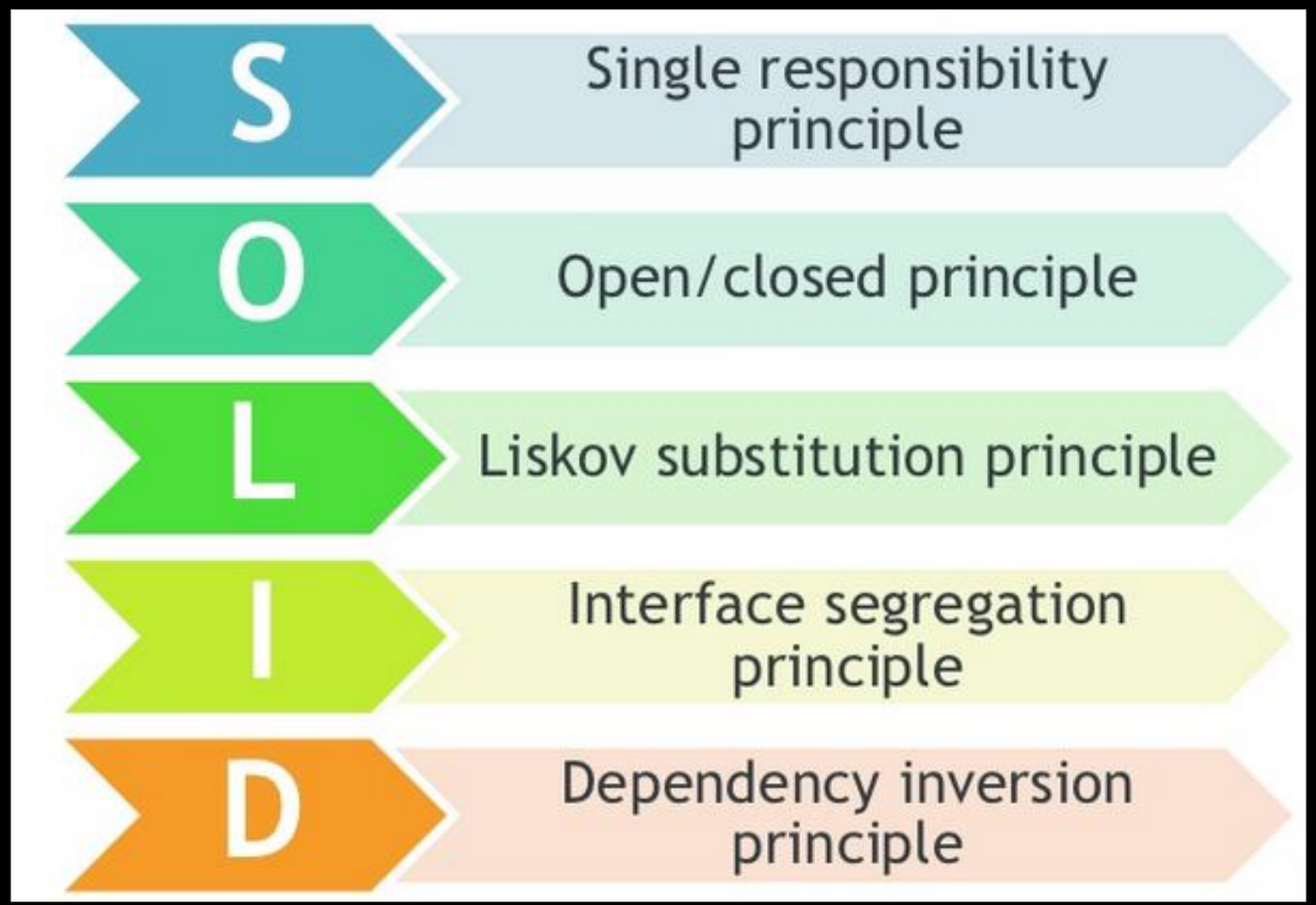
Разбор домашних. Потоки и потоки.

