Вопрос №1: В каких классах пакета java.io можно вызвать метод **long skip(long n)**?

Ответ:

Метод long skip(long n) можно вызвать в классах, которые наследуются от **InputStream** или **Reader**, так как он объявлен в этих абстрактных классах.

Теория:

InputStream (байтовые потоки):

Метод skip(long n) пропускает n байт в потоке.

**Примеры классов: FileInputStream, BufferedInputStream, DataInputStream**.

Reader (символьные потоки):

Метод skip(long n) пропускает n символов.

**Примеры классов: FileReader, BufferedReader, InputStreamReader.**

*InputStream input = new FileInputStream("file.txt");*

*input.skip(10); // Пропускаем 10 байт*

*Reader reader = new FileReader("file.txt");*

*reader.skip(5); // Пропускаем 5 символов*

Вопрос 1.1. Классы пакета java.io **для чтения/записи последовательности байтов:**

1. **InputStream** (абстрактный класс для чтения байтов):

* FileInputStream — чтение из файла.
* ByteArrayInputStream — чтение из массива байтов.
* BufferedInputStream — буферизованное чтение.

1. **OutputStream** (абстрактный класс для записи байтов):

* FileOutputStream — запись в файл.
* ByteArrayOutputStream — запись в массив байтов.
* BufferedOutputStream — буферизованная запись.

Вопрос 1.2 Классы пакета java.io, где можно вызвать метод void write(char[] buf)

Метод void **write**(char[] buf) предназначен для **записи массива символо**в (не байтов!). Он принадлежит к символьным потокам (character streams), которые работают с данными в формате Unicode.

**Writer** и его подклассы:

* FileWriter
* BufferedWriter
* CharArrayWriter
* OutputStreamWriter

Вопрос №2: Какие значения элементов и в каком порядке будут находиться в коллекции после выполнения кода?

*Код:*

*Set<Integer> set = new LinkedHashSet<>();*

*set.add(4); set.add(5); set.add(2); set.add(3); set.add(5); set.add(1);*

*set.remove(4);*

Ответ:

Элементы: **[5, 2, 3, 1]** в таком порядке.

Теория:

**LinkedHashSet** сохраняет порядок добавления элементов, но не допускает дубликатов.

Вопрос 2.1

*Queue<Integer> queue = new LinkedList<>();*

*queue.add(4); queue.add(3); queue.add(2); queue.add(1);*

*queue.remove();*

*queue.add(2);*

**remove()** → удаляется голова очереди (4) → [3, 2, 1]

**Ответ:** [3, 2, 1, 2] (порядок от головы к хвосту)

Вопрос 2.2 Deque

Состояние Deque после выполнения операций:

*Deque<Integer> deque = new ArrayDeque<>();*

*deque.add(3); // [3]*

*deque.addLast(1); // [3, 1]*

*deque.addFirst(4); // [4, 3, 1]*

*deque.add(2); // [4, 3, 1, 2]*

*deque.removeLast(); // [4, 3, 1]*

*deque.addFirst(5); // [5, 4, 3, 1]*

Результат: [5, 4, 3, 1] (порядок от головы к хвосту).

**Deque (двусторонняя очередь)** поддерживает операции с обоих концов.

Вопрос №2. Шаблоны проектирования

1. Singleton (Одиночка)

Гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к нему.

1. Factory Method (Фабричный метод)

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать.

1. Abstract Factory (Абстрактная фабрика)

Предоставляет интерфейс для создания семейств связанных объектов без указания их конкретных классов.

1. Builder (Строитель)

Позволяет создавать сложные объекты пошагово, используя один и тот же процесс строительства.

1. Prototype (Прототип)

Позволяет копировать существующие объекты без зависимости от их классов.

1. Adapter (Адаптер)

Преобразует интерфейс одного класса в интерфейс, ожидаемый клиентом.

1. Decorator (Декоратор)

Динамически добавляет объекту новые обязанности, оборачивая его в объект-декоратор.

1. Proxy (Прокси)

Позволяет подставлять вместо реальных объектов специальные объекты-заменители для контроля доступа.

1. Observer (Наблюдатель)

Определяет зависимость "один-ко-многим" между объектами, чтобы при изменении состояния одного объекта все зависящие от него оповещались.

1. Strategy (Стратегия)

Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми.

