



Клиент-серверные СУБД (системы управления базами данных):

- PostgreSQL
- Oracle
- MySQL
- Microsoft SQL Server

Встроенные СУБД:

- H2
- Apache Derby

## 54. Реляционные СУБД для работы с java

### приложениями. Пакет `java.sql` и его классы

Реляционная база данных — это набор таблиц, между которыми установлены определенные взаимосвязи. Для обслуживания реляционной базы данных и создания запросов к ней система управления базой данных использует язык структурированных запросов (Structured Query Language, SQL) — обычное пользовательское приложение, предоставляющее простой интерфейс программирования для взаимодействия с базой данных.

Реляционные базы данных состоят из строк, называемых кортежами, и столбцов, называемых атрибутами.

## Реляционные базы данных

Язык SQL, под-язык Data Manipulation Language (DML):

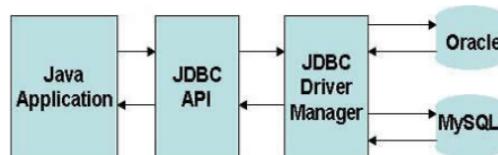
- Create:  
`INSERT INTO books (author, title)  
VALUES  
( 'Л.Н. Толстой', 'Война и мир' )`
- Retrieve:  
`SELECT id, author, title  
FROM books`
- Update:  
`UPDATE books  
SET author = 'Л.Н. Толстой', title = 'Война и мир'  
WHERE id = 1`
- Delete:  
`DELETE FROM books WHERE id = 1`



## Как это работает? Роль JDBC

Пакет `java.sql` – классы JDBC (Java Database Connectivity – соединение с БД на Java):

- Connection
- PreparedStatement
- ResultSet
- SQLException
- DriverManager
- ...

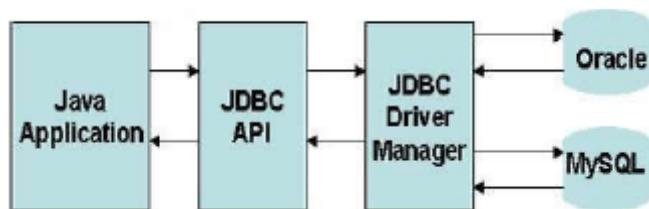


Концепция JDBC – драйверы позволяющих получать соединение с базой данных по специально описанному URL.

[online.mirea.ru](http://online.mirea.ru)

## 55. Роль интерфейса JDBC для работы с джава приложениями

JDBC создает прослойку между Java приложением и драйвером базы данных. JDBC позволяет устанавливать соединение с источником данных, отправлять запросы и операторы обновления, а также обрабатывать результаты. Проще говоря, JDBC позволяет делать в Java-приложении следующие вещи: Устанавливать соединение с источником данных. Отправляйте запросы и операторы обновления в источник данных.



Получается, что программисту не важно, какая реализация находится под капотом JDBC Driver Manager - приложение работает через интерфейс JDBC и никак не зависит от драйвера.

## 56. Основные компоненты JDBC API



### *DriverManager:*

Это класс, использующийся для управления списком Driver (database drivers).

### *Driver:*

Это интерфейс, использующийся для соединения коммуникации с базой данных, управления коммуникации с базой данных.

### *Connection :*

Интерфейс со всеми методами связи с базой данных. Он описывает коммуникационный контекст. Вся связь с базой данных осуществляется только через объект соединения (connection).

### *Statement :*

Это интерфейс, включающий команду SQL отправленный в базу данных для анализа, обобщения, планирования и выполнения.

### *ResultSet :*

*ResultSet* представляет набор записей, извлеченных из-за выполнения запроса.

## 57. JDBC URL и его использование

### JDBC URL

`jdbc:<субпротокол>:<имя, связанное с СУБД или протоколом>`

PostgreSQL: `jdbc:postgresql://localhost:5432/postgres`

Oracle: `jdbc:oracle:thin:@//localhost:1521/orcl`

H2: `jdbc:h2:~/example_db`

H2 – встроенная СУБД:

```
dependencies {  
    runtimeOnly("com.h2database:h2:1.4.200")  
}
```

Connection c = DriverManager.getConnection("jdbc:h2:~/example\_db")

online.mirea.ru

## 58. Работа JDBC драйвера

### Реляционные базы данных

JDBC-драйвер – библиотека для выполнения SQL-запросов для конкретной БД

`java.sql.Connection`:

`oracle.jdbc.OracleConnection (ojdbc18.jar)`

`org.postgresql.jdbc.PgConnection (postgresql.jar)`

`org.h2.jdbc.JdbcConnection (h2.jar)`

**Если мы используем переносимый SQL, то программа сможет работать с любой БД, поддерживающей стандарты SQL!**

## **59. В чем заключается роль DI в Spring. Использования ServiceLoader**

Dependency Injection (DI) — это шаблон разработки программного обеспечения, который реализует инверсию управления для разрешения зависимостей. Инъекция — это передача зависимости зависимому объекту, который будет ее использовать. DI — это процесс, посредством которого объекты определяют свои зависимости.