Практика 39 - Айрат Хасьянов

Минута Паттерна

SOLID - review



Разбор кода, Stream API

Что происходит?

```
Map<String, Integer> words = new HashMap<>();
        wordSequence.stream()
                .filter(element -> Pattern.compile("[а-яА-Я]
+").matcher(element).matches())
        .map(String::toLowerCase)
        .forEach(element -> {
            if (words.containsKey(element))
                words.put(element, words.get(element) + 1);
            else
                words.put(element, 1);
    });
```

Что делаем?

```
Stream.generate(() -> random.nextInt(100))
                .limit(30)
                .filter(digit -> digit >= 0 && digit < 12)
                .forEach( digit ->
map.putIfAbsent(random.nextInt(100), digit));
```

Какая задача?

```
while (words.size() > 1){
   words.stream().forEach(o -> {
        if(o.length() > 0)symb.add(o.charAt(0));});
   Map characters = new HashMap();
   symb.stream()
           .collect(Collectors.groupingBy(key -> key))
           .forEach((k,v) -> characters.put(k, v.size()));
   symb.clear();

   result += characters.entrySet()
           .stream()
           .max(Map.Entry.comparingByValue()).toString().charAt(9);

   word = word.stream()
          .map(o -> o.substring(1, o.length()))
          .filter(o -> o.length() > 0)
          .collect(Collectors.toList());
}
```

Что происходит?

```
list.stream()
        .map(w->w.length()<4?
w.toUpperCase():w.contains("e")&w.contains("в")?w:"")
                .forEach(System.out::println);
```

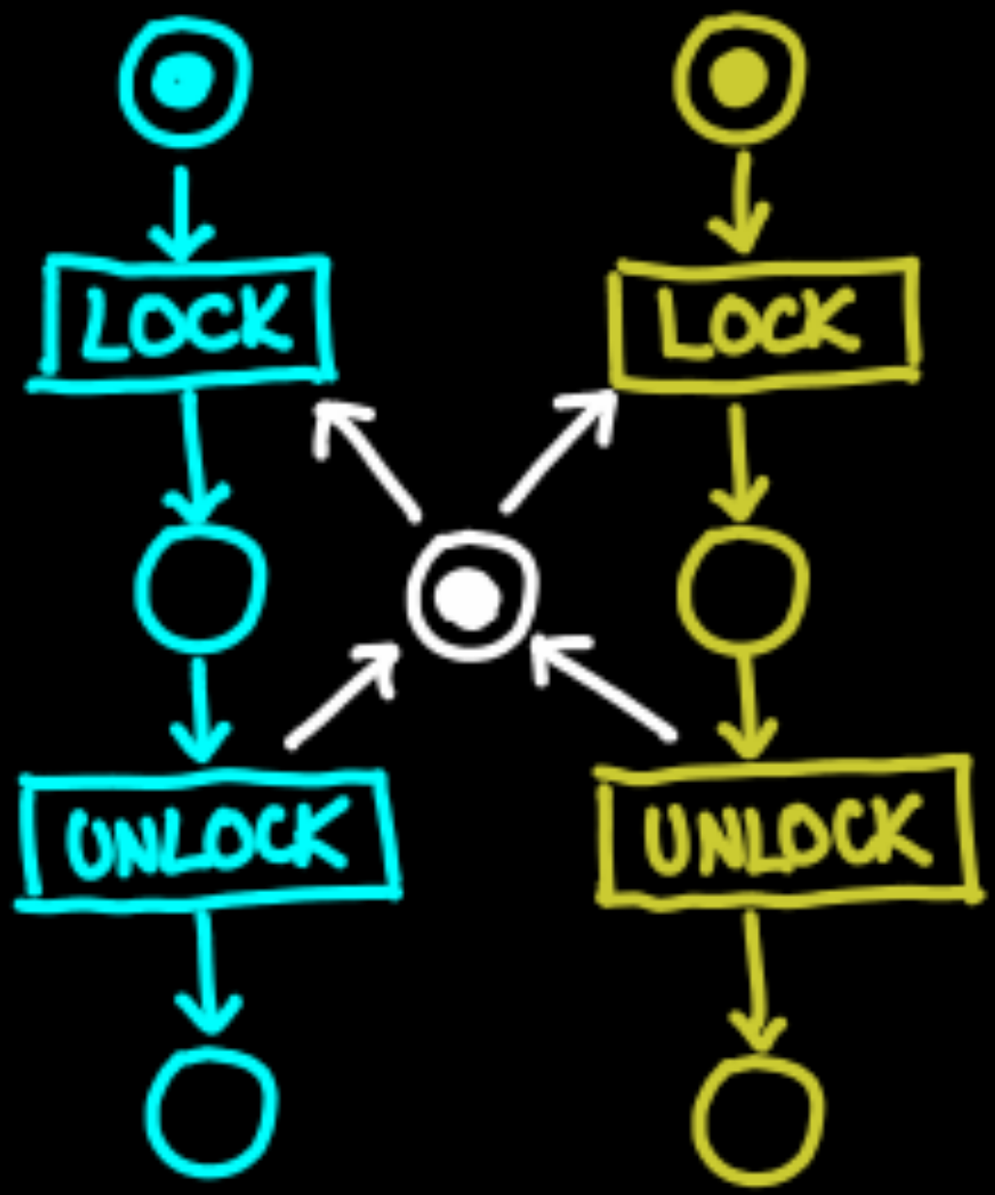
Настраиваемый предикат для фильтрации

```
    Predicate<Employee> isRich = (employee -> employee.getMoney()>1000);
    Predicate<Employee> isHappy = (employee ->
employee.getState().equals("happy"));
    Predicate<Employee> isCompetent = (employee ->
employee.getKnowledge().equals("high"));
    Predicate<Employee> predicate;
      switch (condition){
          case "money":
              predicate = isRich;
              break;
          case "state":
              predicate = isHappy;
              break;
          case "knowledge":
              predicate = isCompetent;
              break;
          default:
              predicate = (employee -> false);
      }
    return predicate;}}
```