1. Command (Команда)

Инкапсулирует запрос как объект, позволяя параметризовать клиенты с разными запросами.

1. Iterator (Итератор)

Предоставляет способ последовательного доступа к элементам составного объекта без раскрытия его внутренней структуры.

1. Template Method (Шаблонный метод)

Определяет скелет алгоритма, перекладывая ответственность за некоторые шаги на подклассы.

1. Facade (Фасад)

Предоставляет простой интерфейс к сложной системе классов.

1. State (Состояние)

Позволяет объекту изменять свое поведение при изменении внутреннего состояния.

Вопрос №3: Что напечатает фрагмент кода, если метод delLast() удаляет последний символ?

Код:

*Stream.of("helium", "neon", "argon", "krypton", "xenon", "radon")*

*.filter(s -> s.length() != 6)*

*.map(s -> s.delLast())*

*.skip(2)*

*.sorted()*

*.forEachOrdered(System.out::print);*

**Ответ: kryptoradoxeno**

Теория:

Шаги обработки Stream API:

Исходный список: ["helium", "neon", "argon", "krypton", "xenon", "radon"]

**Фильтр length != 6:**

"helium" (6) → удаляется

Результат: ["neon", "argon", "krypton", "xenon", "radon"]

Удаление последнего символа (delLast()):

Результат: ["neo", "argo", "krypto", "xeno", "rado"]

Пропуск первых 2 элементов (**skip(2)**):

Остается: ["krypto", "xeno", "rado"]

Сортировка (**sorted()**):

Лексикографический порядок: ["krypto", "rado", "xeno"]

Вывод (**forEachOrdered**):

Печатается слитно: kryptoradoxeno

Вопрос №4: Какой **функциональный интерфейс** соответствует лямбда-выражению?

*(String s) -> s.isEmpty()*

Ответ:

Predicate<String>, так как выражение принимает String и возвращает boolean.

*(String s) -> s.hashCode()*

Ответ: Function<String, Integer>

*(String s) -> System.err.print(s)*

Ответ: Consumer<String>

Теория:

Функциональные интерфейсы в Java:

**Predicate<T>**: проверяет условие (boolean test(T t)).

**Function<T, R>:** преобразует T в R (R apply(T t)).

**Consumer<T>:** принимает T, ничего не возвращает (void accept(T t)).

**Supplier<T>:** не принимает аргументов, возвращает T (T get()).

Проверка:

(String s) -> s.isEmpty():

Вход: String, выход: boolean → подходит Predicate<String>.

Вопрос №5: Какой **класс** используется для получения датаграмм от клиента при использовании **протокола UDP?**

Ответ:

**DatagramSocket** (для получения/отправки датаграмм) и **DatagramPacket** (для хранения данных).

Теория:

**UDP в Java:**

**DatagramSocket:** создает сокет для обмена датаграммами.

**DatagramPacket:** хранит данные (байтовый массив).

Пример сервера:

*DatagramSocket socket = new DatagramSocket(1234); // Порт 1234*

*byte[] buffer = new byte[1024];*

*DatagramPacket packet = new DatagramPacket(buffer, buffer.length);*

*socket.receive(packet); // Получаем данные*

*String message = new String(packet.getData(), 0, packet.getLength());*

**TCP в Java**

Протокол TCP (в отличие от UDP) не использует датаграммы, потому что он работает с потоковыми соединениями (установлением соединения, гарантированной доставкой и контролем ошибок). TCP работает с потоками байтов (InputStream/OutputStream).

Для работы с TCP в Java используются классы:

**ServerSocket** — ожидает подключения клиентов (серверная часть).

**Socket** — обеспечивает двустороннее соединение (клиент и сервер).

Вопрос №6. Правильный порядок инструкций для подключения к базе данных и выполнения запроса

1. **Connection conn = DriverManager.getConnection(...);**

**DriverManager** — класс, который управляет подключением к БД.

**getConnection()** — возвращает объект Connection, представляющий соединение с БД.

Обязательный шаг: Без соединения дальнейшие операции невозможны.

Почему не Driver dm = DriverManager.getDriver(...)?

В современных JDBC-драйверах (начиная с JDBC 4.0) явная регистрация драйвера не требуется — он загружается автоматически.

DriverManager.getConnection() сам находит подходящий драйвер по URL.

1. **Statement st = conn.** **prepareStatement(...);**

В стандартном JDBC без параметров используется **createStatement()**. Statement нужен для выполнения SQL-запросов без параметров (например, простые SELECT).