### **Что использовать тогда? Итоги**

- **Spring — отличная штука, но бывают случаи, когда ServiceLoader будет правильным выбором.**

- **Скорость**

Для консольных приложений время запуска ServiceLoader **НАМНОГО** меньше, чем Spring Boot App.

**Память**

Если вам важно расходование памяти, то следует рассмотреть возможность использования ServiceLoader для DI.

**Модули Java**

Одним из ключевых аспектов Java-модулей была возможность полностью защитить классы в модуле от кода вне модуля.

## **60. Интерфейс Connection. Пулы соединений**

Интерфейс Connection описывает активное соединение с БД (в случае клиент-серверной БД – это сетевое соединение).

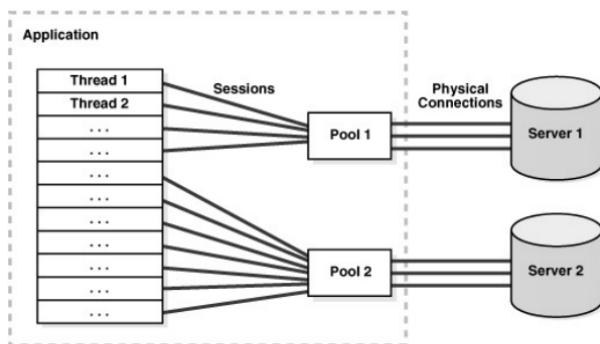
```
public class BookRepository {  
    public List getAllBooks() {  
        try (Connection connection = ???) {  
            }  
    }  
}
```

Плохие варианты работы с соединением:

1. Открывать соединение каждый раз в методах BookRepository:  
Connection connection = DriverManager.getConnection("...") Соединение открывается медленно (порядка 0.1 секунд). Наше приложение не будет справляться с большой нагрузкой

2. Держать Connection в поле BookRepository. Как правило, только один поток может одновременно работать с одним соединением. Хотя соединения обычно потокобезопасны, но это достигается с помощью синхронизации. Поэтому при большом количестве параллельных запросов к BookRepository только один из них будет работать, остальные будут ждать.

## Пулы соединений



**Пул соединений с базой данных это набор заранее открытых соединений с базой данных используемый для предоставления соединения в тот момент, когда оно требуется.**

[online.i](#)

Поэтому используются пулы соединений (по аналогии с пулами потоков). Например, пул с максимальным количеством соединений = 20 позволяет параллельно работать с БД 20 потокам. При этом соединения не закрываются, а по возможности переиспользуются.

Есть несколько реализаций пулов соединений:

- HikariCP
- Apache Commons DBCP
- C3PO

## 61. Принципы SOLID и их использование на Джава

### SOLID

**SOLID** (сокр. от англ. **single responsibility, open-closed, Liskov substitution, interface segregation** и **dependency inversion**) в программировании — мнемонический акроним, введённый Майклом Фэзерсом (*Michael Feathers*) для первых пяти принципов, названных Робертом Мартином<sup>[1][2]</sup> в начале 2000-х<sup>[3]</sup>, которые означали 5 основных принципов объектно-ориентированного программирования и проектирования.

**S** (принцип единой ответственности) - (single responsibility principle) Для каждого класса должно быть определено единственное назначение. Все ресурсы, необходимые для его осуществления, должны быть инкапсулированы в этот класс и подчинены только этой задаче.

**O(принцип открытости/закрытости)** - (open-closed principle) «программные сущности … должны быть открыты для расширения, но закрыты для модификации».

**L(принцип подстановки Лисков)** - (Liskov substitution principle) «функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа не зная об этом».

**I(принцип разделение интерфейсов)** - (interface segregation principle) «много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения».