Использовать легко:

```
String[] strings =list.stream()
    .filter(switcher("state").and(switcher("money"))
            .or(switcher("knowledge").and(switcher("money"))
            .or(switcher("state").negate().and(switcher("knowledge")))))
    .map(Employee::getName)
    .toArray(String[]::new);
```

Concurrency



Пример

```
public class LiftOff implements Runnable{
  protected int countDown = 10;
  private static int taskCount = 0;
  private final int id = taskCount++;
  public LiftOff(){}
  public LiftOff(int countDown){
   this.countDown = countDown;
}
 public String status(){
    return "#"+id+"("+ (countDown>0?countDown+"), ":"Lift Off!!!)");
  }
  public void run(){
    while(countDown-->0){
      System.out.print(status());
      Thread.yield();
    } } }
```

Пример выполнения

```
public class MainThread {
  public static void main(String[] args {
      System.out.println("Waiting for lift off...");
      Thread t = new Thread(new LiftOff());
      t.start();
    }
  }
```

Waiting for lift off... #0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1), #0(Lift Off!!!)

Несколько потоков

```
public class MoreThreads {
  public static void main(String[] args){
     System.out.println("Waiting for lift offs...");
     Thread t;
     for(int i=0; i<5; i++){
          t = new Thread(new LiftOff());
          t.start();
} } }
```

Waiting for lift offs... #1(9), #1(8), #1(7), #1(6), #1(5), #1(4), #1(3), #1(2), #1(1), #1(Lift Off!!!) #2(9), #2(8), #2(7), #2(6), #2(5), #2(4), #2(3), #2(2), #2(1), #2(Lift Off!!!) #3(9), #3(8), #3(7), #3(6), #3(5), #3(4), #3(3), #3(2), #3(1), #3(Lift Off!!!) #4(9), #4(8), #4(7), #4(6), #4(5), #4(4), #0(9), #4(3), #0(8), #4(2), #0(7), #4(1), #0(6), #0(5), #4(Lift Off!!!)#0(4), #0(3), #0(2), #0(1), #0(Lift Off!!!)

Исполнители

- Рекомендуется использовать вместо прямого создания объектов Thread;
- Ускоряют работу с потоками за счет использования пулов:
  - **CachedThreadPool** — новый поток для каждой задачи;
  - **FixedThreadPool** — фиксированное количество потоков;
  - **SingleThreadExecutor** — одновременно исполняет единственный поток; Если такому исполнителю передается более одной задачи, они ставятся в очередь.
- Вызов метода shutdown() исполнителя прекращает передачу ему новых задач.