**PreparedStatement** (для параметризованных запросов). **prepareStatement()** используется для запросов с параметрами (например, WHERE id = ?).

1. **ResultSet rs = st.executeQuery(...);**

**executeQuery()** — выполняет SQL-запрос (обычно SELECT) и возвращает ResultSet — таблицу с результатами.

Для запросов, изменяющих данные (INSERT, UPDATE), используется **executeUpdate().**

prepareStatement() создает объект, но не выполняет запрос. Для выполнения нужен отдельный вызов executeQuery().

1. **if (rs.next()) String name = rs.getString(...);**

**rs.next()** — перемещает курсор на следующую строку в ResultSet. Возвращает false, если строк больше нет.

**rs.getString("name")** — извлекает значение столбца name из текущей строки.

Почему не rs.nextString(...)?

Метода nextString() не существует в JDBC! Корректные методы: **getString(), getInt()**, и т.д.

// 1. Установка соединения

// 2. Создание Statement(запрос)

// 3. Выполнение запроса

// 4. Обработка результатов

Вопрос №7. Потоки и переходы

1. **NEW** (Создан)

Описание:

Поток создан (через new Thread()), но еще не запущен (метод start() не вызван).

Пример:

*Thread thread = new Thread(() -> System.out.println("Hello"));*

*System.out.println(thread.getState()); // NEW*

**Переход** в другое состояние:

→ RUNNABLE после вызова thread.start().

2. **RUNNABLE** (Готов к выполнению)

Описание:

Поток готов к выполнению и ждет выделения CPU. Включает два подсостояния:

**Ready**: Ожидает в очереди планировщика потоков.

**Running**: Выполняется на процессоре.

Пример:

*thread.start();*

*System.out.println(thread.getState()); // RUNNABLE*

**Переход** в другие состояния:

→ BLOCKED при попытке войти в synchronized блок.

→ WAITING/TIMED\_WAITING при вызове wait(), sleep(), join().

→ TERMINATED после завершения run().

3. **BLOCKED** (Заблокирован)

Описание:

Поток заблокирован, ожидая монитор (synchronized блок/метод), который удерживается другим потоком.

Пример:

*Object lock = new Object();*

*Thread thread1 = new Thread(() -> {*

*synchronized (lock) {*

*try { Thread.sleep(1000); } catch (InterruptedException e) {}*

*}*

*});*

*Thread thread2 = new Thread(() -> {*

*synchronized (lock) { // Будет BLOCKED, пока thread1 не освободит lock*

*System.out.println("Получил lock");*

*}*

*});*

*thread1.start();*

*thread2.start();*

*Thread.sleep(100); // Даем thread1 захватить lock*

*System.out.println(thread2.getState()); // BLOCKED*

**Переход** в другое состояние:

→ RUNNABLE после получения монитора.

4. **WAITING** (Ожидание)

Описание:

Поток переходит в это состояние при вызове:

Object.wait() (без таймаута),

Thread.join() (без таймаута),

LockSupport.park().

Пример:

*Thread thread = new Thread(() -> {*

*synchronized (lock) {*

*try { lock.wait(); } catch (InterruptedException e) {}*

*}*

*});*

*thread.start();*

*Thread.sleep(100);*

*System.out.println(thread.getState()); // WAITING*

**Переход** в другое состояние:

→ RUNNABLE после вызова notify()/notifyAll() или LockSupport.unpark().

5. **TIMED\_WAITING** (Ожидание с таймаутом)

Описание:

Поток ожидает ограниченное время при вызове:

Thread.sleep(millis),

Object.wait(timeout),

Thread.join(timeout),

LockSupport.parkNanos(),

LockSupport.parkUntil().

Пример:

*Thread thread = new Thread(() -> {*

*try { Thread.sleep(1000); } catch (InterruptedException e) {}*

*});*

*thread.start();*

*Thread.sleep(100);*

*System.out.println(thread.getState()); // TIMED\_WAITING*

**Переход** в другое состояние:

→ RUNNABLE после истечения таймаута или прерывания.

6. TERMINATED (Завершен)

Описание:

Поток завершил выполнение метода run().