**D(принцип инверсии зависимостей)** - (dependency inversion principle) «Зависимость на Абстракциях. Нет зависимости на что-то конкретное»

\*Тут нужно написать про использование SOLID в Java\*

## **Примеры задач**

3. Опишите паттерн синглтон, приведите пример реализации

Опишите паттерн singleton, приведите пример реализации

(41) и (3)  
Паттерн Singleton (единичка) - повторяющий паттерн, который гарантирует, что класс имеет только один экземпляр и представляет к нему посыпанную почку доступа.

Несообщаемость паттерна Singleton: часто требуется не сообщать о себе, чтобы некоторые объекты могли существовать только в единственной экземпляре. Например:

- Один механизм записи информации в журнал (логер)
- Доступ к СУБД
- Собирщик мусора

По сути, нам нужно создать единственный объект класса и хранить его в статической переменной и запретить создание второго объекта этого класса.

Что можно использовать вместо Singleton:

(1) Глобальные переменные, но они не защищены от записи, поэтому любой код может подменять их значение без ведома.

К глобальной переменной можно организовать способ доступа. Если присвоение многопоточное, и если менять значение глобальной переменной из разных потоков, то могут возникнуть невозможные негативные эффекты.  
Возможна ситуация, когда разные потоки передают данные друг у друга.

(2) Обычный конструктор класса, но он всегда возвращает новый объект

(3) Статические методы (откапывают от создания объектов и создают класс, у которого все методы статические). Часто это оказывается неудобно. Как создать такой класс, чтобы он был Singleton?

(4) Приватный конструктор. Ограничивает возможность создания объектов класса за пределами самого класса. Использование приватного конструктора - защищает класса от внешнего копирования. Здесь, по сути, делаем один единственный экземпляр класса.

② Публичный статический метод, который возвращает экземпляр класса.  
Данный метод часто называют *get Instance()*. Он является публичной  
методой доступа к экземпляру класса. Из-за того что его не вызвать,  
он всегда будет отдавать один и тот же объект. Задача, по сути, это сделать  
механизм единственный метод, только через который можно получить ссылку на этот  
единственный экземпляр класса.

Реализации:

(1-ая (простая) реализация:

```
public class Singleton {  
    private static final Singleton INSTANCE = new Singleton();  
    private Singleton() {}  
    public static Singleton getInstance() {  
        return INSTANCE;  
    }  
}
```

}

(+):

1. Простота и прозрачность кода 2. Томокодежность 3. Вызывается синхронизацией

(-):

1. Не ленивая реализация. В любой момент времени один экземпляр класса, даже если он не нужен, то есть в любой момент существует память.

(2-ая (ленивая) реализация:

```
public class Singleton {  
    private static Singleton INSTANCE;  
    private Singleton() {}  
    private static Singleton getInstance() {  
        if (INSTANCE == null) {  
            INSTANCE = new Singleton();  
        }  
        return INSTANCE;  
    }  
}
```

(+):

1. Простота и прозрачность кода  
2. Ленивая реализация  
3. Экземпляр создается только если он нужен

(-):

1. Не томокодежно  
(потребуется синхронизированием  
из-за того что, когда перенесен  
новый объект)

#### 4. Опишите паттерн фабрика, приведите пример реализации

```
// aka abstract product
interface $ { int $(); }

// aka specific products
class A implements $ { @Override public int $() { return 0; } }
class B implements $ { @Override public int $() { return 1; } }
class C implements $ { @Override public int $() { return 2; } }

enum D { AA, BB, CC; }

// Factory producing specific products either with reflection or by hand
public class F {

    public static <T extends $> T get(Class<T> a)
        throws Exception { return a.newInstance(); }

    public static $ get(D a) { return switch (a) {
        case AA -> new A();
        case BB -> new B();
        case CC -> new C();
        default -> throw new RuntimeException("stub");
    }; }

    public static void main(String... $$) throws Exception {
        int a = get(A.class).$(),
            b = get(B.class).$(),
            c = get(C.class).$();

        assert a == get(D.AA).$() && a == 0;
        assert b == get(D.BB).$() && b == 1;
        assert c == get(D.CC).$() && c == 2;

        System.out.println("" +
            a + ' ' + b + ' ' + c
        ); // 0 1 2
    }
}
```

Опишите паттерн фабричных, приведите пример реализации

④

Предположим, что в приложении есть некоторый класс или интерфейс, у которого есть множество наследников. Несколько создают экземпляров определенного класса в зависимости от некоторого условия.

В качестве примера реализации рассмотрим автоматизацию кофейни, которая должна уметь готовить разные виды кофе.