CachedThreadPool

```
public class CachedThreadPool {
    public static void main(String[] args){
        ExecutorService es = Executors.newCachedThreadPool();
        for(int i = 0; i<5; i++){
            es.execute(new LiftOff());
        }
        es.shutdown();
}}
```

#0(9), #3(9), #2(9), #1(9), #1(8), #2(8), #1(7), #2(7), #1(6), #2(6), #1(5), #2(5), #1(4), #2(4), #1(3), #2(3), #1(2), #2(2), #1(1), #2(1), #1(Lift Off!!!)#2(Lift Off!!!)#0(8), #0(7), #0(6), #3(8), #3(7), #4(9), #0(5), #0(4), #3(6), #4(8), #0(3), #3(5), #4(7), #0(2), #0(1), #3(4), #4(6), #0(Lift Off!!!)#3(3), #4(5), #4(4), #3(2), #4(3), #4(2), #3(1), #4(1), #4(Lift Off!!!)#3(Lift Off!!!)

FixedThreadPool

```
public class FixedThreadPool {
    public static void main(String[] args){
    ExecutorService es = Executors.newFixedThreadPool(2);
    for(int i = 0; i<5; i++){
        es.execute(new LiftOff());
    }
    es.shutdown();
}}
```

#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #1(9), #1(8), #1(7), #1(6), #1(5), #1(4),
#1(3), #0(3), #1(2), #1(1), #0(2), #0(1), #0(Lift Off!!!)#1(Lift Off!!!) #2(9),
#3(9), #2(8), #2(7), #3(8), #3(7), #2(6), #2(5), #3(6), #3(5), #2(4), #2(3), #3(4),
#3(3), #2(2), #2(1), #3(2), #3(1), #2(Lift Off!!!)#4(9), #4(8), #4(7), #4(6), #4(5)
#4(4), #4(3), #4(2), #4(1), #4(Lift Off!!!)#3(Lift Off!!!)

SingleThreadExecutor

```
public class SingleThreadExecutor {
    public static void main(String[] args){
        ExecutorService es = Executors.newSingleThreadExecutor();
        for(int i = 0; i<5; i++){
            es.execute(new LiftOff());
        }
        es.shutdown();
}}
```

#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1), #0(Lift Off!!!)
#1(9), #1(8), #1(7), #1(6), #1(5), #1(4), #1(3), #1(2), #1(1), #1(Lift Off!!!)
#2(9), #2(8), #2(7), #2(6), #2(5), #2(4), #2(3), #2(2), #2(1), #2(Lift Off!!!)
#3(9), #3(8), #3(7), #3(6), #3(5), #3(4), #3(3), #3(2), #3(1), #3(Lift Off!!!)
#4(9), #4(8), #4(7), #4(6), #4(5), #4(4), #4(3), #4(2), #4(1), #4(Lift Off!!!)

Callable - получение значений

- Метод **submit()** класса ExecutorService возвращает объект класса Future, параметризованный типом результата;
- Для объекта **Future** можно вызывать методы:
  - **isDone()** - выяснить завершена ли задача;
  - **get()** - получить результат после завершения задачи;
- Вызов **Future<T>.get()** блокируется до завершения задачи;
- Перегруженный метод **Executors.callable()** получает Runnable и выдает Callable;
- **ExecutorService** содержит методы для выполнения коллекций объектов Callable.

Пример с Callable

```
import java.util.concurrent.Callable;
public class ResultiveTask implements Callable<String> {
    private static int count = 0;
    private final int id = count++;
    @Override
    public String call() throws Exception {
        return "Result from "+id;
    }
}
```

```
public class CallableDemo {
    private static int SIZE = 5;
    public static void main(String[] args) {
        ExecutorService es = Executors.newCachedThreadPool();
        ArrayList<Future<String>> results = new ArrayList<Future<String>>()
        for(int i = 0; i<SIZE; i++){
            results.add(es.submit(new ResultiveTask()));
        }
        es.shutdown();
        for(Future<String> f:results){
        try{
            System.out.println(f.get());
        }catch(InterruptedException e){
            System.err.println(e);
        }catch(ExecutionException e){
            System.err.println(e);
}}}}
```