Пример:

**Thread thread = new Thread(() -> System.out.println("Done"));**

**thread.start();**

**thread.join(); // Ждем завершения**

**System.out.println(thread.getState()); // TERMINATED**

**Переход** в другое состояние:

**Невозможен**. Поток нельзя перезапустить.

**Диаграмма состояний потока**

**NEW**

**↓**

**RUNNABLE ←→ {BLOCKED, WAITING, TIMED\_WAITING}**

**↓**

**TERMINATED**

Ключевые моменты

* Из RUNNABLE в BLOCKED

Только при конкуренции за synchronized. Для ReentrantLock используется WAITING.

* Разница между WAITING и TIMED\_WAITING

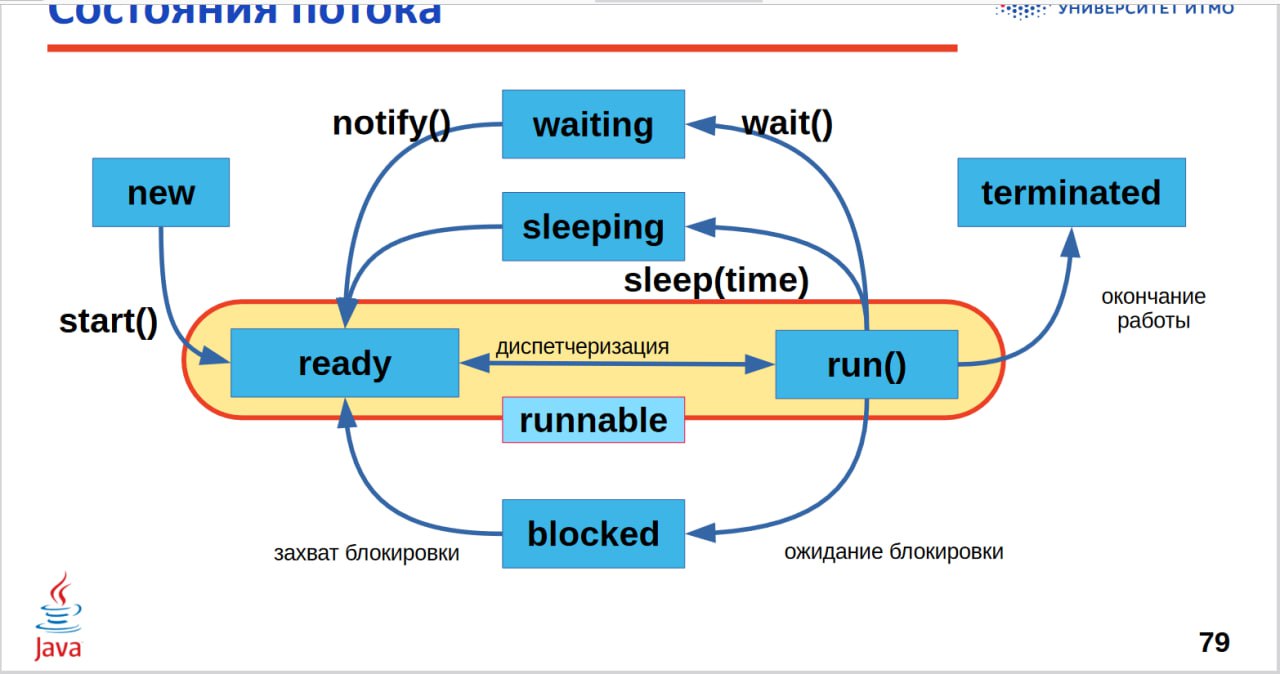
WAITING — бессрочное ожидание (wait(), join() без таймаута).

TIMED\_WAITING — ожидание с ограничением времени (sleep(), wait(1000)).

* Нельзя перейти из TERMINATED обратно в RUNNABLE

Завершенный поток нельзя перезапустить — нужно создать новый.

* Для точности всегда проверяйте текущее состояние через **Thread.getState().**



Вопрос №8. Строка в файле. Локализация

1. **Создание локали (Locale)**

*Locale locale = new Locale("de\_CH");*

Теория:

Locale определяет региональные настройки (язык и страну).

"de\_CH" — немецкий язык (de) в Швейцарии (CH).

Используется для локализации ресурсов (например, файлов .properties).

Зачем:

Чтобы система знала, какие файлы искать (например, hello\_de\_CH.properties вместо hello.properties).