① Создадим класс кофе и все его производные: американо, капучино, экспрессо и латте.

```
P  
O  
D  
U  
T  
E  
J  
B  
C  
K  
I  
Y  
K  
J  
A  
C  
C  
C  
public class Coffee {  
    public void grindCoffee() {  
        // ПЕРЕМАЛЫВАЕМ КОФЕ  
    }  
    public void makeCoffee() {  
        // ВЫЛАЕМ КОФЕ  
    }  
    public void pourIntoCup() {  
        // НАЛИВАЕМ В ЧАШКУ  
    }  
}
```

Методы, описывающие все процессы создания кофе.

! Все эти методы в двух классах-наследниках будут разными

```
public class Americano extends Coffee {}  
public class Cappuccino extends Coffee {}  
public class CaffeLatte extends Coffee {}  
public class Espresso extends Coffee {}
```

② Для удобства будем перечисление, в котором зададим все виды кофе:

```
public enum CoffeeType {  
    ESPRESSO,  
    AMERICANO,  
    CAFFE_LATTE,  
    CAPPUCINO  
}
```

Следующий класс, который наследует купчика экземпляр класса с зависимостями от мира:

```
public class SimpleCoffeeFactory {  
    public Coffee createCoffee (CoffeeType type) {  
        Coffee coffee = null;  
        switch (type) {  
            case AMERICANO: Клиент, который в зависимости от  
                типа кофе вызывает производит  
                экземпляр нужного класса:  
                coffee = new Americano (); присоединяется к имеющейся  
                (реализованной насыщенной) ветвью;  
                break;  
            case ESPRESSO:  
                coffee = new Espresso ();  
                break;  
            case CAPPUCCINO:  
                coffee = new Cappuccino ();  
                break;  
            case CAFFE_LATTE:  
                coffee = new CaffeLatte ();  
                break;  
        }  
        return coffee;  
    }  
}
```

④ Следующий класс работает с применением декомпозиции:

```
public class CoffeShop {  
    private final SimpleCoffeeFactory coffeeFactory;  
    public CoffeShop (SimpleCoffeeFactory coffeeFactory) {  
        this.coffeeFactory = coffeeFactory;  
    }  
    public Coffee orderCoffee (CoffeeType type) {  
    }
```

```
Coffee coffee = coffeeFactory.createCoffeeType();
coffee.grindCoffee();
coffee.makeCoffee();
coffee.pourIntoCup();
System.out.println("Вот ваше кофе!");
return coffee;
```

3

5

В чём удобство паттерна?

Мы бы могли перенести всю логику по созданию кофе в зависимости типа в класс CoffeeShop, но это очень неудобно и пришлось бы делать много switch-case-ов в одном месте. Класс бы превратился в очень большую, неприменимую и неудобную. Поэтому код получился более логичный и читаемый. Ещё плюсы:

- 1) При добавлении нового типа кофе класс CoffeeShop можно не менять, то есть код более поддерживаемый
- 2) За счёт такой структуризации кода очень удобно менять зациклические механизмы распределения.

5. Опишите паттерн стратегия, приведите пример реализации  
смотреть вопрос 6

```
abstract class OnDeleteStrategy { abstract void onDelete(); }

class CascadeOnDeleteStrategy extends OnDeleteStrategy
{ @Override public void onDelete() { /*smth*/ } }

class KeepOthersonDeleteStrategy extends OnDeleteStrategy
{ @Override public void onDelete() { /*another smth*/ } }

public class A {

    public static void main(String... $) {
        final OnDeleteStrategy strategyToBeUsedWhenDeletingSmth
            = new CascadeOnDeleteStrategy();
        // ...
        strategyToBeUsedWhenDeletingSmth.onDelete();
        // ...
    }
}
```