Result from 0
Result from 1
Result from 2
Result from 3
Result from 4

Ожидание

- Поток можно перевести в режим ожидания двумя способами:
  - Вызов **Thread.yeld()** - позволяет другим потокам перехватить управление; Это всего лишь подсказка для планировщика, ни в коем случае не директива!
  - Вызов **TimeUnit.MILLISECONDS.sleep(msecs)** — позволяет приостановить поток на заданное время.
- Вызов метода **sleep()** может привести к исключению InterruptedException;
- Так как исключения, возникающие в потоках не распространяются в main, их надо обрабатывать ***индивидуально для каждого потока***.

```
class SleepyTask extends LiftOff {
    public void run() {
        try {
            while(countDown-->0){
                System.out.println(status());
                TimeUnit.SECONDS.sleep(1);
            }
        } catch (InterruptedException ie){
            System.err.println("Sleep Interrupted!");
        }
    }
    public static void main(String[] args) {
        ExecutorService es = Executors.newCachedThreadPool();
        for(int i = 0; i<5; i++)
            es.execute(new SleepyTask());
        es.shutdown();
    }
}
```

sleep

Приоритет

- В пакете JDK предусмотрено 10 уровней приоритетов, однако, в операционной системе это число обычно иное;
- Переносимость обеспечивается использованием трех констант:
  - Thread.MIN_PRIORITY
  - Thread.NORM_PRIORITY
  - Thread.MAX_PRIORITY
- Thread.setPriority() устанавливает приоритет потока;
- Thread.getPriority() возвращает приоритет потока.

```
public class SimplePriorities implements Runnable {
    private int priority, countDown = 5;
    private volatile double d;
    public SimplePriorities(int priority) {
        this.priority = priority;
    }
    public String toString() { return Thread.currentThread() + ": " + countDown; }
    public void run() {
        Thread.currentThread().setPriority(priority);
        while(true) {
            for(int i = 1; i < 100000; i++) {
                d += (Math.PI + Math.E) / (double)i;
                if(i % 1000 == 0) Thread.yield();
            }
            System.out.println(this);
            if(--countDown == 0) return;
        }
    }
    public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
        exec.execute(new SimplePriorities(Thread.MIN_PRIORITY));
    exec.execute(new SimplePriorities(Thread.MAX_PRIORITY));
    exec.shutdown(); }}
```

Результат работы

Thread[pool-1-thread-6,10,main]: 5
Thread[pool-1-thread-6,10,main]: 4
Thread[pool-1-thread-6,10,main]: 3
Thread[pool-1-thread-6,10,main]: 2
Thread[pool-1-thread-6,10,main]: 1
Thread[pool-1-thread-3,1,main]: 5
Thread[pool-1-thread-2,1,main]: 5
Thread[pool-1-thread-1,1,main]: 5
Thread[pool-1-thread-5,1,main]: 5
Thread[pool-1-thread-4,1,main]: 5 ...

Потоки-демоны

- **Демон** — это служебный поток, который работает в фоновом режиме;
- Программа не завершается пока хотя бы один поток-не-демон продолжает работать;
- Программа может завершиться даже если продолжают работу потоки-демоны;
- Демоны - **служебные потоки**, как только завершится последний поток программ, демоны будут остановлены.
- Демоном поток делает вызов метода **setDeamon()** перед запуском потока.

```
public class SimpleDaemons implements Runnable {
    public void run() {
        try {
            while(true) {
                TimeUnit.MILLISECONDS.sleep(100);
                System.out.println(Thread.currentThread() + " " +
                this.getClass().getSimpleName());
            }
        } catch(InterruptedException e) {
              System.out.print("sleep() interrupted");
        }
    }
    public static void main(String[] args) throws Exception {
        for(int i = 0; i < 10; i++) {
            Thread daemon = new Thread(new SimpleDaemons());
            daemon.setDaemon(true);
            daemon.start();
        }
        System.out.println("All daemons started");
        TimeUnit.MILLISECONDS.sleep(175);
}}
```