2. Загрузка **ResourceBundle**

*ResourceBundle bundle = ResourceBundle.getBundle("hello", locale);*

Теория:

ResourceBundle загружает локализованные строки из файлов .properties.

Ищет файлы в порядке:

1. hello\_de\_CH.properties (точное совпадение локали),
2. hello\_de.properties (только язык),
3. hello.properties (файл по умолчанию).

Если ключ не найден в hello\_de\_CH.properties, используется родительский файл (hello.properties).

Зачем:

Для получения строк, соответствующих переданной локали.

3. Получение строки ("hello")

*String hello = bundle.getString("hello");*

Теория:

getString(key) возвращает значение для ключа key из файла .properties.

Если ключ отсутствует, выбрасывает MissingResourceException.

Зачем:

Чтобы получить локализованное значение для ключа "hello".

**Для вывода ‘halo world’ в hello\_de\_CH.properties достаточно написать: hello = halo. (ключ world будет подхвачен из hello.properties).**

Вопрос №9. Впишите недостающий элемент в код обработки события.

*JTextField w = new JTextField("Return of the Jedi");*

*w.* **addActionListener** *(new ActionListener() {*

*public void actionPerformed(ActionEvent ev) {*

*this.setForeground(Color.ORANGE);*

*}*

*});*

Правильный ответ:

**addActionListener**

* **JTextField** — это компонент из библиотеки Swing, который позволяет пользователю вводить однострочный текст.
* **ActionListener** — это интерфейс, реализующий метод actionPerformed, который вызывается при определённом действии (например, когда пользователь нажимает Enter внутри текстового поля).
* **addActionListener()** — метод, с помощью которого регистрируют обработчик событий у компонента, в данном случае у JTextField.

Вопрос 9.1

*JFrame jcomp = new JFrame("The Force Awakens");*

*jcomp.****addWindowListener****(new WindowAdapter() {*

*public void windowActivated(WindowEvent ev) {*

*jcomp.setBackground(Color.MAGENTA);*

*}*

*});*

* Слушатели событий (Listeners) — интерфейсы, такие как **WindowListener, ActionListener**.
* Адаптеры (Adapters) — абстрактные классы, реализующие эти интерфейсы с пустыми методами (например, **WindowAdapter**).
* JFrame — главное окно приложения.
* **WindowAdapter** — упрощает обработку событий окна (реализует WindowListener).
* **windowActivated()** — метод, вызываемый при активации окна (получении фокуса).
* Метод **addWindowListener()** регистрирует слушатель для событий окна (закрытие, активация, минимизация и т.д.). Без него события не будут обрабатываться.

Вопрос 9.2.

*JFrame component = new JFrame("The Force Awakens");*

*component.****addWindowListener****(new WindowAdapter() {*

*public void windowActivated(****WindowEvent*** *event) {*

*component.setForeground(Color.MAGENTA); // Исправлено this на component*

*}*

*});*

WindowEvent: Тип параметра для методов обработки событий окна, содержит информацию о событии (например, источник)

Вопрос 9.3

*JCheckBox c = new JCheckBox("The Force Awakens");*

*c.****addItemListener****(new* ***ItemAdapter()*** *{*

*public void itemStateChanged(****ItemEvent*** *ev) {*

*c.setForeground(Color.MAGENTA); // Исправлено this на c*

*}*

*});*

1. Обработка событий **JCheckBox**

JCheckBox — компонент Swing для выбора (галочка).

События изменения состояния обрабатываются через ItemListener.

2. **ItemAdapter**

Абстрактный класс, реализующий ItemListener с пустыми методами.

Упрощает код — не нужно реализовывать все методы интерфейса.

Метод itemStateChanged() вызывается при изменении состояния (например, при установке/снятии галочки).

3. **ItemEvent**

Содержит информацию о событии:

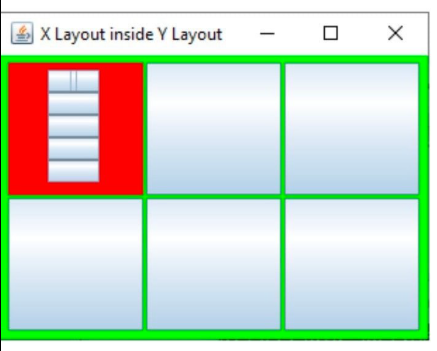
ev.getStateChange() — возвращает ItemEvent.SELECTED или ItemEvent.DESELECTED.

ev.getItem() — возвращает связанный объект.