6. Опишите паттерн строитель, приведите пример реализации

## смотреть вопрос 12

```
import static java.lang.String.valueOf;

// smth buildable
class A {
    // some fields
    int a; int b; int c;

    A(int d, int e, int f)
    { a = d; b = e; c = f; }

    // for debug
    @Override public String toString() { return
        valueOf(a) + ' ' +
        valueOf(b) + ' ' +
        valueOf(c);
    }

    // builder
    static class B {
        int a = 0;
        int b = 0;
        int c = 0;

        // aka setters
        B a(int _a) { a = _a; return this; }
        B b(int _b) { b = _b; return this; }
        B c(int _c) { c = _c; return this; }

        // build
        A $() { return new A(a, b, c); }
    }
}

// smth using buildable through the builder
public class C {

    public static void main(String... $) {
        System.out.println(new A.B().a(0).b(1).c(2).$());
        // 0 1 2
        System.out.println(new A.B().a(6).b(7).c(8).$());
        // 6 7 8
    }
}
```

```
/** "Product" */
class Pizza {
    private String dough = "";
    private String sauce = "";
    private String topping = "";

    public void setDough(String dough) { this.dough = dough; }
    public void setSauce(String sauce) { this.sauce = sauce; }
    public void setTopping(String topping) { this.topping = topping; }
}

/** "Abstract Builder" */
abstract class PizzaBuilder {
    protected Pizza pizza;

    public Pizza getPizza() { return pizza; }
    public void createNewPizzaProduct() { pizza = new Pizza(); }

    public abstract void buildDough();
    public abstract void buildSauce();
    public abstract void buildTopping();
}

/** "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
    public void buildDough() { pizza.setDough("cross"); }
    public void buildSauce() { pizza.setSauce("mild"); }
    public void buildTopping() { pizza.setTopping("ham+pineapple"); }
}

/** "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
    public void buildDough() { pizza.setDough("pan baked"); }
    public void buildSauce() { pizza.setSauce("hot"); }
    public void buildTopping() { pizza.setTopping("pepperoni+salami"); }
}

/** "Director" */
class Waiter {
    private PizzaBuilder pizzaBuilder;

    public void setPizzaBuilder(PizzaBuilder pb) { pizzaBuilder = pb; }
    public Pizza getPizza() { return pizzaBuilder.getPizza(); }

    public void constructPizza() {
        pizzaBuilder.createNewPizzaProduct();
        pizzaBuilder.buildDough();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildTopping();
    }
}
```

```

/** "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
    public void buildDough() { pizza.setDough("cross"); }
    public void buildSauce() { pizza.setSauce("mild"); }
    public void buildTopping() { pizza.setTopping("ham+pineapple"); }
}

/** "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
    public void buildDough() { pizza.setDough("pan baked"); }
    public void buildSauce() { pizza.setSauce("hot"); }
    public void buildTopping() { pizza.setTopping("pepperoni+salami"); }
}

/** "Director" */
class Waiter {
    private PizzaBuilder pizzaBuilder;

    public void setPizzaBuilder(PizzaBuilder pb) { pizzaBuilder = pb; }
    public Pizza getPizza() { return pizzaBuilder.getPizza(); }

    public void constructPizza() {
        pizzaBuilder.createNewPizzaProduct();
        pizzaBuilder.buildDough();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildTopping();
    }
}

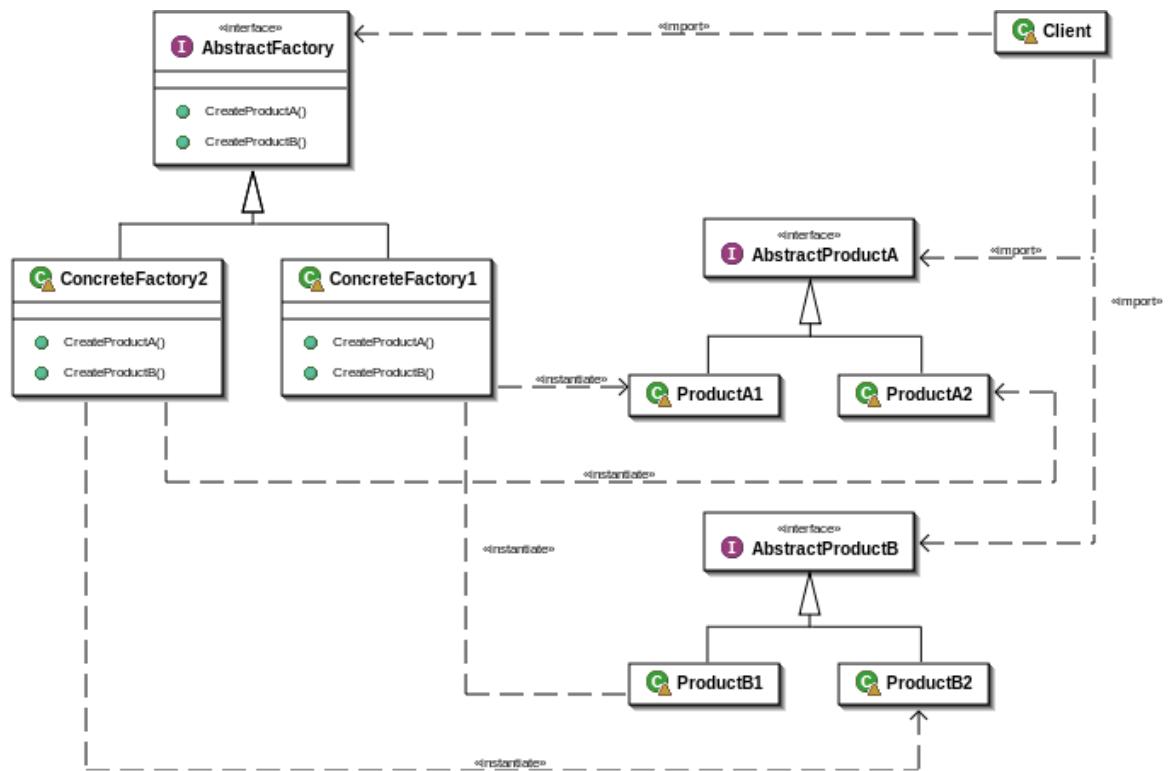
/** A customer ordering a pizza. */
public class BuilderExample {
    public static void main(String[] args) {
        Waiter waiter = new Waiter();
        PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
        waiter.setPizzaBuilder(hawaiianPizzaBuilder);
        waiter.constructPizza();

        Pizza pizza = waiter.getPizza();
    }
}