Результат работы демонов

All daemons started
Thread[Thread-3,5,main] SimpleDaemons
Thread[Thread-4,5,main] SimpleDaemons
Thread[Thread-1,5,main] SimpleDaemons
Thread[Thread-5,5,main] SimpleDaemons
Thread[Thread-0,5,main] SimpleDaemons
Thread[Thread-6,5,main] SimpleDaemons
Thread[Thread-2,5,main] SimpleDaemons
Thread[Thread-7,5,main] SimpleDaemons
Thread[Thread-8,5,main] SimpleDaemons
Thread[Thread-9,5,main] SimpleDaemons

Фабрика демонов

```
public class DaemonThreadFactory implements ThreadFactory{
    @Override
    public Thread newThread(Runnable arg0) {
        Thread t = new Thread(arg0);
        t.setDaemon(true);
        return t;
    } }
```

- Теперь при создании Исполнителя мы можем передать ему нашу фабрику демонов:
  - Executor.newCachedThreadPool(new DeamonFactory());
- Все потоки, порожденные этой фабрикой будут демонами!

Вопросы

- Чем демоны отличаются от обычных потоков?
- Сколько способов порождения потоков Вы знаете?
- Как приостановить поток?
- Что такое Исполнитель?
- Сколько приоритетов определено в JDK? В Windows в Solaris?
- За счет чего достигается переносимость кода при использовании приоритетов?
- В чем отличие интерфейсов Callable и Runnable?

Задачи

1. Реализовать сложение матриц используя параллельное выполнение двумя способами: используя Stream API, и без
2. Аналогично для умножения матриц

3. Текст зашифрован шифром простой подстановки с одним ключом. Написать параллельный алгоритм подбора ключа. Для проверки текста использовать словарь. Решить задачу без Stream API и используя Stream API

Что за паттерн?

```
public class Blanket implements IComponent {
    private String name;
    private double time;
    private String new_description;
    private ArrayList<IComponent> components = new ArrayList<>();
 public Blanket(){}
 public Blankets(String name,String new_description, double time){
     this.name = name;
     this.time = time;
     this.new_description = new_description;
 }
 public void add(IComponent component){}
.....
```

Composite

Что за паттерн?

```
public class SortedArray extends ArrayWrapper {
    public SortedArray(IArray array){
        super(array);
    }
    @Override
    public String[] getarray() {
        String [] arr = array.getarray();
        Arrays.sort(arr);
        return arr;
    }
}
```

Decorator

Что за паттерн?

```
public abstract class Document {

    public final void openDoc(String name) {
        Document doc = loadFile(name);
        doc.initChecker();
        doc.showDocument();
    }
    protected void loadFile(String name);
    ....
}
```

Template Method

Что за паттерн?

```
public class Deputat{
    private Aide aide;
   .......
     public void vote(){
       aide.vote();
     };
     public void propose(){
       aide.propose();
     };
}
```

Delegate

Что за паттерн?

```
class Safe {
    int lvl;
    String weapon;
    public Safe(int lvl, String weapon) {
        this.lvl = lvl;
        this.weapon = weapon;
    }
    public int getLvl() {
        return lvl;
    }
    public String getWeapon() {
        return weapon;
    }
}
```

```
class File {
    Safe safe;

    public Safe getSafe() {
        return safe;
    }

    public void setSafe(Safe safe) {
        this.safe = safe;
    }
}
```

Memento

Домашнее задание

1. Найдите максимум в массиве используя 5 потоков;

2. Напишите поток счётчик, который запускает новый счётчик из своего потока. Запустите 10 потоков и просуммируйте в конце результат работы всех счетчиков. Сделайте при помощи yield и wait так, чтобы сумма отличалась от 100. Перепишите код безопасным образом, чтобы он никогда не ошибался

3. Найдите число от 0 до 10000000, которое имеет наибольшее число делителей; Используйте потоки. Подумайте, как разбить эту задачу на задачи меньшего размера! Используйте ThreadPool.

4. Напишите многопоточное приложение, которое вычисляет значение Pi через вероятностный процесс. Вероятность того, что точка в единичном квадрате окажется в четверти круга, вписанной в этот квадрат = pi/4.

5. (*) напишите сервер, который умеет возвращать пользователям файлы из заданной дериктории.