4. **addItemListener**

Метод, регистрирующий слушатель событий изменения состояния. Без него клики на радиокнопку не будут обрабатываться.

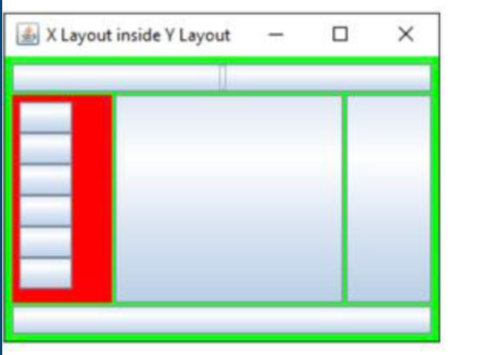
Вопрос 10. Менеджер компоновки



* **Layout Manager (менеджер компоновки)** — отвечает за размещение компонентов внутри контейнера в Java (например, JPanel, JFrame).
* **GridLayout** — менеджер, который располагает компоненты в виде сетки (таблицы), с заданным числом строк и столбцов. Все ячейки одинакового размера.

В показанном на скриншоте примере:

* **Зеленая панель** содержит 6 прямоугольников (компонентов), выровненных по сетке — по 3 в ряд и 2 в колонке.
* Это явно указывает на использование GridLayout(2, 3) или GridLayout(3, 2) — главное, что используется **GridLayout**.
* Внутри одной ячейки (в левом верхнем углу) находится **красная панель** с вертикально размещёнными элементами — это признак **BoxLayout** с вертикальной осью (BoxLayout.Y\_AXIS)



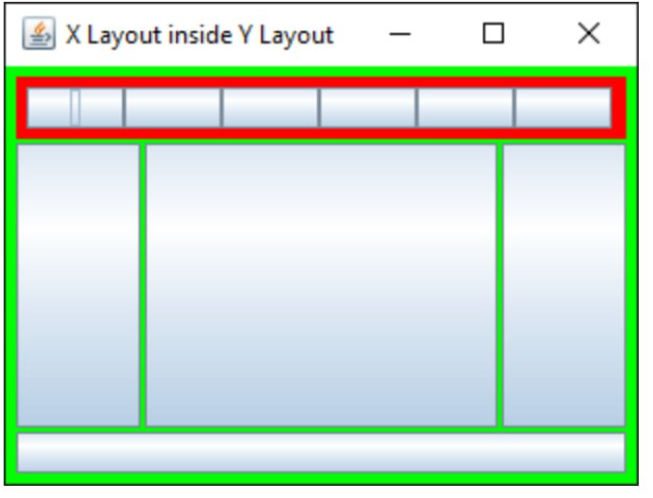
На зеленом: **BorderLayout.**  Зеленая панель содержит 5 возможных регионов:

* слева — красная панель
* справа — пустой прямоугольник
* в центре — большая светлая область
* снизу — узкая панель

 Это характерно именно для **BorderLayout**, который располагает компоненты в следующих направлениях:

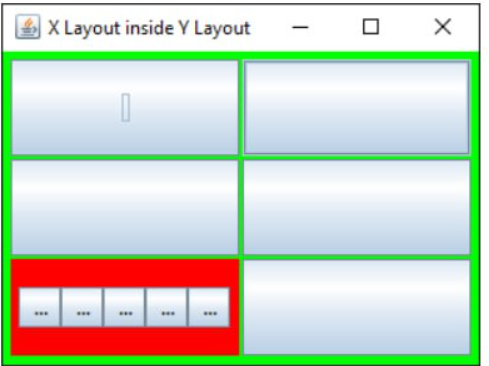
* BorderLayout.NORTH (сверху)
* BorderLayout.SOUTH (снизу)
* BorderLayout.WEST (слева)
* BorderLayout.EAST (справа)
* BorderLayout.CENTER (центр — занимает всё оставшееся пространство)

На красном: **BoxLayout.Y\_AXIS.** Красная панель (внутри зеленой, слева) содержит несколько узких горизонтальных компонентов, выстроенных **в столбик**, друг под другом. Это классическая работа **BoxLayout по вертикальной оси**.



На зеленом: **BorderLayout**

На красном: **BoxLayout.X\_AXIS**



На зеленом: **GridLayout**

На красном: **BoxLayout.X\_AXIS**