```

7. Опишите паттерн абстрактная фабрика, приведите пример реализации

**Абстрактная фабрика** (англ. *Abstract factory*) — порождающий шаблон проектирования, предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов. Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы (например, для оконного интерфейса он может создавать окна и кнопки). Затем пишутся [классы](#), реализующие этот интерфейс<sup>[2]</sup>.



## Плюсы [ [править](#) | [править код](#) ]

- изолирует конкретные классы;
- упрощает замену семейств продуктов;
- гарантирует сочетаемость продуктов.

## Минусы [ [править](#) | [править код](#) ]

- сложно добавить поддержку нового вида продуктов.

```
public class AbstractFactoryExample {

    public static void main(String[] args) {

        AbstractFactory factory1 = new ConcreteFactory1();
        Client client1 = new Client(factory1);
        client1.execute();

        AbstractFactory factory2 = new ConcreteFactory2();
        Client client2 = new Client(factory2);
        client2.execute();
    }
}

class Client {
    private AbstractProductA productA;
    private AbstractProductB productB;

    Client(AbstractFactory factory) {
        productA = factory.createProductA();
        productB = factory.createProductB();
    }

    void execute() {
        productB.interact(productA);
    }
}

interface AbstractFactory {
    AbstractProductA createProductA();
    AbstractProductB createProductB();
}

interface AbstractProductA {
    void interact(AbstractProductB b);
}

interface AbstractProductB {
    void interact(AbstractProductA a);
}

class ConcreteFactory1 implements AbstractFactory {

    @Override
    public AbstractProductA createProductA() {
        return new ProductA1();
    }
    @Override
    public AbstractProductB createProductB() {
        return new ProductB1();
    }
}
```

```

class ConcreteFactory2 implements AbstractFactory {
    @Override
    public AbstractProductA createProductA() {
        return new ProductA2();
    }
    @Override
    public AbstractProductB createProductB() {
        return new ProductB2();
    }
}

class ProductA1 implements AbstractProductA {
    @Override
    public void interact(AbstractProductB b) {
        System.out.println(this.getClass().getName() + " interacts with " + b.getClass().getName());
    }
}

class ProductB1 implements AbstractProductB {
    @Override
    public void interact(AbstractProductA a) {
        System.out.println(this.getClass().getName() + " interacts with " + a.getClass().getName());
    }
}

class ProductA2 implements AbstractProductA {
    @Override
    public void interact(AbstractProductB b) {
        System.out.println(this.getClass().getName() + " interacts with " + b.getClass().getName());
    }
}

class ProductB2 implements AbstractProductB {
    @Override
    public void interact(AbstractProductA a) {
        System.out.println(this.getClass().getName() + " interacts with " + a.getClass().getName());
    }
}

```

8. Опишите паттерн адаптер, его отличие от декоратора, приведите пример реализации

**Адаптер** (англ. *Adapter*) — структурный шаблон проектирования, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный [интерфейс](#). Другими словами — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

```
// файл Chief.java

public interface Chief {

    public Object makeBreakfast();
    public Object makeDinner();
    public Object makeSupper();

}

// файл Plumber.java

public class Plumber {

    public Object getPipe() {
        return new Object();
    }

    public Object getKey() {
        return new Object();
    }

    public Object getScrewDriver() {
        return new Object();
    }

}

// файл ChiefAdapter.java

public class ChiefAdapter implements Chief {

    private Plumber plumber = new Plumber();

    @Override
    public Object makeBreakfast() {
        return plumber.getKey();
    }

    @Override
    public Object makeDinner() {
        return plumber.getScrewDriver();
    }

    @Override
    public Object makeSupper() {
        return plumber.getPipe();
    }

}
```

```
// Файл ChiefAdapter.java

public class ChiefAdapter implements Chief {

    private Plumber plumber = new Plumber();

    @Override
    public Object makeBreakfast() {
        return plumber.getKey();
    }

    @Override
    public Object makeDinner() {
        return plumber.getScrewDriver();
    }

    @Override
    public Object makeSupper() {
        return plumber.getPipe();
    }

}

// Файл Client.java

public class Client {

    public static void main(String [] args) {
        Chief chief = new ChiefAdapter();

        Object key = chief.makeDinner();
    }

}
```

```
// Target
public interface Chief
{
    public Object makeBreakfast();
    public Object makeLunch();
    public Object makeDinner();
}

// Adaptee
public class Plumber
{
    public Object getScrewNut()
    { ... }
    public Object getPipe()
    { ... }
    public Object getGasket()
    { ... }
}

// Adapter
public class ChiefAdapter extends Plumber implements Chief
{
    public Object makeBreakfast()
    {
        return getGasket();
    }
    public Object makeLunch()
    {
        return getPipe();
    }
    public Object makeDinner()
    {
        return getScrewNut();
    }
}

// Client
public class Client
{
    public static void eat(Object dish)
    { ... }

    public static void main(String[] args)
    {
        Chief ch = new ChiefAdapter();
        Object dish = ch.makeBreakfast();
        eat(dish);
        dish = ch.makeLunch();
        eat(dish);
    }
}
```

```

// Adapter
public class ChiefAdapter extends Plumber implements Chief
{
    public Object makeBreakfast()
    {
        return getGasket();
    }
    public Object makeLunch()
    {
        return getPipe();
    }
    public Object makeDinner()
    {
        return getScrewNut();
    }
}

// Client
public class Client
{
    public static void eat(Object dish)
    { ... }

    public static void main(String[] args)
    {
        Chief ch = new ChiefAdapter();
        Object dish = ch.makeBreakfast();
        eat(dish);
        dish = ch.makeLunch();
        eat(dish);
        dish = ch.makeDinner();
        eat(dish);
        callAmbulance();
    }
}

```

*А ниче что тут (в примере выше ~~~^) вместо завтра получаем прокладку, вместо ланча трубы, а вместо ужина гайку? Вот клиент то порадуется... На следующий день все звезды у ресторана отнимут*

Попробуем пример

Давай посмотрим, как это будет выглядеть на примере:

```
1 public interface USB {  
2     void connectWithUsbCable();  
3 }  
4 }
```

Это наш интерфейс USB с единственным методом — вставить USB-кабель:

```
1 public class MemoryCard {  
2     public void insert() {  
3         System.out.println("Карта памяти успешно вставлена!");  
4     }  
5     public void copyData() {  
6         System.out.println("Данные скопированы на компьютер!");  
7     }  
8 }  
9 }  
10 }
```

Это наш класс, реализующий карту памяти. В нем уже есть 2 нужных нам метода, но вот беда: интерфейс USB он не реализует. Карту нельзя вставить в USB-разъем.

```
1 public class CardReader implements USB {  
2     private MemoryCard memoryCard;  
3     public CardReader(MemoryCard memoryCard) {  
4         this.memoryCard = memoryCard;  
5     }  
6     @Override  
7     public void connectWithUsbCable() {  
8         this.memoryCard.insert();  
9         this.memoryCard.copyData();  
10    }  
11 }
```

```
1 public class Main {  
2     public static void main(String[] args) {  
3         USB cardReader = new CardReader(new MemoryCard());  
4         cardReader.connectWithUsbCable();  
5     }  
6 }
```

Что же у нас в результате получилось?

9. Опишите паттерн декоратор, его отличие от адаптера, приведите пример реализации

**Декоратор** ([англ. Decorator](#)) — [структурный шаблон проектирования](#), предназначенный для динамического подключения дополнительного поведения к [объекту](#). Шаблон Декоратор предоставляет гибкую альтернативу практике создания [подклассов](#) с целью расширения функциональности.

**Адаптер - частный случай декоратора.** Отличие скорее в том, что декоратор может быть масштабнее (содержать в себе много сущностей, в то время как адаптор, как правило, реализует интерфейс для определенного типа). Прокси - противоположность первых двух, т.к. ее задача - быть прозрачной на стыке взаимодействия типов и не менять интерфейс.

```
public interface InterfaceComponent {
    void doOperation();
}

class MainComponent implements InterfaceComponent {

    @Override
    public void doOperation() {
        System.out.print("World!");
    }
}

abstract class Decorator implements InterfaceComponent {
    protected InterfaceComponent component;

    public Decorator (InterfaceComponent c) {
        component = c;
    }

    @Override
    public void doOperation() {
        component.doOperation();
    }

    public void newOperation() {
        System.out.println("Do Nothing");
    }
}

class DecoratorSpace extends Decorator {

    public DecoratorSpace(InterfaceComponent c) {
        super(c);
    }

    @Override
    public void doOperation() {
        System.out.print(" ");
        super.doOperation();
    }

    @Override
    public void newOperation() {
        System.out.println("New space operation");
    }
}
```

```

@Override
public void newOperation() {
    System.out.println("New space operation");
}
}

class DecoratorComma extends Decorator {

    public DecoratorComma(InterfaceComponent c) {
        super(c);
    }

    @Override
    public void doOperation() {
        System.out.print(",");
        super.doOperation();
    }

    @Override
    public void newOperation() {
        System.out.println("New comma operation");
    }
}

class DecoratorHello extends Decorator {

    public DecoratorHello(InterfaceComponent c) {
        super(c);
    }

    @Override
    public void doOperation() {
        System.out.print("Hello");
        super.doOperation();
    }

    @Override
    public void newOperation() {
        System.out.println("New hello operation");
    }
}

class Main {

    public static void main (String... s) {
        Decorator c = new DecoratorHello(new DecoratorComma(new DecoratorSpace(new MainComponent())));
        c.doOperation(); // Результатом выполнения программы "Hello, World!"
        c.newOperation(); // New hello operation
    }
}

```

## 10. Опишите паттерн фасад, приведите пример реализации

**Шаблон фасад (англ. *Facade*)** — структурный шаблон проектирования, позволяющий скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

```
/* Complex parts */

class CPU {
    public void freeze() {
        System.out.println("freeze");
    }

    public void jump(long position) {
        System.out.println("jump position = " + position);
    }

    public void execute() {
        System.out.println("execute");
    }
}

class Memory {
    public void load(long position, byte[] data) {
        System.out.println("load position = " + position + ", data = " + data);
    }
}

class HardDrive {
    public byte[] read(long lba, int size) {
        System.out.println("read lba = " + lba + ", size = " + size);
        return new byte[size];
    }
}

/* Facade */
```

```

class Computer {
    private final static long BOOT_ADDRESS = 1L;
    private final static long BOOT_SECTOR = 2L;
    private final static int SECTOR_SIZE = 3;

    private CPU cpu;
    private Memory memory;
    private HardDrive hardDrive;

    public Computer() {
        this.cpu = new CPU();
        this.memory = new Memory();
        this.hardDrive = new HardDrive();
    }

    public void startComputer() {
        cpu.freeze();
        memory.load(BOOT_ADDRESS, hardDrive.read(BOOT_SECTOR, SECTOR_SIZE));
        cpu.jump(BOOT_ADDRESS);
        cpu.execute();
    }
}

/* Client */

class Application {
    public static void main(String[] args) {
        Computer computer = new Computer();
        computer.startComputer();
    }
}

```

```

// one of possible Facade pattern implementations
// from the real world

// Facade hiding the true logic
public interface Context {
    // ...
    SharedPreferences getSharedPreferences();
    // ...
}

// Wrapper delegating all calls to the hidden implementation
class ContextWrapper implements Context {
    private Context delegate;
    // ...
    public ContextWrapper(Context a) { delegate = a; }
    //...
    @Override public SharedPreferences
    getSharedPreferences() { return delegate.getSharedPreferences(); }
}

// Implementation whose logic is needed to be hidden
class ContextImpl implements Context {
    private SharedPreferences sp;
    // ...
    @Override public SharedPreferences
    getSharedPreferences() { /*true logic*/ }
    // ...
}
// and the implementation is injected by reflection

```